Connected Information Management

Axel Rauschmayer

München 2010

Connected Information Management

Axel Rauschmayer

Dissertation
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig-Maximilians-Universität München
zur Erlangung des Grades Doctor rerum naturalium (Dr. rer. nat.)

vorgelegt von
Axel Rauschmayer

München, 2010-01-29

Erstgutachter: Martin Wirsing
Zweitgutachter: Marcus Spies
Auswärtiger Gutachter: Don Batory
Tag der mündlichen Prüfung: 2010-02-18

Short contents

Contents

Abstract

Society is currently inundated with more information than ever, making efficient management a necessity. Alas, most of current information management suffers from several levels of disconnectedness: Applications partition data into segregated islands, small notes don't fit into traditional application categories, navigating the data is different for each kind of data; data is either available at a certain computer or only online, but rarely both. *Connected information management* (CoIM) is an approach to information management that avoids these ways of disconnectedness. The core idea of CoIM is to keep all information in a central repository, with generic means for organization such as tagging. The heterogeneity of data is taken into account by offering specialized editors.

The central repository eliminates the islands of application-specific data and is formally grounded by a *CoIM model*. The foundation for structured data is an RDF repository. The *RDF editing meta-model* (REMM) enables form-based editing of this data, similar to database applications such as MS access. Further kinds of data are supported by extending RDF, as follows. Wiki text is stored as RDF and can both contain structured text and be combined with structured data. Files are also supported by the CoIM model and are kept externally. Notes can be quickly captured and annotated with meta-data. Generic means for organization and navigation apply to all kinds of data. Ubiquitous availability of data is ensured via two CoIM implementations, the web application HYENA/Web and the desktop application HYENA/Eclipse. All data can be synchronized between these applications. The applications were used to validate the CoIM ideas.

Zusammenfassung

Unsere Gesellschaft sieht sich zur Zeit mit einer immer größer werdenden Informationsflut konfrontiert. Daraus erwächst die Notwendigkeit, Information effizient zu verwalten. Als Hindernis hierzu erweist sich, dass in der heutigen Informationsverwaltung die Daten auf mehrfache Art getrennt sind: Daten werden durch spezifische Anwendungen in schwer überbrückbare Inseln aufgeteilt; kleine Notizen passen in keine der traditionellen Anwendungskategorien; das Navigieren ist bei jeder Datenart anders; Daten sind entweder nur auf bestimmten Rechnern oder nur online verfügbar, aber selten beides. *Connected information management* (CoIM) ist ein Ansatz, Informationen so zu verwalten, dass die erwähnten Trennungen vermieden werden. Die Kernidee des CoIM ist, die Informationen in einer zentralen Datenbank zu halten, unterstützt von generischen Organisationsmechanismen wie Tagging. Der Heterogenität der Informationen wird mit spezialisierten Editoren Rechnung getragen.

Die zentrale Datenhaltung vermeidet die Inselnstruktur anwendungsspezifischer Daten und wird durch ein *CoIM-Modell* formal unterstützt. Das Fundament für strukturierte Daten ist eine RDF-Datenbank. Das *RDF editing meta-model* (REMM) sorgt dafür, dass diese Daten Formular-basiert editiert werden können, ähnlich einer Datenbankanwendung wie MS Access. Weitere Arten von Daten werden wie folgt unterstützt, als Erweiterungen von RDF. Wiki-Text wird im RDF abgelegt und kann sowohl unstrukturierten Text enthalten, als auch mit strukturierten Daten vermischt werden. Dateien werden ebenfalls vom CoIM-Modell integriert und extern aufbewahrt. Notizen lassen sich schnell eingeben und mit Meta-Daten versehen. Generische Organisations- und Navigationskonzepte sind auf alle Daten anwendbar. Die universelle Verfügbarkeit

der Daten wird durch zwei CoIM-Implementierungen ermöglicht, von denen die eine, HYENA/Web, eine Web-Anwendung, die andere, HYENA/Eclipse, eine Desktop-Anwendung ist. Zwischen diesen Anwendungen können die Daten synchronisiert werden. Beide Anwendungen wurden auch genützt, um die CoIM-Ideen zu validieren.

Chapter 1

Introduction: Connected information management

Contents

1.1 Overview

In our society, the information each individual has to manage digitally continues to grow. Current tools are inadequate, because data exists in segregated islands and cannot be accessed and organized freely, in a manner that transcends these islands. *Connected information management* (CoIM) is a new approach to information management that meets these challenges. It proposes an architecture where all information is kept in a central repository that offers generic means of organization. Specific aspects of the information in a repository are supported by specialized editors. The benefits of this approach are that the separations between the information islands disappear. Thus, information can be managed in a uniform manner, for example if a topic cuts across kinds of data. The specialized editors guarantee that unique aspects of some kinds of information can also be managed.

The core idea of a central repository has been formalized as a model. The capabilities of CoIM are further extended by other contributions such as synchronizing repositories to ensure ubiquitous availability of data (online and offline), organization schemes, navigation mechanisms and a framework that allows one to extend the given facilities to support new kinds of data.

1.2 Connected information management

Society is currently inundated with more information than ever: stored in wikis, blogs, news feeds, web browser bookmarks, emails, and files such as PDFs, spreadsheet documents, and text documents. When working with this information, many areas of disconnectedness come up which prevent efficient management of it. First, specific kinds of information are handled well by dedicated applications, but each kind exists in separation. If a topic crosses these kinds, then it is usually impossible to bring together the relevant data, be it via linking or otherwise. An example is a workshop, where one would like to collect contact information of the participants, emails that were written during planning, the papers that were presented during the workshop, and the web pages of the workshop. The second way of disconnectedness concerns online and offline availability of data: With the increasing sophistication of web applications, much data is now available online, ubiquitously anywhere a web browser can be used to connect to the Internet. But, online data and offline data are still largely disjoint, meaning that some of it is only available when online, some of it is only available when one has access to one's offline data. There is no integrated way of making the same data available on the web and offline. For example, wikis are usually online-only, while most graphics programs are desktop applications. The third kind of disconnectedness exists between users. With desktop applications, sharing data is either impossible or inefficient. When collaboratively editing a document, it is often sent back and forth between collaborators. Exchanging document fragments is even more complicated. Most web applications do have ways of sharing and collaboratively editing data. But not all of them allow one to do so, and there is no solution that crosses web application boundaries. Fourth, there is disconnectedness between small unstructured notes and other kinds of data. Small notes are usually stored in applications that were not made for handling them, such as word processors and wikis. These programs tend to introduce arbitrary groupings (into documents or pages) and make recall difficult. Often, using a computer to capture a note is not convenient enough and people use physical media such as post-it notes. Then the divide between the notes and the rest of the (electronically stored) data is even greater. Fifth and last, the process of finding information suffers from fragmentation. Every kind of data brings its own way of navigation: Files are usually browsed hierarchically, web pages are accessed via keyword search, online galleries can find photos by location, and electronic calendars can return all events in a given month. What is missing is a solution that integrates all these ways of recall. For example, the geo-location where a file has been created can help to distinguish between files created at work and files created at home, which might be crucial information for finding it.

Connected information management (CoIM) and its implementation HYENA (an akronym of *Hypergraph Editor and Navigator*) set out to eliminate these kinds of disconnectedness. The foundation of CoIM is RDF. Being graph-based, relational and semi-structured, RDF is ideally suited for distributed, heterogeneous linked data. CoIM enhances RDF with a simple type system and the means for form-based editing, so that it can be used in a manner similar to database systems such as MS Access or Filemaker. An additional enhancement is the integration of unstructured data such as files and unstructured text. Such text is expressed in a special markup language called *Wikked* that is a hybrid of free-from wiki markup and the formal LaTeX syntax. Wikked plays an important role in CoIM as it is used for integrating, collating, and annotating data. Offline and online availability is ensured by two versions of HYENA, a web application and a desktop version based on an Eclipse plugin. All data that HYENA manages can

be synchronized between both versions. The web application has multi-user support and authentication. Collaboration is further helped by synchronization, which can be performed between peers. This dissertation also explores the requirements of quick note taking and what prevents users from doing it electronically. HYENA supports note taking by allowing quick capture, piling instead of filing, simple addition of meta-data and easy migration of note data. CoIM's multi-paradigm search integrates view-based navigation, keyword search and other filtering criteria across all supported data. It is based on a formal model and offers users various ways of retrieving their data. The final contribution of CoIM is a framework that provides important data management building blocks so that one can quickly experiment with new kinds of data and new kinds of navigation. The disconnectedness between HYENA's RDF repositories and the external, mainly file-based, world is bridged by an infrastructure for importing and exporting files.

Partial solutions for the problems mentioned above exist, but none of them have the necessary breadth of features: RDF editors support database-like handling of RDF, but don't have the free-form data entry and ease of annotation of HYENA's wiki markup. Semantic wikis are wikis with support for structured data in RDF; they mix structured and unstructured data. Alas, the dominance of their wiki user interface does not make them a good platform for general-purpose applications. CoIM's navigation model and synchronization also goes beyond their capabilities. Desktop applications with web companions and web applications with an offline mode solve the problem of online and offline availability of data, but only for one particular kind of data.

1.3 Hyena, a platform for connected information management

There are two versions of HYENA (Fig. 1.1): A desktop version called HYENA/Eclipse is implemented as an Eclipse plugin (Fig. 1.2). A web application called HYENA/Web is implemented with the Google Web Toolkit (Fig. 1.3). It supports multi-user access and collaboration. Data is grouped into *projects* that can be synchronized between HYENA/Eclipse and HYENA/Web. The idea is that one can work on one's data offline and then publish it as needed. But the web version has feature parity with the desktop version, so using HYENA in a web browser is not a compromise. As a side benefit, the offline data is a complete backup of the online data.

1.4 The structure of this dissertation

As there are many chapters, some of them relatively short, the author decided to group them into *parts* (meta-chapters, if you will). Each part begins with a summary of its contents. The contributions of this dissertation (Fig. 1.4) start in part II; to understand them, the reader should have at least basic knowledge of RDF (which is explained in Chap. 2). There are the following parts:

I. Background: provides background information on the concepts and technologies involved in connected information management: RDF is the data format that CoIM uses for structured data. *Linked data* is a collection of best practices and protocols to access and share RDF in a distributed manner. *Folksonomies* are used

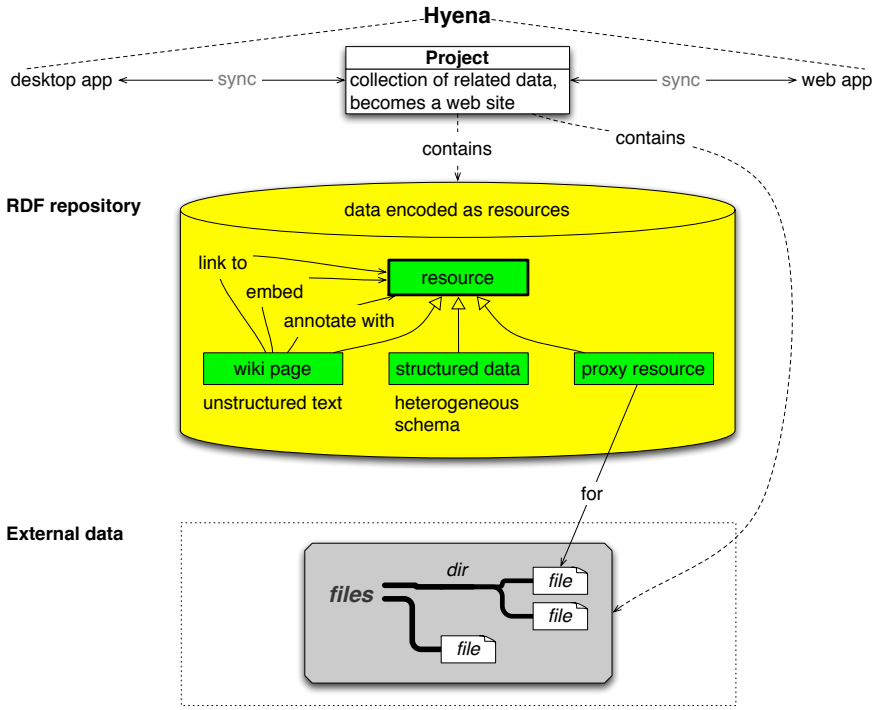


Figure 1.1: There are two versions of HYENA, a desktop version and a web application. Data is managed in *projects* and synchronized between the two. The dominant construct of a project is an RDF repository which is used to manage all of the data. Data that does not fit into the repository is kept externally and considered unstructured to HYENA. Currently only files are handled this way.

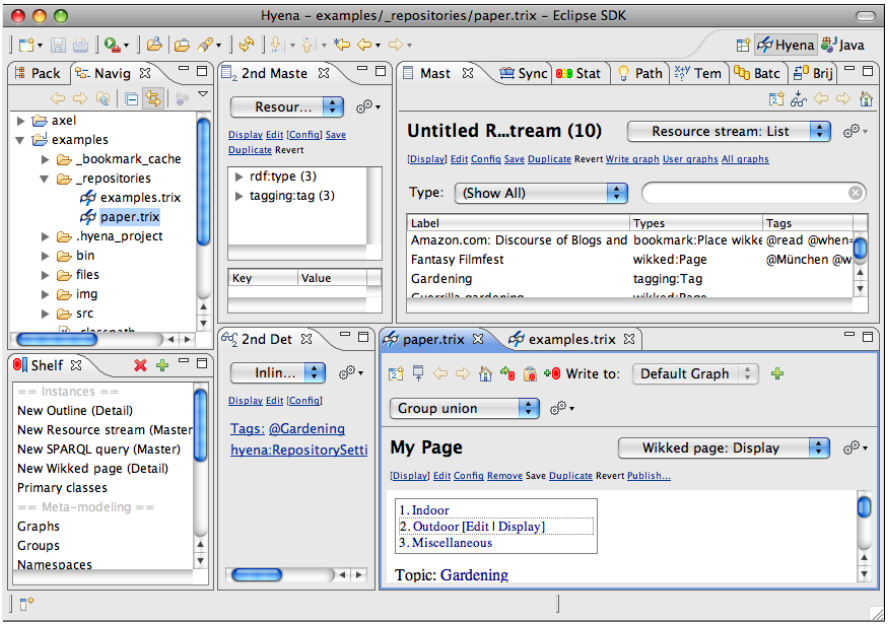


Figure 1.2: HYENA/Eclipse. The left column gives access to the projects and their files and gives access to common RDF management operations. Top center and right are "master" views for listing RDF resources. Bottom center and right are "detail" views for editing individual resources.

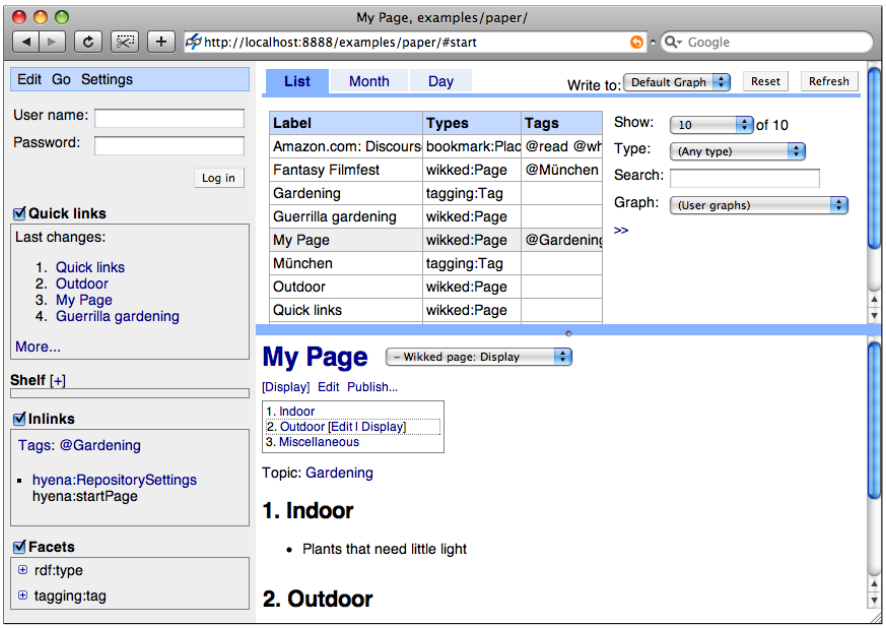


Figure 1.3: HYENA/Web. The left column contains a menu bar, login text fields, and several widgets with context information. The top right area is a "master" view that lists RDF resources. The bottom right area is a "detail" view for editing individual resources.

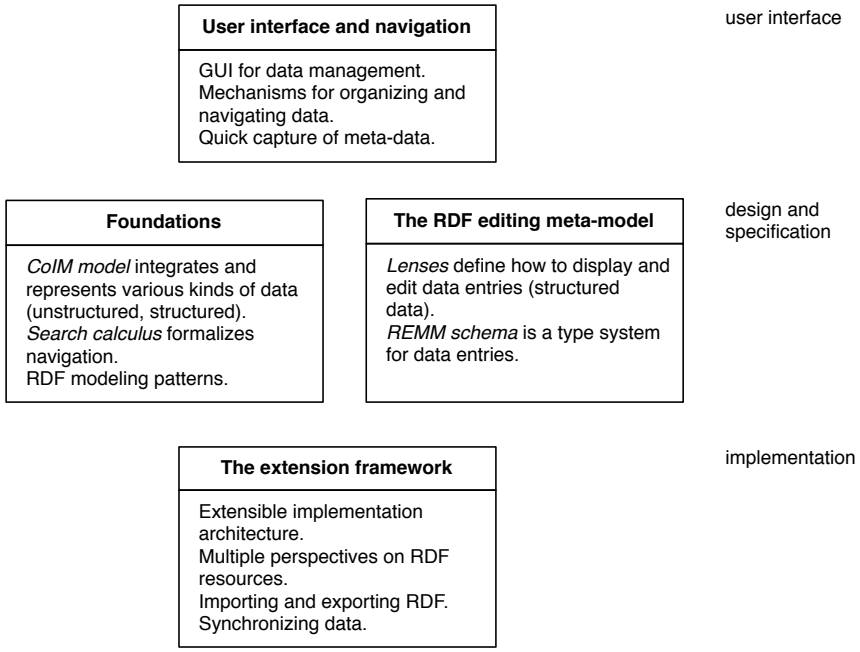


Figure 1.4: The diagram above shows the main contributions of this document. They can roughly be partitioned into user interface, design and specification, and implementation.

by CoIM to organize data. Schema and ontology languages support the editing of structured data.

II. User interface and navigation: This part gives an overview of the HYENA user interface and describes its foundations such as tagging, faceted navigation and assisted querying. CoIM supports *multi-paradigm search*, the integration of several search approaches such as faceted navigation and keyword search.

III. Foundations: A model for CoIM describes how to integrate different kinds of data. The *search calculus* is a formal foundation for multi-paradigm search. *Wikked*, a formal wiki markup language that is part of CoIM, is explained in detail. *Templates* define a meta-syntax on top of Wikked, so that RDF data can be flexibly displayed using Wikked markup. The part finishes by describing RDF patterns.

IV. The RDF editing meta-model (REMM): The canonical way of editing used by database management systems such as MS Access and Filemaker is based on forms. RDF has two obstacles for implementing this kind of editing: First, its most capable schema language, OWL, which has been created for distributed knowledge representation, is unintuitive for crucial aspects of data editing. Second, forms depend on order, making using them for the relational RDF more difficult than, for example, the tree-based XML. REMM meets both challenges; it defines a simple type system derived from OWL on which editing is based and introduces ways of specifying how to project ordered data from RDF and how to edit that data.

V. The extension framework: If supporting a new vocabulary demands more than HYENA's generic tools can provide then one can extend HYENA in Java. This part explains the programming framework that allows one to do so. It also explains how HYENA implements import and export of external, often file-based, data, which has some challenging aspects. Managing the different ways in which the same resource can be interpreted is also challenging and described in detail. The part concludes by explaining how HYENA synchronizes projects between installations: Files are synchronized in a straightforward manner and then the principles of file synchronization are extended to RDF synchronization, with resource granularity.

VI. Related work: describes how systems similar to HYENA approach the problems tackled by CoIM and how the solutions compare. Topics include navigation mechanisms, RDF editing (which is challenging to edit in forms, in contrast to, say, XML), hypertext, information management, data synchronization, semantic annotation of text, synchronization and versioning.

VII. Evaluation, summary, and future research: This part investigates the requirements of small-grained note taking and of integrating structured and unstructured data. It is shown how HYENA's constructs can be combined to fulfill those requirements. A user study evaluates how HYENA is used in practice. A survey among wiki users examines what makes wikis appealing and how they can be evolved without losing their appeal. The dissertation ends with a summary and future research.

1.5 Running example

Shenzi is a researcher writing a paper, together with two co-authors. To that end, she would like to manage the following information:

- Citations of relevant publications stored as BibTeX entries.
- PDF files of papers.
- Photos of posters she has taken at conferences.
- Local copies of papers that have been published as HTML.
- Other bookmarks related to the paper.
- Notes and ideas.
- Todos, each assigned to a particular co-author.
- Tags for grouping any of the data above by topic.

Shenzi would like to keep all of this data in one place and "take it with her", so that it is available everywhere she goes, both when she's online and when she's not connected to the Internet. The data should also be easy to back up and to edit, collaboratively with her co-authors. The CoIM contributions (Fig. 1.4) are applied to this example in the following manner:

- User interface and navigation: Faceted navigation is used to access todos by co-author. Tagging is used for grouping by topic.

- Foundations: make it possible to keep all the data mentioned above in a single project. Notes are stored as wiki pages. They can link to structured data such as citations or bookmarks. Or they can *embed* it. This means that such data appears inside the wiki page when displaying it.
- RDF editing meta-model (REMM): Allows one to declaratively specify how to edit citations and bookmarks. REMM specifications can also be used for embedding.
- The extension framework: This part is most interesting if one intends to extend HYENA with new functionality. Pertinent to the example, it also explains how to import data (such as citations and bookmarks) and how to synchronize projects. The latter feature allows Shenzi to work on the data offline and then publish it to a HYENA server so that all co-authors can see it.

Part I

Background

This part provides background information for subsequent parts. First, the *Resource Description Format* RDF is explained. It is a standard for data management and exchange which is used by CoIM to handle structured data. *Linked data* is a collection of best practices that help with making deployments of RDF data self-describing and easy to discover. A chapter on folksonomies and ontologies presents two ways of representing knowledge, each with different expressiveness and complexity. CoIM uses folksonomy-based meta-data to organize data. Finally, a chapter on schema and ontology languages introduces the *RDF schema language* RDFS and the RDF-based *Web Ontology Language* OWL. OWL will later become the foundation of a simple type system to support RDF editing. The *Ontology Definition Metamodel* is a standard for integrating OWL (and other ontology languages) with UML. It is not currently used by CoIM, but provides a wider perspective on the field of ontology languages. It might also be used to support UML-based creation of schemas in the future.

Chapter 2

Data modeling with RDF

Contents

2.1 Overview

The RDF data format is one of the foundations of Connected Information Management (CoIM). It is used as a universal repository for all structured data. This chapter begins by collecting requirements for a universal storage format: Its schema needs to be flexible; it needs standardized symbols to universally express concepts such as people, locations, and languages; it needs to be composable—combining data from different sources must be seamless even if some of the entities they describe are the same; links between entities must be expressible, and it needs a standardized exchange format so that external tools can be used. Given these requirements, the deficiencies of the RDF competitors XML, relational databases and "No SQL" stores are pointed out. Then RDF is briefly explained and it is shown how it fulfills the requirements.

The remainder of the chapter is a thorough overview of RDF. Apart from the basic constructs and how to use them, it introduces schema languages, rules, and the SPARQL query language. The chapter finishes by giving examples of useful RDF vocabularies and RDF applications. RDF is also the data format of the *semantic web*. Within the semantic web community, there are two factions: One that uses RDF to represent knowledge and one that uses RDF to store data. Where the former faction uses ontologies, the latter faction uses schemas. This chapter describes RDF from the vantage point of data modeling.

2.2 Finding a data format for structured data

The general scenario of CoIM is one where data from many different sources is consolidated in a single repository. This results in the following requirements:

- Flexible schema: It should be easy to add data to entities, be it to annotate them, be it to extend them with data that was not foreseen by the schema. This kind of flexibility also helps with schema evolution, because entities can be migrated to a new schema incrementally.

- Globally unique symbols: Symbols are needed to express concepts such as locations or languages in a globally unique way. This ensures that it is easy to find all data relevant to a given concept, even if it comes from different sources. As any source can introduce new symbols, there should be a way to make symbols *self-describing*; that is, a mechanism to express what a symbol means. Sometimes, two parties might introduce different symbols for the same concept. Thus, one must be able to declare that those symbols mean the same thing.

- Composability: It must be simple to combine data from different sources and one source must be able to add data to entities from another source. One application of this feature is a user adding annotations to data imported from an external source.

- Links between entities: When organizing entities, it helps if one does not just have a list of them, but can also cross-reference them.

- Standardized exchange format: If one wants to apply external tools to the data one manages, there has to be a way to get data in and out of a repository.

The next sections take a look at how XML, relational databases, and "No SQL" stores fare in light of these requirements. Then a brief introduction to RDF is given and it is examined how RDF fulfills the requirements.

2.2.1 XML

One of the great features of XML is that one can easily model new data domains: When one creates new tags, one goes beyond simple data storage towards modeling relationships between things. Furthermore, XML has a standardized exchange format, making it easy to exchange data. Lastly, it works well for marking up unstructured data.

On the flip side, XML is not ideal for semi-structured data. XML is difficult to enter manually (it has been called having a "write-only syntax"). Whitespace handling is complicated, because it some situation you want to honor whitespace, while in others honoring it can make accessing data difficult. XML is bad at handling binary data. The overhead of the tags becomes an issue when storing many small pieces of information. Attributes compete with tags, but add no real power. The roles of tags for data and attributes for meta-data are often confused. Due to the tree-structure of XML, unobtrusively annotating and composing data is difficult. Note that XML is often wrongly considered similar to RDF, because RDF has an XML serialization format. But conceptually, RDF (relational) and XML (tree-based) are very different.

2.2.2 Relational databases

Relational databases are one of the great success stories of computer science: They have both solid theoretical underpinnings and high-performance implementations.

What is missing are standardized ways of modeling facts and exchanging data. Schema evolution is difficult, for example, adding a new attribute to a table is a non-trivial operation. These deficiencies lead to a difficulty in assembling data from different sources. Lastly, relational databases don't handle semi-structured data well. For example, if an attribute is only needed for one row in the database, it must be added to all of them.

2.2.3 No SQL

The "No SQL" movement is concerned with distributed non-relational databases. It includes examples such as CouchDB[1], and BigTable [CDG+06]. No SQL abandons the query language and the schema constraints of relational databases in favor of greater scalability and flexibility. Such databases usually manage collections of untyped records. Thus, they handle semi-structured data well. They do have deficiencies when it comes to querying or to composing data that cross-cuts records.

2.2.4 RDF in a nutshell

An RDF repository consists of a set of triples, that is, a single relation. The three components of a triple are called (subject, predicate, object). The subject is the identifier of an entity, the predicate is the key of a field, and the object is the value of a field. For subjects and predicates, RDF uses URIs which are roughly the addresses one uses in a web browser. For example, a predicate for the first name of a person is `http://xmlns.com/foaf/0.1/firstName`. To make URIs less unwieldy, they are abbreviated via *namespaces* (more details on namespaces are given later). The following are four URIs and their abbreviations:

Abbreviation	**URI**
`foaf:firstName`	`http://xmlns.com/foaf/0.1/firstName`
`gen:37542`	`http://hypergraphs.de/generated#37542`
`vcard:Country`	`http://www.w3.org/2001/vcard-rdf/3.0#Country`
`country:GB`	`http://downlode.org/rdf/iso-3166/countries#GB`

The first URI is the above mentioned predicate for the first name, the second URI is a generated ID, the third URI is the predicate for the country of residence, the fourth URI is the URI of Great Britain. The following two triples use these URIs:

```
gen:37542 foaf:firstName "John" .
gen:37542 vcard:Country country:GB .
```

The first triple says that the first name of the subject, the person with ID `gen:37542`, is "John". The second triple says that John lives in Great Britain. Objects are thus either *literals* (text strings) or URIs. RDF is obviously a relational data format, similar to storing all relations of a relational database in a single table. The *resource* is a concept that is similar to a database record. A resource is specified via a URI and comprises all triples whose subject is that URI. Thus, the actual data of a resource is a set of

[1] http://couchdb.apache.org/

(predicate,object) pairs, each such pair is called a *property* (with a key and a value). The same predicate can be used several times, which is the same as saying that a property can have multiple values. In a way, RDF is like a normalized relational database that uses URIs as column names and as tuple IDs. In practice, RDF repositories are often based on relational databases and thus build on that technology instead of replacing it.

Fullfilling the requirements. This relatively simple format fulfills all of the requirements mentioned above. The schema is flexible, because, apart from the triples, basic RDF has no schema. Adding a property to a field means adding a new triple to the RDF repository. Conflicts cannot occur, because the same predicate can be used several times and because the use of URIs as property keys prevents name clashes. RDF has external schema languages for checking consistency and inferring data, but these can be applied selectively and otherwise ignored. RDF's globally unique symbols are the URIs. We have already seen the concept "Great Britain" expressed as the URI `country:GB`. Self-description is guaranteed by two conventions: First, each URI, which might be used as a predicate or as a subject, can be annotated with descriptive properties. Second, most standard URIs can be entered into a web browser to retrieve a page describing the URI's meaning to humans. The OWL schema language (Sect. 2.6.2) allows one to declare two URIs as equivalent, which effectively leads to the two resources they denote being merged. Composing two RDF data sources is simple, one constructs the union of the two sets of triples and resources with the same URI are merged automatically (due to how resources are defined). Linking resources can be performed via properties. One thus interprets RDF as a graph, where each triple is an edge between the subject and the object, whose label is the predicate (Fig. 2.2). There are several standardized exchange formats available for RDF. They allow one to export RDF data to a file, which is then either published or processed via external tools. Some of the exchange formats are XML-based, with the goal of machine readability, while the syntax of others makes it easy for humans to author RDF. Alas, RDF is not handling binary data well. A typical work-around is to store binary data externally and reference it from RDF.

2.3 Basic constructs and interpretations of RDF

This section describes how RDF data is structured and explains several ways of understanding it: as tuples, as resources (almost object-based), or as mathematical graphs.

2.3.1 URIs and namespaces

RDF uses URIs (universal resource identifiers) as *symbols*, to refer to concepts such as "Great Britain". They are roughly the internet addresses one types into a web browser. For example, one can use `http://downlode.org/rdf/iso-3166/countries#GB` to identify Great Britain. Because URIs are globally unique, data from different sources can be combined and different symbols will not be mistaken for each other, while two identical symbols will also be recognized as such. Further advantages of URIs are that, due to the popularity of the web, people are already familiar with them. And a URI can be self-documenting by referring to a document that explains its meaning.

To make URIs easier to read and write, they can be abbreviated by defining a *namespace*. A namespace is a short alias for a URI fragment. For example, let

us assume that several URIs for countries all start with the URI fragment `http://downlode.org/rdf/iso-3166/countries#`. The namespace would thus define an alias `country` called the *namespace prefix* for this URI fragment, which is called the *namespace URI*. With this definition, the abbreviated form of `http://downlode.org/rdf/iso-3166/countries#GB` becomes `country:GB`. This way of writing a URI is called a *qualified name* (or *qname*). This particular qname has the *prefix* `country` and the *local name* `GB`. URIs and qnames are written differently in most RDF syntaxes, to avoid confounding them. URIs are usually quoted with angle brackets (as they have many legal characters), while qnames can be free-standing. In this chapter, we use the following namespaces:

Prefix	URI
country	`http://downlode.org/rdf/iso-3166/countries#`
ex	`http://example.com/`
foaf	`http://xmlns.com/foaf/0.1/`
gen	`http://hypergraphs.de/generated#`
rdf	`http://www.w3.org/1999/02/22-rdf-syntax-ns#`
rdfs	`http://www.w3.org/2000/01/rdf-schema#`
vcard	`http://www.w3.org/2001/vcard-rdf/3.0#`

Some of those namespaces end with a slash, making the local name a file name, others end with a hash, making the local name a URI fragment identifier. Sect. 2.5.1 has more details.

2.3.2 Statements: subject, predicate, object

The basic data structure of RDF is the *statement* or *triple*. It is a tuple consisting of three components: The *subject*, the *predicate*, and the *object*. Take, for example, the following statements.

```
gen:37542 foaf:firstName "John" .
gen:37542 foaf:surname "Doe" .
gen:37542 vcard:Country country:GB .
gen:37542 rdf:type foaf:Person .
```

The first and the second statement express that whoever is identified by the URI `gen:37542` has the first name “John” and the surname “Doe”. The third statement expresses that John Doe lives in Great Britain. The last statement says that `gen:37542` is a person. The subject of a statement is always a URI or a blank node (which is similar to a URI and introduced later). The predicate is always a URI. The object can be a URI, a blank node or a *literal* (a text string, enclosed by quotes above).

2.3.3 Repositories, resources and properties

RDF databases are called *repositories*. All they contain are the statements explained above. A *resource* describes a different view on statements. It has an identifier (a URI or a blank node) and all statements whose subject is that identifier *are* that resource. Those statements are called the *properties* of a resource. Each property has a key (the predicate of the statement) and a value (the object of the statement). Continuing our example, resource `gen:37542` has the properties `foaf:firstName`, `foaf:surname`, `vcard:Country`, and `rdf:type`.

This compares very directly to something the reader is probably already familiar with: a (relational) database with contact information. Such a database contains several records where each record holds the information for a single person. The information itself is structured as fields. Each field will have a key such as "first name", "residence" and a value such as "Joe", "Great Britain". Obviously, a resource corresponds to a record and a property corresponds to a field (where property keys are a bit more standardized than field keys). Fig. 2.1 visualizes the comparison.

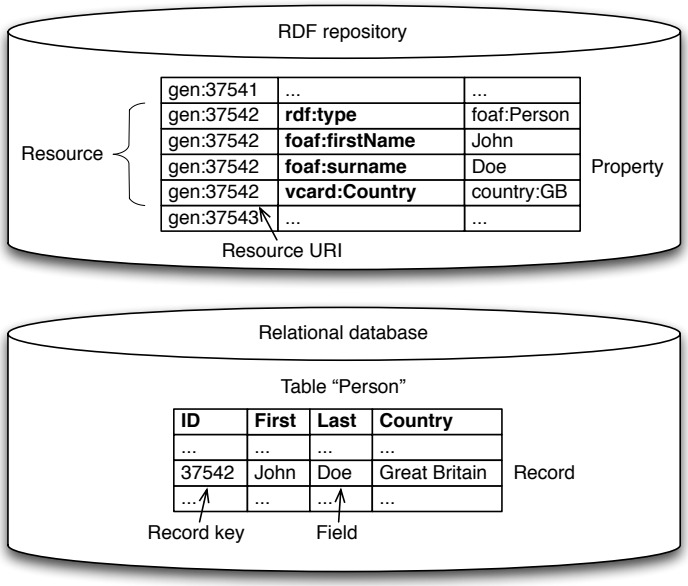


Figure 2.1: An RDF repository contains resources with properties that include a type. A relational database contains tables whose rows are records with fields.

While this feels similar to a record in an imperative programming language, statements are actually immutable, similar to functional data structures. That means that to change a property value, you have to remove the old statement and add a new statement to the repository. Furthermore, the same statement can only exist once in a repository; adding a statement that already exists has no effect. A property always has a set of values, depending on how often its key is used as a predicate (within the same resource). Thus, where programming languages force you to decide whether a variable can have zero or one value (a scalar) or a set of values, RDF does not do so.

2.3.4 Types, classes and instances

Each resource can have a type. A type defines how to interpret the contents of a resource. This is similar to giving files types such as .doc or .jpg. RDF does not explicitly distinguish between data and meta-data in this case and stores the type in the property `rdf:type`, just like any other property. Continuing our example, resource `gen:37542` contains contact information for which the type `foaf:Person` has been standardized. Accordingly, the property `rdf:type` has the value `foaf:Person`.

"Class" is roughly a synonym for "type". One also says that class `foaf:Person` has the *instance* `gen:37542` or that resource `gen:37542` is an instance of class

`foaf:Person`. Right now, classes are just tokens without any semantics. Schema languages (Sect. 2.6) can be used to define what it means that two resources are an instance of the same class.

2.3.5 Graphs and merging repositories

Resources, properties and types are an object-oriented view on RDF, while statements are a relational view on it. A third view on RDF is as a mathematical graph, a set of nodes and a set of edges. The nodes are the RDF resources, the edges are the statements. As an RDF repository only contains statements and not single resources, a node comes into existence by being the subject or object of a statement. If it is neither, we cannot store it in RDF. In practice, this is not a problem, because the only interesting aspect of such a resource is whether it exists or not. And that can be easily expressed differently.

Graphs can be visualized as diagrams where resources are ovals, literals are rectangles, and statements are lines between resources and literals that are labeled with the predicates (see Fig. 2.2).

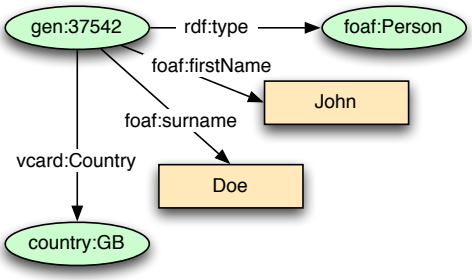


Figure 2.2: The resource `gen:37542` about John as a diagram.

Note how merging of repositories translates elegantly from the union of statements to the diagram: if there are initially two separate diagrams that both have the same URI u, the version of u in the merged diagram has the properties from both original diagrams. This is a pleasant side effect of assembling resources from statements.

2.3.6 Blank nodes and more literals

So far, we have only encountered a single kind of resource: URI nodes that have URIs as an identifiers. *Blank nodes* are a second kind of resource: anonymous resources, without an identifier. To make working with them practical, they do get internal temporary identifiers, but these are not stable. And when merging two repositories, blank nodes will always stay distinct.

All literals that we have seen up to now were simple text strings. RDF actually distinguishes two kinds of literals:

- *Plain literals* are strings combined with optional language tags. This allows one to specify the same text in multiple languages. For example, a property could have the values `"chaise"@fr` and `"Stuhl"@de`. The former is chair in French (language tag “fr”), the latter is chair in German (language tag “de”).

- *Typed literals* are strings combined with a datatype URI. This URI is the type of the literal that specifies how the content of the string is to be interpreted. Standard RDF datatypes include XML schema datatypes such as `xsd:string`, `xsd:boolean`, or `xsd:integer`. Additionally, there is the datatype `rdf:XMLLiteral` for literals that contain XML markup. Arbitrary type literals are written as `"literal value"^^my:datatype`. Literals of type `xsd:integer` and `xsd:boolean` can be written without datatypes and quotes. Thus, `123` is the same as `"123"^^xsd:integer`, and `true` is the same as `"true"^^xsd:boolean`.

2.4 Advanced features

This section explains advanced RDF features: How to encode sequences of RDF nodes and sets of RDF statements; how to refer to statements; and how to exchange RDF data.

2.4.1 Collections and containers

The information contained in resources is always unordered, which is part of the reason why RDF repositories are so easy to merge. Properties contain *sets* of values which is a problem if order matters or if we want to mention the same value twice. One way of fixing this is to introduce a new kind of node for ordered multi-sets (OMS). RDF instead opts for two ways of encoding ordered data using the existing means. To explain them, we use the example of the OMS of literals ⟨`"Harold"`, `"Maude"`⟩. Note that we could have just as well used an OMS of resources.

Collections are of type `rdf:List`. The data structure is a chain of resources where each of the resources uses the property `rdf:first` to store one element of the OMS (Fig. 2.3). The data structure resources are connected by `rdf:rest` properties and the last resource in the chain is always the URI `rdf:nil`.

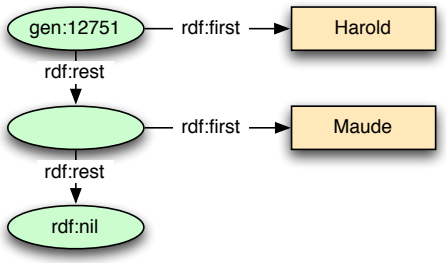


Figure 2.3: A collection with the member literals `"Harold"` and `"Maude"`. The nodes on the right are those member literals. The nodes on the left are the data structure, a chain of resources connected via `rdf:rest` properties. The whole data structure has the URI `gen:12751`.

Containers have one of three types:

- `rdf:Alt` for *alternatives*, i. e. a set of alternative values for, say, a property (applications should choose one them when processing the RDF data);
- `rdf:Bag` for *bags* with unordered members;
- `rdf:Seq` for *sequences* of members where order matters.

All members are directly stored in properties `rdf:_`i where i is the index of the member (Fig. 2.4).

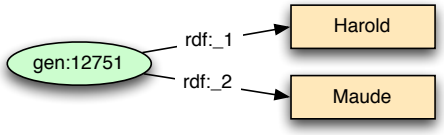


Figure 2.4: A container with the member literals `"Harold"` and `"Maude"`. The whole container has the URI `gen:12751`.

2.4.2 Named graphs

Especially when different data sources are being merged, it is desirable to track sets of statements. In order to do so, most modern RDF repositories add another component to a statement, turning it from a triple to a quadruple: the *graph*. For example such a repository might contain the following quadruples:

```
graph:1 gen:37542 foaf:firstName "John" .
graph:1 gen:37542 foaf:surname "Doe" .
graph:2 gen:37542 vcard:Country country:GB .
graph:2 gen:37542 rdf:type foaf:Person .
```

Grouping by graph, we can say that the repository contains `graph:1` with the triples

```
gen:37542 foaf:firstName "John" .
gen:37542 foaf:surname "Doe" .
```

and `graph:2` with the triples

```
gen:37542 vcard:Country country:GB .
gen:37542 rdf:type foaf:Person .
```

During most operations, the graph URI is ignored, leading to a view that a repository stores triples that are tagged with where they came from. Thus, the resource `gen:37542` is still considered to have four properties, but we can now determine that they came from two different sources.

2.4.3 Reification

Most data can be easily expressed in binary relations. But sometimes, one needs to annotate a relationship itself. For example, when one assigns a body height to a person, one might want to record when this assignment has been made. Thus the statement

```
gen:37542 ex:bodyHeight "180"^^ex:cm .
```

has to be annotated somehow. RDF does not have a direct way of doing this, so the statement is *reified* as a resource:

```
gen:12345   rdf:type         rdf:Statement .
gen:12345   rdf:subject      gen:37542 .
gen:12345   rdf:predicate    ex:bodyHeight .
gen:12345   rdf:object       "180"^^ex:cm .

gen:12345   dcterms:date     "2009-10-14"^^xsd:date .
```

The last statement is the annotation. Especially for data modeling, reification is often too brittle. An alternative is to use one of the RDF patterns described in Chap. 12.

2.4.4 Exchange formats

RDF is often being confused with being an XML dialect, because its first standardized interchange format was XML/RDF. This format had many problems. It was neither easy to read nor easy to write for humans, nor does it support named graphs. Thus new external syntaxes were invented:

- Turtle (Terse RDF Triple Language): the syntax used in this document. So far, the only syntactic construct that has been shown was the triple. Additionally, Turtle has syntactic sugar to concisely express collections, several statements with the same subject, several statements with the same subject and predicate, etc. [BBL08].
- Notation 3 (N3): A more powerful syntax than Turtle (which has been created as a simplified version of N3) [BLb].
- Syntaxes supporting named graphs [C+]: Trix is an XML-based syntax for machine-readable RDF with named graphs. Trig is a version of Turtle that can handle named graphs.

2.5 Best practices for basic RDF constructs

2.5.1 Creating good URIs

The local name of a URI is the actual term denoted by the URI. It is separated from the namespace URI either by a slash or a hash:

1. Separated by a slash: `http://example.com/namespace/localname`
   The namespace URI is `http://example.com/namespace/`
2. Separated by hash: `http://example.com/namespace#localname`
   The namespace URI is `http://example.com/namespace#`

In the former case, the local name is a file name, in the latter case, it is a URI fragment. This determines how self-description of URIs is handled for humans: Either there is one document for each term or one document for each namespace that explains all the terms. Each of the two approaches has its benefits, Sect. 3.2.2 on information resources has details. For general information on how to choose URIs consult the paper "Cool URIs for the semantic web" [SC]. The following section covers the special case of how to name properties.

Property names

Finding good property names is difficult, because they are used in different contexts. Sometimes one displays RDF as statements and a verb is a good choice for the predicate:

```
:Jane foaf:knows :Tarzan
```

On the other hand, properties whose values strongly belong to a resource are often displayed in a form-like manner. Then nouns (so-called *role nouns*[2]) make more sense.

```
:Jane foaf:name "Jane Porter"
```

An anti-pattern of property naming is to combine a verb and a noun or a noun and a preposition. The rule of thumb is: What property name would look good in a form? Clearly, `name` works better than `hasName` and `superclass` works better than `subClassOf`. N3 has special support for role nouns. The following two statements are equivalent in N3:

```
:Jane :father :Archimedes .
:Archimedes is :father of :Jane .
```

Note that `foaf:knows` does not have a good corresponding noun and thus is an example where a verb (on its own) makes sense as a property name.

A third way of naming is to use adjectives. Examples include `skos:broader` and `skos:related`. In both cases the reading is *has X* and not *is X than/to*. This is already indicated by the standard SKOS labels "has broader" and "has related". As adjectives fit neither of the use cases "display as a statement" and "display in a form", it often makes sense to use a noun or a verb instead.

2.5.2 Using literals

The following rules help with using literals.

Use datatypes to mark text that has special meaning. If a text string should be interpreted in a special way, it is common practice to give the literal holding it a custom datatype. An example is the datatype `wikked:Markup` for wiki markup that is stored in properties. When saving a resource, this allows HYENA to do special post-processing of the markup. An additional use case of datatypes is for quickly looking up IDs. If one were to define a datatype for ISBN numbers, such numbers could be found via SPARQL in any property, without any false positives.

Use language tags for internationalization. This is what language tags have been created for: a resource is given several labels, one for each language. The language tags of these labels help distinguish them. If an application can be switched to a specific language, only the labels in those language will be shown.

2.6 Schema, rules, and querying

A number of standards provide more sophisticated ways of processing and describing RDF.

[2] http://esw.w3.org/topic/RoleNoun

2.6.1 RDF schema language (RDFS)

An *RDF vocabulary* is a set of URIs used for a particular purpose. It can be seen as a less powerful version of an ontology (Sect. 2.6.2). RDFS defines a vocabulary for describing RDF vocabularies. The main components of this meta-vocabulary are:

- Classes and class hierarchies: One can explicitly describe and type classes for resources and literals in RDF. Furthermore, a hierarchy of classes can be built via `rdfs:subClassOf`.
- Properties and property hierarchies: Properties are described by giving the property key the type `rdf:Property` and by assigning a domain and a range via `rdfs:domain` and `rdfs:range`.
- Helper properties: `rdfs:label` is used to attach short human-readable labels to RDF resources, `rdfs:comment` provides a longer description of a resource, `rdfs:seeAlso` points to related resources.
- Other vocabulary: Containers, collections, and reification are also part of RDFS.

RDFS performs *inference*: RDF definitions are applied as rules to RDF statements to generate new statements. One example is the transitivity of the property `rdfs:subClassOf`: Given two triples

```
B rdfs:subClassOf A .
C rdfs:subClassOf B .
```

RDFS *infers* a third triple

```
C rdfs:subClassOf A .
```

If a repository performs inference, the inferred statements are added to the repository and are indistinguishable from non-inferred statements (except for not being removable). Sect. 5.2 has more information on RDFS.

2.6.2 Web ontology language (OWL)

Similar to a vocabulary, an ontology defines a set of URIs. But the means for specifying the meaning of these URIs tend to be more powerful. OWL can roughly be considered an extension of RDFS. Like RDFS, it does inferencing. OWL goes a step further by allowing custom inferencing rules: In addition to the built-in transitivity of, for example, `rdfs:subClassOf`, *any* predicate can be declared transitive. Other features that are relevant for data modeling are:

- Inverse properties: If property `ex:child` is declared the inverse of `ex:parent`, then a triple

  ```
  ex:Jill ex:parent ex:Bruce .
  ```

  will lead to the inference of the triple

  ```
  ex:Bruce ex:child ex:Jill .
  ```

- Declaring equivalence: `owl:equivalentClass` declares two classes to be equivalent, `owl:equivalentProperty` and `owl:sameAs` do the same for two properties and two resources. This can be useful when integrating different data sets.

OWL has many more features, but those are beyond the scope of this chapter. Sect. 5.2 provides more information.

2.6.3 Semantic web rule language (SWRL)

SWRL [HPSB+] adds the Unary/Binary Datalog RuleML sublanguages of the Rule Markup Language [Rul] to OWL. It thus enables Horn-like rules for OWL knowledge bases. These rules control inference. An example from [HPSB+] is

```
hasParent(?a,?b) ∧ hasBrother(?b,?c) ⇒ hasUncle(?a,?c)
```

2.6.4 The SPARQL query language

SPARQL [PS05] is the standard query language for RDF. Its syntax has been inspired by SQL. The following example looks for wiki pages and displays the ones first that have been modified last.

```
SELECT ?modified ?title WHERE {
    ?page rdf:type wikked:Page .
    ?page rdfs:label ?title .
    ?page dcterms:modified ?modified .
} ORDER BY DESC(?modified)
```

2.7 Useful vocabularies

Vocabularies define sets of URIs that are published for public use. Common vocabularies help to make data exchangeable. The following list is a sampling of interesting RDF vocabularies. It gives an impression how and for what one would define vocabularies.

Dublin Core

> The Dublin Core Metadata Element Set is a vocabulary of fifteen properties for use in resource description. The name "Dublin" is due to its origin at a 1995 invitational workshop in Dublin, Ohio; "core" because its elements are broad and generic, usable for describing a wide range of resources. [dublincore.org]

Among other constructs, the Dublin Core RDF vocabulary [dcm] defines properties such as `dcterms:date`, `dcterms:creator`, `dcterms:hasVersion`, `dcterms:license`, `dcterms:publisher`.

Friend of a friend (FOAF)

> The Friend of a Friend project is creating a Web of machine-readable pages describing people, the links between them and the things they create and do. [foaf-project.org]

FOAF defines [BM] classes such as `Person` and properties such as `name`, `weblog`, `knows`, `publications`, `currentProject`, `interest`, `myersBriggs`, `logo`, `fundedBy`.

Semantically-Interlinked Online Communities (SIOC)

> The SIOC Core Ontology provides the main concepts and properties required to describe information from online communities (e.g., message boards, wikis, weblogs, etc.) on the Semantic Web.

The SIOC ontology [B+b] defines classes such as `Community`, `Forum`, `Post`, `Site`, `Thread`, `User`, and `Usergroup`. It defines properties such as `has_modifier`, `has_parent`, `has_reply`, `has_subscriber`, etc.

Other vocabularies

- Simple Knowledge Organisation System (SKOS): "SKOS is an area of work developing specifications and standards to support the use of knowledge organisation systems (KOS) such as thesauri, classification schemes, subject heading lists, taxonomies, other types of controlled vocabulary, [...]" [MB]. SKOS defines classes such as `Concept` and properties such as `broader`, `narrower`.
- The Fresnel display vocabulary[3]: allows one to specify, in RDF, how to display RDF data in a browser-independent manner.
- Description of a project (DOAP[4]) describes open source projects (homepage, maintainer, etc.).
- RDF Calendar[5]: encodes the iCalendar standard for calendar data in RDF.
- vCard in RDF[6]: for encoding contacts (name, address, telephone number etc.) in RDF.
- Creative Commons[7]: lets you describe copyright licenses in RDF.

2.8 RDF applications

This section describes RDF-based applications, illustrating how RDF can be used in practice. It contains the following sub-sections:

1. The *Extensible Metadata Platform* (XMP): attaches RDF-encoded meta-data to files. RDF helps with standardizing file meta-data.
2. Semantic web search engines: catalog RDF that has been published online and make it searchable. They complement the traditional "web of (unstructured) documents" with the "web of data", making a different part of the web accessible, via more precise and sophisticated querying.

[3] http://www.w3.org/2005/04/fresnel-info/
[4] http://usefulinc.com/doap
[5] http://www.w3.org/TR/rdfcal/
[6] http://www.w3.org/TR/vcard-rdf
[7] http://creativecommons.org/ns

3. PiggyBank: extracts structured data such as citations or contact information from web pages and makes it available as RDF. This helps with collecting and processing such data.

4. RDF online databases: Online databases are increasingly based on RDF, due to the ease of exchanging data and of handling heterogeneous data.

5. Internal use of RDF: Some applications use RDF internally, but hide that fact from the user as much as possible.

2.8.1 Extensible Metadata Platform (XMP)

XMP[8] is a standard initiated by Adobe that allows one to store RDF-based meta-data about a file inside the file. Supported file formats are, among others, HTML, SVG, JPEG, PNG, PDF, Photoshop documents, and Illustrator documents. Inside the files, RDF data is serialized as RDF/XML (minus a few of RDF/XML's non-essential features). Standard XMP vocabularies are Dublin Core, XMP Basic, XMP Rights Management, XMP Media Management, XMP Basic Job Ticket, XMP Paged Text, XMP Dynamic Media. Specialized XMP vocabularies are Adobe PDF, Photoshop, Camera Raw, and EXIF. Properties typically used by Adobe applications are:

- Dublin Core: `dc:title`, `dc:creator`, `dc:description`, `dc:subject`, `dc:format`, `dc:rights`
- XMP basic: `xmp:CreateDate`, `xmp:CreatorTool`, `xmp:ModifyDate`, `xmp:MetadataDate`
- XMP rights management: `xmpRights:WebStatement`, `xmpRights:Marked`
- XMP media management: `xmpMM:DocumentID`

But XMP is not limited to these vocabularies, any RDF can be embedded. Furthermore, it can be embedded in almost any kind of file, even if its format is not unknown to XMP. This technology is called *XML Packet* and the algorithm to find XMP meta-data in unknown files is to look for the text string

```
<?xpacket begin=
```

This is the beginning of the opening tag of the XML packet. Subsequently, there is XML packet specific data, then the RDF data, serialized as RDF/XML.

2.8.2 Semantic web search engines and supporting tools

Semantic web search engines collect RDF data that has been published on the web and make it searchable. Where traditional web search engines search the web as a collection of documents, semantic web search engines search the web as a collection of databases. That means that more precise results can be produced and more sophisticated queries can be formulated. Recent examples of semantic search engines are:

- Falcons at http://iws.seu.edu.cn/services/falcons/
- SWSE at http://swse.deri.org/

[8] http://www.adobe.com/products/xmp/

- Sindice at http://sindice.com/
- Watson at http://watson.kmi.open.ac.uk/WatsonWUI/
- Swoogle at http://swoogle.umbc.edu/

Two tools have been created to help semantic search engines find data on the web. **Ping the semantic web**[9] (PTSW) has been created to complement semantic search engines. It is a web service that functions as a registry of all RDF data on the web. It is to be used by RDF crawlers looking for content. Updates to the data are tracked, so that one can find out what has most recently been updated. With an account, one can leave a server IP address where notifications of updates are sent. When a URL is registered with PTSW, it tries to get an RDF representation via content negotiation (Sect. 3.2) and additionally follows RDF files referenced via a `<link>` tag. Any file that is found is only accepted if it can be validated. **Semantic radar for Firefox**[10] is a Firefox extension that watches the pages a user visits and indicates the availability of RDF data by showing an icon the status bar. Currently supported are RDFa and references via `<link>`. Semantic radar works in conjunction with Ping the semantic web and can send it pings of the RDF data that has been found.

2.8.3 Piggybank: Extracting RDF content from online sources

Piggybank[11] is an RDF screen scraper which has been implemented as a Firefox extension. While visiting sites such as Flickr, the ACM portal, or LinkedIn, Piggybank can extract data from those sites such as events, citations, and contact information. The extracted data can then be tagged and shared with others via an online account.

2.8.4 RDF online databases

Online databases are increasingly based on RDF, due to the ease of exchanging data and of handling heterogeneous data. With RDF, a set of databases can feel like a single large database; all that is necessary are a few URIs being shared. **DBpedia** is Wikipedia translated to RDF:

> DBpedia is a community effort to extract structured information from Wikipedia and to make this information available on the Web. DBpedia allows you to ask sophisticated queries against Wikipedia, and to link other data sets on the Web to Wikipedia data. [http://dbpedia.org/]

DBpedia's extraction algorithm is helped by the fact that Wikipedia already uses subject-specific templates for entering data. It is a rich source of RDF vocabulary that is linked against by other members of the linked web of data (Chap. 3). Its data can be downloaded and is published by a web service. The web application DBpedia mobile[12] uses a map to display DBpedia information that has geographical coordinates.

Freebase[13] is a public, collaboratively edited, database. Users enter structured data: things and facts about things. Compare this to mainly text input managed by Wikipedia. While Freebase is not directly based on semantic web principles, its way of organizing data is very similar and it can export any of its data as RDF.

[9] http://www.pingthesemanticweb.com/
[10] http://sioc-project.org/firefox
[11] http://simile.mit.edu/wiki/Piggy_Bank
[12] http://wiki.dbpedia.org/DBpediaMobile
[13] http://freebase.com/

2.8.5 Internal use of RDF

More and more applications are based on RDF, but hide that fact from their users as much as possible. This includes information managers such as Haystack [QHK03] or the Social Semantic Desktop [BDE+08]. **Relo**[14] is another example of an RDF application. It helps developers explore Java code bases. Its interface is a UML-like diagram where users can directly edit code. Relo stores its data as RDF, opening the door to using SPARQL to make complex queries about source code.

[14] http://relo.csail.mit.edu/

Chapter 3

Linked data on the web

Contents

3.1 Overview

Linked data tackles the specifics of deploying RDF data on the web. It is currently not supported by CoIM (and thus not prerequisite knowledge for understanding it), but the logical next step in its RDF support. What this next step might look like is explained as "future research" at the end of this chapter.

Linked data is a collection of best practices for the deployment of RDF data: How can it be distributed and processed on a large scale? How can machine readability, navigability and usability for humans be ensured at that scale? Linked data is based on the ideas that made the web of documents successful and extends them to build a web of data [Hea09]. For practical applications, the term *linked data* is increasingly replacing the term *semantic web*. There is no clear consensus[1] how the two terms are related, the most common understanding is that linked data is a sub-topic of the semantic web, providing a data infrastructure on which semantics can be based. This chapter first explains the core concepts surrounding linked data and shows how linked data can be made easy to discover. A number of technologies help with write-enabling linked data and turn sets of linked data into lean web services that are founded on simple principles. The chapter concludes with future research.

3.2 Core concepts

[BHBL09] describes linked data as follows:

[1] http://tomheath.com/blog/2009/03/linked-data-web-of-data-semantic-web-wtf/

> Linked data refers to data published on the web in such a way that it is machine-readable, its meaning is explicitly defined, it is linked to other external datasets, and can in turn be linked to from external datasets.

Linked data adheres to and builds on the core principles of the classic world wide web [BHBL09] to create a *web of data*: The web of data can contain any kind of data, published by anyone, without being subject to vocabulary constraints. Entities are linked into a global data graph that spans data sources and enables the discovery of new sources.

Application developers enjoy the following benefits [BHBL09]: Data is clearly separated from presentational aspects. Data is also self-describing, because there is a standard way of looking up the definition of a URI, for machines and for humans. HTTP as a standard data access mechanism and RDF as a standard data format make linked data simpler than most web APIs which rely on heterogeneous data models and access interfaces. Finally, the web of data is open, meaning that applications need not work with a fixed set of data, but can be written to discover new data by following links. In the following, linked data's core concepts are explained. Sect. 24.3 describes "Tabulator", a linked data client program for end users.

3.2.1 Linking documents and data

The following concepts refer to the linking aspect of linked data.

The web of documents is "the web as we currently know it": it is the equivalent of a global file system, designed for humans, its primary objects are documents, there are untyped links between documents, the content is relatively low on structure and semantics are implicit. Even though there is a lot of data on the web of documents, it is difficult to access, because it is typically unstructured and it does not adhere to a common standard.

The graph of linked things (the web of data) is the equivalent of a global database, designed for machines, primary objects are things, there are typed links between things, the content has a high amount of structure and explicit semantics. This is the core component of linked data, but the graph complements the web of documents, it does not replace it.

Dereferencing a URI is the process of looking up on the web a so-called *resource*, information that is pointed to by a URI. The two main involved parties are the client making the request and the server answering it. During the request, *content negotiation* takes place: The client tells the server what formats it prefers and the server can then create and deliver the appropriate *representation* of the resource. For example, web browsers normally request HTML via the HTTP header `Accept: text/html`, while linked data clients requesting RDF might send `Accept: application/rdf+xml`. The idea is that when humans looks up a resource, they usually want to read text and browse data, while machines are only interested in the raw data. Flexible servers react accordingly and generate the appropriate representation on the fly.

3.2.2 Information resources and non-information resources

URIs can represent two kinds of resources. *Information resources* are things (documents, images, etc.) that actually exist on the web. During content negotiation, there are two ways of providing different representations: one can directly serve a representation or one can forward to an appropriate resource with the response code `302 Found`. That means in both cases that data has been found at the location of the URI.

Non-information resources such as abstract concepts and people do not actually exist on the web. To honor the semantics of this fact while still serving data, servers usually forward to a resource that contains related information with the response code `303 See Other`. Alternatively, a non-information resource can have a URI with a fragment identifier such as `http://example.com/file#fragment`. Then the non-fragment part of the URI can be served and the URI with fragment retains its abstract status. The former method has the advantage of being more flexible, but it requires redirects. The latter method has to serve the complete parent resource every time one of its fragments is accessed. On the other hand, no redirects are required. Fragments work well for small stable vocabularies, while 303 redirects should be used for large evolving vocabularies [SC].

As an example, here are three URIs related to "Russia" [BCH07]:

1. `http://www4.wiwiss.fu-berlin.de/factbook/resource/Russia`
   (URI identifying the non-information resource Russia)
2. `http://www4.wiwiss.fu-berlin.de/factbook/data/Russia`
   (information resource with an RDF/XML representation describing Russia)
3. `http://www4.wiwiss.fu-berlin.de/factbook/page/Russia`
   (information resource with an HTML representation describing Russia)

In this context, the following usage scenarios are conceivable: accessing the non-information resource (1) will 303 forward web browsers to (2) and RDF clients to (3). Accessing the HTML page (3) with a web browser will directly serve that page, while RDF clients will be 302 forwarded to (2).

To avoid the complexities of content negotiation, one can also serve HTML and RDF at the same time, by marking up an HTML page with RDFa (Sect. 3.3.3). This allows humans to read the rendered HTML, while machines can extract the embedded RDF.

3.2.3 Berners-Lee's four rules for linked data

Tim Berners-Lee [BLa] states four rules for linked data:

- **Use URIs as names for things.** URIs can denote anything, not just documents.
- **Use HTTP URIs so that people can look up those names.** HTTP URIs have the advantage of being globally unique with distributed ownership.
- **When someone looks up a URI, provide useful information.** What representation (RDF, HTML, ...) of a URI is useful depends on who accesses it. Content negotiation is a clean solution for this problem.
- **Include links to other URIs, so that they can discover more things.** Note that this rule holds for both RDF and HTML. In both cases, linking is essential to the distributed nature of the web.

3.2.4 Datasets

In the linked data universe, a *dataset* is a graph of RDF data. It is often hosted in a single place and there are several possible ways of accessing it:

- As linked data. Granularity is the namespace for hash URIs and the resource for slash URIs. In the former case, one usually serves all statements whose subjects are part of the namespace. In the latter case, one usually serves all statements whose subject is the URI.
- Via SPARQL endpoints: web services where one sends SPARQL queries and receives query results, see Sect. 3.4.4.
- As a data dump, a complete files with all of the data.
- Via *URI lookup*, as defined by the voiD specification (Sect. 3.3.2): one URL-encodes a resource URI R, appends it to a *URI lookup endpoint*, and downloads the resulting URI which is expected to contain an RDF description of R.

3.3 Discovery

This section describes techniques to make a dataset easy to discover, given either an HTML web page, a web server, or another data set. Fig. 3.1 provides an overview.

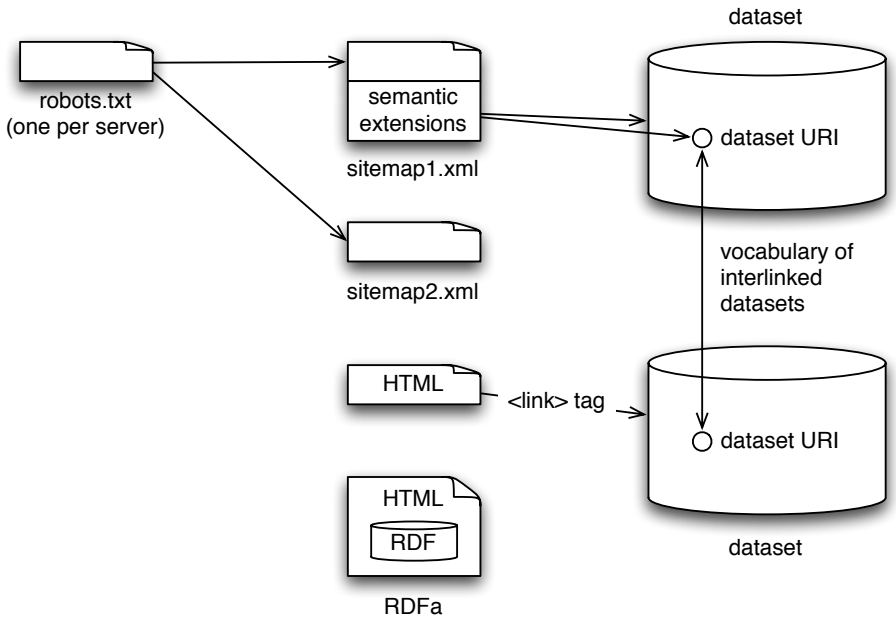


Figure 3.1: Datasets can be discovered by following the per-server `robots.txt` to semantic sitemaps. HTML files can refer to related RDF datasets, if those are stored in files. The RDFa standard allows one to embed related RDF data inside HTML. The RDF vocabulary of interlinked datasets is used to provide further meta data.

3.3.1 Semantic sitemaps

Classic sitemaps are XML files that are declared in the per-domain `robots.txt` file. *Semantic sitemaps* [CSD+08] extend this format to allow one to describe datasets:

- datasetLabel: a human description of the dataset.
- datasetURI: points to a URI in the dataset that contains more information about the dataset. This information can be expressed in the voiD vocabulary (Sect.3.3.2).
- linkedDataPrefix: describes how the resources in the dataset are served as linked data. URIs that begin with this prefix must resolve to RDF descriptions. What exactly such a description entails can also be specified (statements whose subject the URI is, statements whose subject or object the URI is, etc.).
- sparqlEndpoint: where can SPARQL be used to query the dataset?
- dataDump: where can a dump of the dataset be downloaded?
- changefreq: how frequently does the data change?
- sparqlGraphName: what is the graph URI of the dataset inside the SPARQL endpoint?
- sampleURI: provides starting points for human exploration of the dataset.
- authority: normally, the only way to get information about who is responsible for a web site is via the DNS records, which are external to the site. By specifying a URI for an authority, this kind of information can be served via linked data principles.

Having this kind of information solves the following problems:

- Crawling performance and exhaustive data enumeration: crawling linked data can be expensive, because each resource might be served separately and there is no definitive way of retrieving all of the dataset. By associating a linked data prefix with a data dump, this can be fixed.
- Scattered RDF files: instead of having to crawl a complete site to find RDF data, crawlers simply look up the site map.
- Cataloging SPARQL endpoints: semantic sitemaps provide a standard way of discovering SPARQL endpoints.
- Discovering a SPARQL endpoint for a given resource: is especially interesting for linked data, because it allows one to go from looking up a resource to making sophisticated queries about it. Sitemaps allow this by associating linked data prefixes with SPARQL endpoints.
- Closed-world queries about self-contained data: Usually, RDF's open world assumption makes queries such as "Does Example Inc. have an employee named Eric?" impossible. By delineating datasets, one can temporarily assume a closed world. If a query fails to produce results, the dataset label can be used to report the failure.

3.3.2 Vocabulary of Interlinked Datasets (voiD)

The *Vocabulary of Interlinked Datasets* (voiD, [ACHZ]) extends semantic sitemaps by providing a detailed RDF vocabulary for describing datasets and how different datasets are linked.

Describing datasets

The vocabulary for datasets comprises the type `void:Dataset` for datasets and lets one specify the following data:

- About the dataset: title, description, home page.
- Who created it: creator (main author), publisher, contributor.
- URIs in the dataset: example resources (as entry points) and a regular expression that matches one or more URIs in the dataset.
- Dates: date of creation, issuance, modification.
- Describing the content: URIs tag a dataset with categories (such as "computer science" or "biology"), a URL points to the license under which the dataset has been published.
- Composition of the content: one can specify a source from which the current dataset has been derived, sub-datasets which are combined to form the whole dataset, and the vocabularies used in the dataset.
- Access methods: one can declare SPARQL endpoints, URI lookup and data dumps.

Describing links between datasets

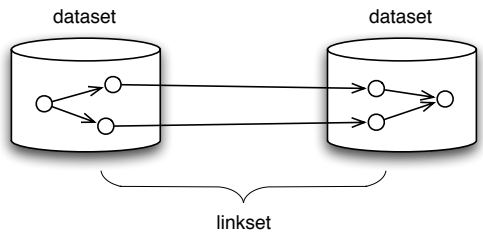


Figure 3.2: A linkset is a dataset that connects two datasets.

A *linkset* is a dataset that contains links between two datasets (Fig. 3.2). These links can declare a URI from one dataset to be equivalent to a URI from the other dataset. Or they can otherwise relate URIs (who knows whom, who lives where, etc.). voiD distinguishes two kinds of linksets: "classic linked data" means that one of the involved datasets contains the links, "3rd-party" means that the linkset is external to the datasets. The description of the links can specify a direction or be undirected. The following URIs are available:

- `void:Linkset` is the type of linksets.

- `void:subset` (read as "has subset") declares that one dataset is a superset of another one. In particular, it can be used to declare that one of the datasets contains the linkset (which is also a dataset), as is the case in the "classic linked data" scenario.
- `void:linkPredicate` states one predicate that is used for linking the datasets.
- `void:target` to declare the involved datasets in the undirected case.
- `void:subjectsTarget`, `void:objectsTarget` to describe the direction of the links between the datasets.

The following example (taken from [ACHZ]) describes a linkset connecting the Jamendo and Geonames datasets. The former describes a collection of Creative Commons licensed songs, including the artists that performed them. The latter is an ontology of places (continents, countries, cities, etc.). The linkset is contained in the Jamendo vocabulary (classic linked data). The links are directed from Jamendo to Geonames and have the predicate `foaf:based_near`. Such links can be used to express where, for example, bands are located.

```
:Jamendo void:subset :Jamendo2Geonames .

:Jamendo2Geonames a void:Linkset ;
                  void:linkPredicate foaf:based_near ;
                  void:subjectsTarget :Jamendo ;
                  void:objectsTarget :Geonames .
```

Discovering voiD descriptions

Usually, one uses the dataset URI specified in a semantic sitemap to look for linksets whose target is that URI. Should the dataset be spread across several documents, each of those documents can use `dcterms:isPartOf` to refer to the main dataset URI with the voiD description:

```
<document.rdf> dcterms:isPartOf <void.ttl#MyDataset>
```

3.3.3 Associating RDF data with HTML files

To associate an HTML file with related RDF data, one can either refer to an RDF file via an HTML tag or embed the RDF inside the HTML. There are two reasons for performing such an association. First, the RDF can be meta-data, describe what the page is about, its intended audience, topics covered, etc. Second, it can be a more precise version of the data displayed on the web page, given that many web pages display structured data (such as contacts and events) as unstructured text. Obviously, translating that unstructured text to structured data for further processing is a non-trivial task.

Linking from HTML to RDF.

`<link>` tags allow one to tell browsers about content that is related to the current HTML document. This is already frequently used for RSS feeds: If a feed version of the current pages is available, it is linked to in HTML and a status icon indicating the

additional content shows up in the web browser. In a similar fashion, one can link to RDF data by including the following tag in the header:

```
<link rel="alternate" type="application/rdf+xml"
      title="RDF Version" href="mydoc.rdf">
```

RDFa: Embedding RDF inside HTML

Many web pages are produced by translating a database to HTML. Alas, this process is not reversible: Even though there are many lists of contacts or dates available on the web, one cannot easily import this data into address books and calendars and usually ends up entering it manually. Help comes in the form of an (X)HTML extension called *RDFa* [AB06]. It defines a small set of attributes that remain hidden inside the HTML, but their values allow one to extract the RDF triples that were used to generate the HTML in the first place.

In the following example (which is taken from [Wikb]), relatively free text is marked up with RDFa:

```
<p xmlns:dc="http://purl.org/dc/elements/1.1/"
   about="http://www.example.com/books/wikinomics">
  In his latest book
  <em property="dc:title">Wikinomics</em>,
  <span property="dc:author">Don Tapscott</span>
  explains deep changes in technology,
  demographics and business.
  The book is due to be published in
  <span property="dcterms:date"
        content="2006-10-01">October 2006</span>.
</p>
```

All RDFa-specific parts are underlined: we initially define the namespace prefix `dc`[2] for the URI `http://purl.org/dc/elements/1.1/`. Then we say that the paragraph is `about` the resource `http://www.example.com/books/wikinomics`, meaning that the properties we are about to define should be attached to it. Then we define the `dc:title` and `dc:author` of that book. For the date, we use an RDFa mechanism that lets us display `October 2006` to the user, but store `2006-10-01` in RDF. Note that even though we discard the former, it is still semantically marked up as the value of property `dcterms:date`. Thus, the following three triples have been embedded in the HTML fragment and can be easily extracted:

```
http://www.example.com/books/wikinomics
    dc:title "Wikinomics" ;
    dc:author "Don Tapscott" ;
    dcterms:date "2006-10-01" .
```

There are tools such as *bookmarklets* [rdf] available with which it is easy to extract the RDF from RDFa-enriched web sites. All HTML generated via Fresnel lenses in HYENA has RDFa in it.

RDFa is similar to *microformats* [sit]. Microformats introduce a new set of HTML attributes and CSS classes for each kind of data. Examples include People and Organizations; Calendars and Events; Opinions, Ratings and Reviews; Licenses; Tags,

[2] `dc` stands for the standard RDF vocabulary *Dublin Core* which defines predicates for author information etc.

Keywords, Categories; Lists and Outlines. In contrast, RDFa relies on the universality of RDF and just needs a new RDF type. On the other hand, microformats do not require one to handle RDF and are already broadly supported via browser plugins.

Google has recently begun using RDFa to improve search efficiency. Google supports what they call "rich snippets"[3] to improve how a web page is displayed in the search results. Snippets are HTML fragments that are marked up in either a microformat or RDFa. Initial support is for reviews and people. The former allows Google to collect all reviews for a given product. The latter allows Google to distinguish several people with the same name.

3.4 Write-enabling linked data

The technologies that have been described turn datasets into lean web services that are founded on simple principles. But so far, only read access to data has been possible. This section describes the means[4] for write-enabling linked data: Web IDs are public IDs for people and other entities; FOAF+SSL is an authentication protocol that is based on Web IDs; Web Access Control is an authorization scheme and protocol; the update protocols WebDAV and SPARQL Update are used for effecting changes in a distributed setting.

3.4.1 Web ID: entity identifiers

The core idea of Web ID is to extend linked data principles to personal identity. To create a web ID for user Romeo, one performs the following steps:

- Create a FOAF file that describes in RDF who Romeo is, refers to his homepages, his pictures, his friends, etc. Put the file at `http://example.com/~romeo/foaf.rdf`
- Romeo's web ID is `http://example.com/~romeo/foaf.rdf#me`. Other fragment identifiers are possible such as initials, `this`, etc. This means that the URI with `#me` is a non-information resource, while the URI without `#me` is an information resource.
- The FOAF file refers to Romeo's friends and acquaintances via their web IDs, creating a network of social data.

The next section explains how web IDs can be used for automatic authentication.

3.4.2 FOAF+SSL: authentication scheme and protocol

FOAF+SSL is an authentication scheme based on Web IDs. It involves cryptography for secure communication (no one listens in, data is not changed). There are two main kinds of cryptography:

- Symmetric-key cryptography: both communication parties have the same secret key. Disadvantage: one has to manage the keys of all of one's communication partners. This was the only way of doing cryptography until 1976 [DH76].

[3] http://www.google.com/support/webmasters/bin/answer.py?answer=99170
[4] http://esw.w3.org/topic/WriteWebOfData

- Public-key cryptography: Encryption is asymmetrical and one always uses pairs of keys: A *private key* is kept secret, a *public key* can be freely published. One of the two keys is used for encryption, the other one for decryption. Naturally, the two keys are related, but computing the private key given the public key is assumed to be too computationally expensive to be practical.
  - Encryption: For encryption, a public key is used to encrypt messages and a private key is used to decrypt them. This ensures that only the owner of the private key can read the encrypted message.
  - Signing: When signing a document, the role of the public key and the private key is reversed. The signer encrypts the document or a summary (e.g. a hash) of it to become the signature, with the private key. The signature is attached to the document and the recipient of the document uses the public key to decrypt it, after which it can be used to ensure the authenticity of the document.
  - Public key certificate[5]: a document that binds together a public key and an identity (full name, alias, or some other kind of identifier). This document is used to verify that a public key belongs to an individual. To ensure authenticity, the certificate is signed, often by a trusted public entity, but self-signing is also common.

The FOAF+SSL protocol works as follows [SHJJ09]:

1. The client, Romeo, dereferences a secure URL such as `https://juliet.net/location`.
2. As part of the exchange with the client, the server receives the client's certificate with the client's public key and the client's web ID, `http://example.com/~romeo/foaf.rdf#me`. As the connection is encrypted, the server can be sure that the client knows the private key corresponding to the public key in the certificate (and has not just copied the public key from somewhere).
3. Juliet's server downloads `http://example.com/~romeo/foaf.rdf` and checks that that document mentions the same public key that is used in the current exchange. Juliet can now be sure that Romeo has write access to the resource `http://example.com/~romeo/foaf.rdf`. That means that the communication partner has been authenticated "as" the given URI.
4. The web ID can now be used to look up Romeo's trustworthiness, for example by (recursively) crawling Juliet's FOAF file to find out how direct a friend Romeo is. Or authorization criteria is looked up locally. In both cases, access is either granted or denied.

3.4.3 Web Access Control: authorization scheme and protocol

The Web Access Control[6] (WAC) system provides an ontology for specifying access control lists for web resources. Dereferenceable URIs are used for resources, users and groups. That means that resources, users and groups can be hosted on different systems and are brought together by the access control ontology.

[5] http://en.wikipedia.org/wiki/Public_key_certificate
[6] http://esw.w3.org/topic/WebAccessControl

Users are specified by web IDs which are authenticated via FOAF+SSL. **Groups** are classes, group membership is defined as

```
webId rdf:type group .
```

This statement is either stored locally, at the server of the resource or remotely at the server of the group URI where it can be accessed via linked data principles. In the latter case, the server of the resource obviously trusts the server that hosts the group definition. `foaf:Agent` is the special group of all users. That means that the rights of this group are granted to anyone, even if they are not logged in.

Access control data is specified per resource, in RDF. It can be stored in a single repository for the complete server or by defining a convention that associates resources with access control files (for example, access control for `/foo/bar.txt` could be defined in `/foo/.meta/bar.txt.n3`). The data itself is stated in the **Access control vocabulary** which specifies

- What can be accessed: either a single resource or a class (set) of resources.
- What access mode is being granted: *read*, *write*, or *control*, where *control* means that one can change the access control information of a resource.
- Who is being granted access: either an agent (specified by a web ID) or a class of agents (a group URI).

The following example specifies that the resource `card.rdf` can be read by anyone and written by `http://example.com/~romeo/foaf.rdf#me`.

```
[ acl:accessTo <card.rdf> ;
  acl:mode acl:Read;
  acl:agentClass foaf:Agent ]
[ acl:accessTo <card.rdf> ;
  acl:mode acl:Read, acl:Write ;
  acl:agent <http://example.com/~romeo/foaf.rdf#me> ]
```

3.4.4 Update protocols

Update protocols allow one to modify a repository hosted at a given URI. Two protocols are often used by linked data applications:

- WebDAV: is used for reading, writing, or deleting files that contain data dumps.
- SPARQL Update: is used for incremental updates to a repository.

WebDAV

WebDAV extends the standard HTTP verbs (OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE, CONNECT) in order to enable complete remote management of a file system. Many operating systems (Windows, Mac OS, Linux, ...) support it in a way where managing remote files is indistinguishable from managing local files. The following verbs[7] are new:

[7] http://en.wikipedia.org/wiki/WebDAV

- PROPFIND: retrieves properties, stored as XML, from a resource. In can also be used to retrieve the collection structure (the directory hierarchy) of a remote system.
- PROPPATCH: changes and deletes multiple properties on a resource in a single atomic act.
- MKCOL: creates collections (directories).
- COPY: copies a resource from one URI to another.
- MOVE: moves a resource from one URI to another.
- LOCK: puts a lock on a resource. WebDAV supports both shared and exclusive locks.
- UNLOCK: removes a lock from a resource.

Note that for most linked data write applications, standard HTTP verbs (mainly GET, PUT, and DELETE) are enough.

SPARQL Update

SPARQL update [SMB+08] extends normal SPARQL with two categories of update operations on a repository. First, graph update operations add and remove triples from one of the graphs in the repository:

- INSERT DATA: adds triples to a graph, as specified directly in the request.
- DELETE DATA: removes triples from a graph, as specified directly in the request.
- MODIFY: can add and/or remove triples from a graph. In both cases, the data is produced from a *template* whose variables are filled in via a query *pattern*.
- INSERT: MODIFY without the (optional) DELETE.
- DELETE: MODIFY without the (optional) INSERT.
- LOAD: copies all triples of a remote repository (which is specified by a URI) into a graph.
- CLEAR: removes all triples of a given graph.

Second, there are operations for graph management:

- CREATE: create a graph.
- DROP: remove a graph.

Example (from [SMB+08]): delete all records of old books (with date before year 2000).

```
PREFIX dc:  <http://purl.org/dc/elements/1.1/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

DELETE {
    ?book ?p ?v .
} WHERE {
    ?book dc:date ?date .
    FILTER ( ?date < "2000-01-01T00:00:00"^^xsd:dateTime )
    ?book ?p ?v .
}
```

Most of linked data fits into the paradigm of *Representative State Transfer* (ReST, [Fie00]) where URLs denote data sources and the HTTP verb GET is used for retrieval, while POST is used for updates. SPARQL update does not adhere to this clear separation. Usually the POST verb is used to send a SPARQL Update request to a SPARQL endpoint, denoted by a URL. Such a request can perform both retrievals and updates.

3.5 Future research: linked data and HYENA

HYENA does not currently support linked data. This section sketches how support for linked data will be built into future versions of it. HYENA/Web serves data as a web application. Thus, URIs contain the state of the user interface. In the future, linked data could be supported in three ways. The first way is to enable FOAF+SSL for HYENA/Web, so that linked data users have single sign-on. The second way is as a linked data client. HYENA would read and write linked datasets. This could happen either directly (e.g. via SPARQL update) or by importing a set into HYENA, editing it there, and writing it back to the original location. The third way is for HYENA to be a linked data server. This means that in addition to accessing HYENA's data via a web application, one could also access it via linked data principles. The web application part could help the linked data part by assisting users with the creation of semantic sitemaps and voiD data.

Chapter 4

Folksonomies and ontologies

Contents

4.1 Overview

This chapter introduces two popular ways of knowledge representation on the semantic web: *Folksonomies*, user-generated systems of tags and *ontologies*, formal, partially self-describing specifications of concepts and their relationships [Wel07]. It then proceeds to classify them as two extremes of expressiveness along the spectrum of knowledge representation systems. CoIM uses folksonomy-based meta-data to organize data.

4.2 Folksonomies

To retrieve entities such as documents, they have traditionally been annotated with keywords taken from a pre-defined vocabulary that has been created by domain experts. Thus, if one is looking for entities that are related to a given topic, one can find them if one of the keywords matches the topic.

With the web, everyone can publish content and using keywords for retrieval becomes harder, because they cannot be managed in a centralized fashion any more. This is due to the fact that the number of entities being added to sites such as Flickr is simply too great to be monitored and managed by a central agency. Thus, many web sites with user-generated content use *folksonomies*: keywords are called *tags* and the collection of all tags is called a folksonomy. Two new ingredients are that users can create their own tags and that the web is used to collaboratively manage a folksonomy. Vander Wal distinguishes two kinds of folksonomies: in *broad folksonomies*, does not only attach tags to entities, but also who has added a tag, so that the same tag can be attached more than once. In *narrow folksonomies*, the person who has tagged the entity is not recorded; usually only the author of an entity is allowed to attach tags. Broad folksonomies allow one to compute how popular a tag is for a given entity. A common way

of visualizing the result is as a *tag cloud*: the tags are displayed as a single continuous text, in alphabetical order. The more popular a tag is, the larger its font size becomes.

4.2.1 Active vocabularies versus controlled vocabularies

Adding metadata in a distributed, uncontrolled manner has the advantage of being cheap and offering several perspectives, in parallel. Such metadata reflects the active vocabulary of a community and is constantly being updated to fit current needs. Lastly, especially with broad folksonomies, one can extract knowledge that is implicit in the system: For example, how different tags are popular at different times permits conclusion as to what developments are important in a community or how it reacts to certain events.

In contrast, controlled vocabularies are less flexible, but they avoid the *vocabulary problem* that folksonomies face: synonyms, trans-language synonyms, spelling variants and abbreviations are not recognized as referring to the same concept; homonyms are not distinguished. This leads to reduced precision and recall when searching folksonomy-organized information.

4.2.2 Reasons for tagging

Not all tags are for categories that are of public interest. Some tags are for personal use only. Al-Khalifa and Davis [AKD07] have grouped tags into three categories in a study:

- Factual tags (62%) referred to the actual content of a document.
- Personal tags (32%) only had personal relevance. Self-referencing tags such as `me` or `my-dog` fall into this category.
- Subjective tags (4%): express opinions such as `cool`.

Golder et al. [GH06] list seven functions of tags:

1. Identifying what (or who) it is about: What is the topic of the tagged entity?
2. Identifying what it is: Especially when entities refer to something, tags such as `book` or `blog-entry` are used.
3. Identifying who owns it: Who owns or created the content of the entity?
4. Refining categories: Some tags do not stand alone, but rather refine other tags. Examples include numbers (`25 people`) or colors (`yellow flower`).
5. Identifying qualities or characteristics: The tagger's opinion is expressed via adjectives such as scary, funny, or stupid.
6. Self-reference: For example, tags that start with "my" such as `mystuff` and `mycomments` can only be understood in relation to the tagger.
7. Task organization: Some tags help with organizing data for performing a task. Examples include `toread` and `jobsearch`.

4.2.3 Retrieval versus exploration

As a method for retrieval, folksonomies lack precision and recall. Both can be improved using natural language processing techniques (such as edit distance between a query and the tag names).

It has been argued [Qui05] that folksonomies lend themselves to serendipity: simply by browsing, one can discover new content. Serendipity can be supported by exploiting that broad folksonomies contain relations between three kinds of entities: documents, tags, and users. For example, users are related if they use the same tags or tag the same documents. Documents can be related by examining who tags them and how. Similarly, it is possible to find related tags.

4.3 Ontologies

The term ontology is difficult to define, because it is used in many different fields, with different backgrounds and requirements: computer science, information science, philosphy, computer linguistics, artificial intelligence, life sciences, bioinformatics, etc. There are two ways of defining the term ontology:

1. as a general concept that subsumes all other kinds of knowledge representation systems (such as thesauri and taxonomies).

2. as a new type of knowledge representation system that goes beyond traditional systems.

The first way is justified by the fact that, while powerful, ontology languages easily scale down to the level of simpler knowledge representation systems and are often used in that capacity. In the following, the author adopts the second way, because it enables one to distinguish semantically richer systems from thesauri, classifications, and folksonomies. Using this approach, one can categorize knowledge representation systems in several ways:

- Availability of vocabulary control: for many, less formal, applications this is the main distinguishing criterion that pits uncontrolled keyword systems against ontologies, thesauri and classifications.

- Expressiveness: Uncontrolled keywords are a very simple system, folksonomies add social dimensions, etc. (Fig. 4.1).

- Complexity of relational constructs: Where folksonomies have no explicit relations at all, ontologies provide specified associations (Fig. 4.2). The next section provides more information on explicit relations.

4.3.1 Explicit relations

One example of an explicit relation is specifying that two concepts are synonymous. Some of the most common kinds of relations used in knowledge representation systems are [WP07]:

- Relations of equivalence: for synonyms. Important for recall and consistent use of a vocabulary.

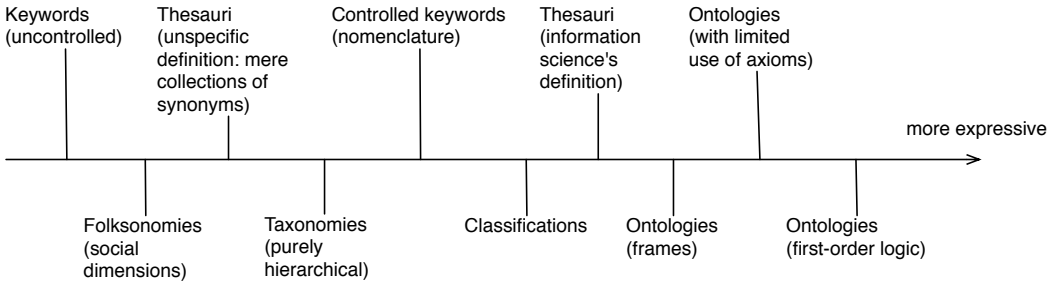


Figure 4.1: Distinguish knowledge representation systems by their expressiveness. Taken almost verbatim from [Wel07].

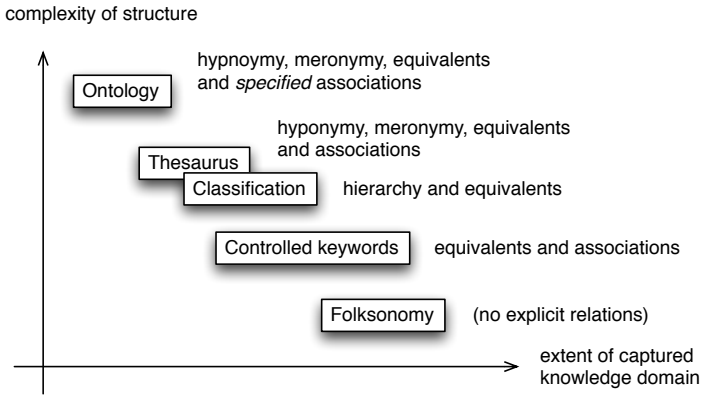


Figure 4.2: Distinguish knowledge representation systems by the complexity of their relational constructs. The more complex the formalism, the harder it is to capture as much of a knowledge domain as possible. Taken almost verbatim from [Wel07].

- Hierarchical relations: for defining the core structure of a knowledge domain. Includes meronymy, and hyponymy. Meronymy is the part-of relation, “lips” is a meronym of “mouth”. Hyponomy is the kind-of relation, “duck” is a hyponym of “bird”.
- Associative relations: other kinds of connections between concepts.

These relationships are used by knowledge representations as follows:

- Controlled keyword indexing: focuses on synonyms for vocabulary control. Hierarchical relations are not used (which would result in a thesaurus). Additional unspecified references may be included.
- Classification: a single kind of hierarchy for concepts.
- Thesaurus: the main focus is on equivalence, on grouping synonyms. Uses the two hierarchical relations meronymy and hyponomy. Includes undifferentiated associative relations.
- Ontology: supports meronymy (usually as a subclass-of relation), hyponomy (usually as an is-a relation), and the definition of any kind of differentiated associative relation.

Chapter 5

Schema and ontology languages

Contents

5.1 Overview

Structured databases such as relational databases usually have built-in schema mechanisms that apply to all the data. In contrast, RDF is a semi-structured database and has no fixed schema (apart from the structure of the triples). Most repositories support schema and ontology languages that extend the core RDF with the ability to selectively declare a schema, whose constraints can be applied per entity/object. This chapter describes the main RDF schema and ontology languages: *RDF Schema* (RDFS), *RDFS-Plus* (a subset of OWL), and the *Web Ontology Language* (OWL). All have encodings in RDF which means that the schema can be stored and delivered together with the data it describes. The intention of the chapter is to convey what the standard RDF schema approaches are capable of. They have limitations when it comes to data modeling. Which is why CoIM includes *REMM schema* (Chap. 14), a schema system for data modeling whose definitions are derived from OWL.

To give a broader understanding of this area, the *Ontology Definition Metamodel* (ODM) is explained. The ODM integrates two worlds: on one hand the world of ontologies, on the other hand the world of modeling as defined by the *Object Management Group* (OMG) for software engineering. Both worlds can profit from each other, ontologies profit from tools for visual data modeling, the OMG modeling profits from how ontologies handle semantics.

5.2 RDFS

The *RDF Schema Language* (RDF) is a relatively simple schema language that focuses on the concepts of classes and properties.

5.2.1 Constructs

RDFS has several areas where one can specify things about RDF in RDF.

Classes

Classes are sets of RDF nodes. Plain literals and tagged literals have implicit classes, datatype literals have an attached type, and resources declare their memberships of classes via the `rdf:type` property. A subclass hierarchy is declared via the property `rdfs:subClassOf`. Predefined classes for resources are `rdfs:Resource`, the class of all resources and superclass of all classes; `rdfs:Class` the class of classes; and `rdf:Property` the class of RDF properties. Classes for literals are `rdfs:Literal` the class of literals and its subclass `rdfs:Datatype`, the class of datatypes. Subclasses of `rdfs:Datatype` are `rdf:XMLLiteral`, the class of XML literal values, and all simple types of XML schema.

Properties

If one takes the predicates of the triples in an RDF repositories as names of relations, then a property is one such relation. A property relates subjects (the *domain* of the property) and objects (the *range* of the property). It is a binary relation and (in general) not a function. Property domains are declared via `rdfs:domain`, the values of this property are classes whose instances have the given property. Property ranges are declared via `rdfs:range`, the values of this property are classes whose instances can be a value of the given property.

The `rdfs:subPropertyOf` property declares that one property is a subproperty of another. If a property P' is a subproperty of a property P then all pairs of resources which are related by P' are also related by P. Sect. 5.2.3 shows this concept written as a rule.

Additional vocabulary

A few standard properties defined by RDFS have special meaning.

- Special properties: `rdf:type` declares class membership, `rdfs:label` specifies a human readable label for a resource, `rdfs:comment` provides a longer descriptive text. `rdfs:seeAlso` states that the object provides additional information about the subject.
- Container vocabulary: Class `rdfs:Container` with subclasses `rdf:Bag`, `rdf:Seq`, `rdf:Alt` and the super-property `rdfs:member` of all container indices. See Sect. 2.4.1 for details.
- Collections: The class `rdf:List`, the properties `rdf:first` and `rdf:rest`, and the empty list `rdf:nil`. See Sect. 2.4.1 for details.
- Reification: The class `rdf:Statement` and the properties `rdf:subject`, `rdf:predicate`, `rdf:object`. See Sect. 2.4.3 for details.

Two other utility properties have not been mentioned before:

- `rdfs:isDefinedBy` states that the object defines the subject.

- `rdf:value` is used for *structured values*; the value is compound and encoded in a resource with properties that hold an amount, a unit of measurement etc. `rdf:value` marks the main property in such a resource (usually the amount).

5.2.2 Model-theoretic semantics

An *interpretation* maps resources and literals to a set of individuals and properties to a set of binary relations. Ground graphs are interpreted as true if for each triple (s, p, o), the interpretation of the pair (s, o) is an element of the interpretation of p (that is, a binary relation). Blank nodes are interpreted as existential variables. If an interpretation evaluates a graph to true, then it is said to *satisfy* the graph.

- *Simple entailment* between RDF graphs: A set S of RDF graphs entails a graph E if every interpretation which satisfies every member of S also satisfies E.

Simple interpretation and simple entailment ignore any meaning that has been given to names via vocabularies. To handle vocabularies, interpretations have to consider more conditions and some property interpretations have to be "pre-filled" (as if several RDF statements were in every graph to be interpreted).

Interpreting the RDF vocabulary

Conditions for the interpretation ensure that `rdf:Property` and `rdfs:XMLLiteral` are used correctly (only resources of type `rdf:Property` can be mapped to a relation, etc.). Furthermore, all URIs in the RDF vocabulary are typed correctly, for example:

```
rdf:type rdf:type rdf:Property .
rdf:nil  rdf:type rdf:List .
```

Interpreting the RDFS vocabulary

For the RDFS vocabulary, a new mapping from URIs to classes is introduced. It is used to express the constraints imposed on relations by `rdfs:domain` and `rdfs:range`. The interpretation of `rdfs:subClassOf` and `rdfs:subPropertyOf` are required to be transitive and reflexive. Finally, domains and ranges are declared for several properties, as are some more types and subclass relationships.

5.2.3 Inference

To make RDFS entailment visible to all clients of an RDF repository, *inference* is used. It creates (read-only) *inferred* statements derived from the asserted statements according to the semantics outlined above. Inference can be written as rules. In those roles, if certain RDF statements are present in the RDF repository, one or more new statements are created (forward-chaining). For example, the transitivity of `rdfs:subClassOf` corresponds to:

$$(a \texttt{ subClassOf } b), (b \texttt{ subClassOf } c) \rightarrow (a \texttt{ subClassOf } c)$$

This means that if there are two statements declaring a a subclass of b and b a subclass of c then a statement is created that declares a a subclass of c. Subproperties lead to the following inference rule:

$$(p' \texttt{ subProperty } p), (s\ p'\ o) \rightarrow (s\ p\ o)$$

Subproperties are useful when a set of resources has a given predicate, but should have another one (instead or additionally). For example: `dc:title` can be used to assign a title for resources such as documents. But, most RDF tools only support `rdfs:label` for human-readable labels. Thus, one simply makes the following declaration and every resource gets a label that is the same as its title:

```
dc:title rdfs:subPropertyOf rdfs:label .
```

The rule for property domains is:

$$(p \texttt{ domain } c), (s\ p\ o) \rightarrow (s \texttt{ type } c)$$

This means that if a resource s has the property p, it is automatically assigned the type c. Note how this works differently from typical schema languages which would signal an error if s has a property p and is not of type c. With inference, the problem is being "fixed" instead and the necessary type is inferred. The inferences caused by domain and range can be used to assign types to resources that are related in a certain way. For example, a `isMarriedTo` b could assign the same type `MarriedPerson` to both a and b.

5.3 RDFS-Plus

Allemang and Hendler describe [AH08] a subset of OWL that contains many useful features without being overly complex: RDFS-Plus.

5.3.1 Properties

By making properties instances of the following special classes, one can control inference.

- Inverse properties: With the property `owl:inverseOf` one can define a property q to be the inverse of a property p. Then for every statement $s\ p\ o$ the statement $o\ q\ s$ is inferred. For example, one could define

  ```
  :childOf owl:inverseOf :parentOf .
  ```

- Symmetric properties: If a property p has the type `owl:SymmetricProperty` then each statement $s\ p\ o$ leads to the inference of $o\ p\ s$. For example, one could define

  ```
  :marriedTo rdf:type owl:SymmetricProperty .
  ```

- Transitive properties: If a property p has the type `owl:TransitiveProperty` then the statements $r_1\ p\ r_2$ and $r_2\ p\ r_3$ lead to the inference of $r_1\ p\ r_3$. For example, one could define

  ```
  rdfs:subClassOf rdf:type owl:TransitiveProperty .
  ```

To distinguish between directly related and transitively related, one can combine subproperties and transitivity:

```
:direct rdf:type rdf:Property .
:trans rdf:type owl:TransitiveProperty .
:trans rdfs:subPropertyOf :direct .
```

The last statement "seeds" `:trans` with all pairs of resources that are related via `:direct` and then infers the transitive closure.

5.3.2 Equivalence

Equivalence is a concept that is needed in the context of heterogeneous distributed data: When two vocabularies define the same class with a different URI or two data sets contain different resources for the same entity, it would be helpful if one could automatically merge them. RDFS-Plus provides the mechanisms to do so.

- Equivalent classes: the property `owl:equivalentClass` is used to declare that two classes are the same.
- Equivalent properties:

  ```
  :p owl:equivalentProperty :q .
  ```

  means that `:p` is the same property as `:q`. That is, every statement whose predicate is `:p` leads to an inferred statement whose predicate is `:q` and vice versa.
- Same individuals:

  ```
  :s owl:sameAs :t .
  ```

  means that `:s` is the same individual as `:t`. That is, every statement whose subject is `:s` leads to an inferred statement whose subject is `:t` and vice versa.
- Functional properties: if a property is declared functional that means that every domain element is related to exactly one range element. This can be used to infer sameness. For example, the following statements

  ```
  p rdf:type owl:FunctionalProperty .
  s p a .
  s p b .
  ```

  lead to the inference

  ```
  a owl:sameAs b .
  ```
- Inverse functional properties: sometimes there is a property whose values are better unique identifiers for resources than their URIs. Or the property enables unique identification of blank nodes. In both cases, such a property is declared of type `owl:InverseFunctionalProperty`. In the former case, it leads to URIs being `owl:sameAs` if their IDs are the same. That is, different resources with the same (property-assigned) ID are effectively being merged into a single resource.

5.4 OWL Web Ontology Language

OWL is an ontology language that has been created to take into account the distributed nature of data on the world wide web. As a consequence, OWL makes several assumptions that are unusual when compared to traditional logic and schema approaches:

- An open world is assumed: When a statement cannot be proven true, it is not considered to be false (which is sometimes called "negation as failure"), because it might still be true somewhere on the web.
- Unique names are *not* assumed: If two names are different, they might not necessarily refer to different entities. Thus, OWL makes no a priori assumption about two names being different. Several mechanisms can be used to prove that two names are equivalent and there are ways of explicitly stating that two names are different or equivalent (Sect. 5.3.2).
- Classes are *not* assumed to be disjoint: No assumption as to whether two different classes are disjoint are made until that fact is either positively or negatively stated or proven. This has the benefit of making modeling more flexible.

5.4.1 Constructs

The following sections describe the constructs available in OWL [B+a].

Classes

Class descriptions are used to construct classes:

- Enumeration: defines a class by enumerating its values.
- Property restriction: defines a class where the values of a single property are restricted regarding their kind or their amount. By subclassing several property restrictions, a class can define something similar to a schema for itself. Restrictions of the kind are called *value constraints* and can specify that all property values are from a given class, that at least one property values is from a given class, or that the property values always include a given value. Restrictions of the amount are called *cardinality constraints* and specify a minimum and/or a maximum cardinality. Restrictions cause inferences similar to domain and range in RDFS.
- Intersection, union, and complement: construct a class by combining other class descriptions with the standard set-theoretic operators.

Class axioms are relations between classes: `rdfs:subClassOf` is the subclass relation, `owl:equivalentClass` declares two classes as equivalent, `owl:disjointWith` declares two classes as disjoint (that is, by default, OWL does not make the assumption that two classes with different names are disjoint).

Properties

All property declarations are part of RDFS-Plus and thus have already been described:

- From RDF schema: `rdfs:subPropertyOf`, `rdfs:domain`, `rdfs:range`.

- Relations between properties: `owl:equivalentProperty`, `owl:inverseOf`.
- Global cardinality restrictions on properties: `owl:FunctionalProperty`, `owl:InverseFunctionalProperty`.
- Logical characteristics of properties: `owl:TransitiveProperty`, `owl:SymmetricProperty`

Individuals

In the abstract OWL model, individuals are defined with individual axioms (also called *facts*):

- Facts about class membership and property vlaues of individuals. When encoding OWL in RDF, these are the non-ontology (a-box) statements.
- Facts about individual identity: OWL does not make the "unique name assumption", that two different names refer to the different individuals. Accordingly, one explicitly declares two individuals as the same via `owl:sameAs` or as different via `owl:differentFrom`.

5.4.2 OWL sublanguages

OWL defines three sublanguages with varying degrees of expressiveness:

- OWL full contains all of OWL's features at the cost of losing some abilities of automated reasoning.
- OWL DL: limits OWL full so that it can be handled by current description logic reasoners.
- OWL Lite: limits OWL DL so that it is in a lower complexity class. This usually makes OWL Lite reasoners simpler to implement and more efficient. OWL Lite provides the basics for subclass hierarchy construction: subclasses and property restrictions. Additionally, OWL Lite allows properties to be optional or required.

5.4.3 Model-theoretic semantics

Apart from expressing OWL DL in description logics, the standard OWL semantics [PSHH04] is model-theoretic. An interpretation maps the (syntactic) OWL constructs to a semantic domain of individuals, sets of individuals, etc. It distinguishes between classes, properties, etc. It maps, for example, classes to sets of semantic individuals and resource-valued properties to relations between individuals. Embedded constructs such as intersections are interpreted as sets that are constructed out of interpretations of their operands. Facts such as equivalentClass lead to conditions on interpretations (for equivalentClass, the interpretations of the given class names have to be equal).

- An interpretation is said to *satisfy* an ontology if the mappings (interpreting URIs as either classes, properties, etc.) in it are correct and all conditions hold.
- A collection of ontologies and axioms and facts is *consistent* if and only if there is some interpretation that satisfies each ontology and axiom and fact in the collection.

- A collection O of ontologies and axioms and facts *entails* an abstract OWL ontology or axiom or fact O' if each interpretation that satisfies O also satisfies O'.

5.4.4 OWL reasoners

OWL reasoners are logic engines (typically using rule-based or tableaux-based algorithms) that provide more services than just inference. Examples of such services are:

- Satisfiability: is a class satisfiable, can it have instances?
- Subsumption: does class A subsume class B, is A a superset of B?
- Classification: given an OWL ontology, compute a subsumption-based class hierarchy.
- Entailment: does one OWL ontology entail another one?

5.4.5 Pitfalls of OWL

There are several pitfalls [dBLPF05] for OWL when it comes to interoperability, scalability of reasoning, and intuitive conceptual modeling.

Problem: interoperability. To accommodate RDFS, the layering of the OWL sublanguages is not hierarchical. On one hand, the less expressive sublanguages OWL Lite and OWL DL are only layered on a subset of RDFS. On the other hand, the most expressive layer OWL Full is layered on all of RDFS, but not on top of OWL Lite and OWL DL. This can impede interoperability between agents using RDFS or OWL Full and agents using OWL Lite or OWL DL.

Problem: scalability. Research on description logic reasoning has so far focused on subsumption inference. Research on optimizing query answering is ongoing. This task involves very complex reasoning, meaning that it does not scale to fulfill its important role on the semantic web.

Problem: conceptual modeling. OWL exhibits a few deficiencies when it comes to conceptual modeling:

- Difference in treatment of abstract and concrete values: Abstract values don't adhere to the *unique name assumption* (that two individuals with different names are different) and an open world is assumed. Concrete values assume unique names and a closed world. Restrictions involving the abstract domain are used to *infer* new knowledge. Restrictions involving the concrete domain are used to *check* whether the knowledge satisfies certain constraints. This distinction makes OWL unintuitive.
- Deriving equality through cardinality restrictions: If a property has more values than its cardinality allows, the values are assumed to be equal. As a consequence, situations that should be treated as errors lead to wrong inferences.

- Deriving class membership through value restrictions: If a property is declared to have a class C as its range and that property has a value that is not an instance of C, OWL infers that the individual is an instance of C, instead of treating it as an inconsistency. Similar to inferring equality, for most applications, disjointness of classes should be assumed unless specified otherwise. In OWL, classes are assumed to be equivalent unless specified as disjoint.
- Limited support for datatypes: The three major limitations are: (1) no negated datatypes ("every integer except 0-3"), (2) no predicates for datatypes (such as $\leq$ for integers), (3) no user-defined datatypes (except enumerated datatypes).

Problem: extensibility. Rule languages are a useful complement to ontology languages and should be based on them. Alas, straightforward extension of OWL DL with rules leads to undecidability and existing rule systems cannot be used for such a language.

Solutions. The literature mentions two solutions for the above mentioned problems. First, OWL Flight ([dBLPF05], whose semantics is based on logic programming instead of description logics and which supports integrity constraints). Second, OWL plus integrity constraints ([MHS09], which adds integrity constraints to standard OWL).

Sect. 14.2 describes a third solution, a simple type system—similar to type systems in data modeling languages and programming languages—for which a translation from OWL is provided.

5.5 Ontology Definition Metamodel (ODM)

The the goal of the *Ontology Definition Metamodel* (ODM, [omg09]) is to provide the modeling technologies of the Object Management Group (OMG) with the formal grounding for representation, management, interoperability, and application of business semantics. Benefits of the ODM include: the availability of several formalisms with varying levels of expressivity; precise semantics; profiles and mappings for exchanging models expressed in one of the formalisms, and for validating and consistency-checking them; the foundations for combining MDA and semantic web technologies to support semantic web services and other declarative, policy-based applications. The ODM can be used for knowledge representation, conceptual modeling, formal taxonomy development and ontology definition. ODM-based ontologies can be used to support interchange of knowledge among heterogeneous computer systems, representation of knowledge in ontologies and knowledge bases, specification of expressions that are the input—or output—of inference engines.

The OMG's metamodeling architecture is called *Meta-Object Facility* (MOF). It has a layered design that usually comprises four layers. These layers are universal, but the following list gives examples for programming languages.

- M3: Meta-meta-model. MOF expressed in itself.
- M2: Meta-model. A modeling language for a specific programming language, expressed in M3. This is typically UML.
- M1: Model. The classes of a programming language, expressed in M2.
- M0: User object layer. The objects of a programming language.

The ODM was not based on the UML 2.0 metamodel for the following reasons. UML operates mainly syntactically. The lack of reliable set semantics and model theory prevents the use of automated reasoners on UML models. UML cannot express objects independently of classes where in most ontology languages, one can start with individuals and add classes later on. Finally, elements of an ontology frequently cross meta-levels. For example, if there is an annotation mechanism for adding meta-data to (object-level) instances, that same mechanism can also be used to annotate classes (which are a meta-level construct). UML only partially supports this feature. Even though the ODM is not semantically based on the UML, it brings UML benefits to ontology languages: a standardized persistence format and tools for modeling, visualization and transformation.

Six *metamodels* provided by the ODM encode ontology and modeling languages in MOF:

- Two formal logic languages: descriptions logics [BCM+03] and Common Logic [iso07].
- Three structure-oriented, descriptive formalisms: RDF Schema, OWL and Topic Maps [PH03].
- Two traditional, software engineering approaches to conceptual modeling: UML2 and Entity Relationship diagramming.

Three *UML profiles* enable the use of UML notation and tools for ontology modeling in: RDF, OWL, and Topic Maps. Finally, mappings between a number of the metamodels are provided. These mappings are expressed in the MOF Query View Transformation (QVT) language [Gro08] for model transformation.

The ODM is not currently used by CoIM, but will be useful if CoIM adds UML-based (possibly visual) schema editing.

Part II

User interface and navigation

This part describes HYENA's user interface and navigation concepts. It begins by giving an overview of the user interface and continues by introducing navigation concepts: faceted navigation is concerned with making entities with attributes easy to navigate. Its ideas can be directly applied to RDF. Tags, label-like meta-data attached to resources, are treated as a special case of facets. Meta-faceted navigation is an extension of faceted navigation that supports *meta-facets*, kinds of values that can appear across attributes. Examples of meta-facets are time and location. In addition to navigating with facets, tags, and meta-facets, HYENA also supports editing them. The last navigation construct that is explained is *assisted querying*, an interactive way of creating queries for the RDF query language SPARQL. *Multi-paradigm search* is the concept of having a single integrated way of performing several kinds of searches (such as faceted navigation and keyword search). The last chapter of this part introduces *title tags*, a wiki-like notation for quickly adding meta-data such as tags and faceted data to resources. The contributions of this part are faceted editing, meta-faceted navigation, and assisted querying.

Chapter 6

User interface

Contents

6.1 Overview

This chapter gives a brief introduction to the user interface of HYENA/Web and the rationales behind it. The responsibilities of the user interface comprise three sets of tasks:

- Handling sets of resources: Given the whole of the repository, one must be able to find relevant resources and display them appropriately. Sometimes, one needs to sort the results alphabetically by name, sometimes one would like to know when something was created or modified, sometimes a visual representation of the structure (how resources are related) is useful.
- Handling single resources: Given a set of resources, one often wants to "zoom in" on the details of a single resource, of which only a summary is shown in set mode. A detailed view of a resource will have to take into consideration that the same resource can be interpreted differently (Chap. 19), as it can be viewed at several levels of abstraction and can have multiple types. In a similar vein, the operations that can be performed on a resource also depend on how it is currently interpreted.
- Complementary tasks: Some tasks are more indirectly concerned with exploring the data. Such tasks include logging in and out, and temporarily remembering interesting resources (similar to bookmarks in a web browser).

The above suggests that a master detail approach be used (Fig. 6.1); the user interface for managing the data of a repository is split into three panes: A master pane shows

collections of resources, a detail plane shows the currently selected resource and a sidebar pane provides context information for both master and detail. An additional challenge facing the user interface is how to accommodate users with different skill sets, ranging from beginners to experts. For this reason, *skill levels* allow HYENA to incrementally introduce features: Beginners can use it as a simple wiki, advance to a wiki plus tags, etc.

Note that while HYENA/Web manages several repositories, one always accesses a single repository at a time, whose name is specified as part of the URL. For more information on the user interface of HYENA/Web and HYENA/Eclipse (which is not covered in this chapter), please refer to the HYENA manual [Rau].

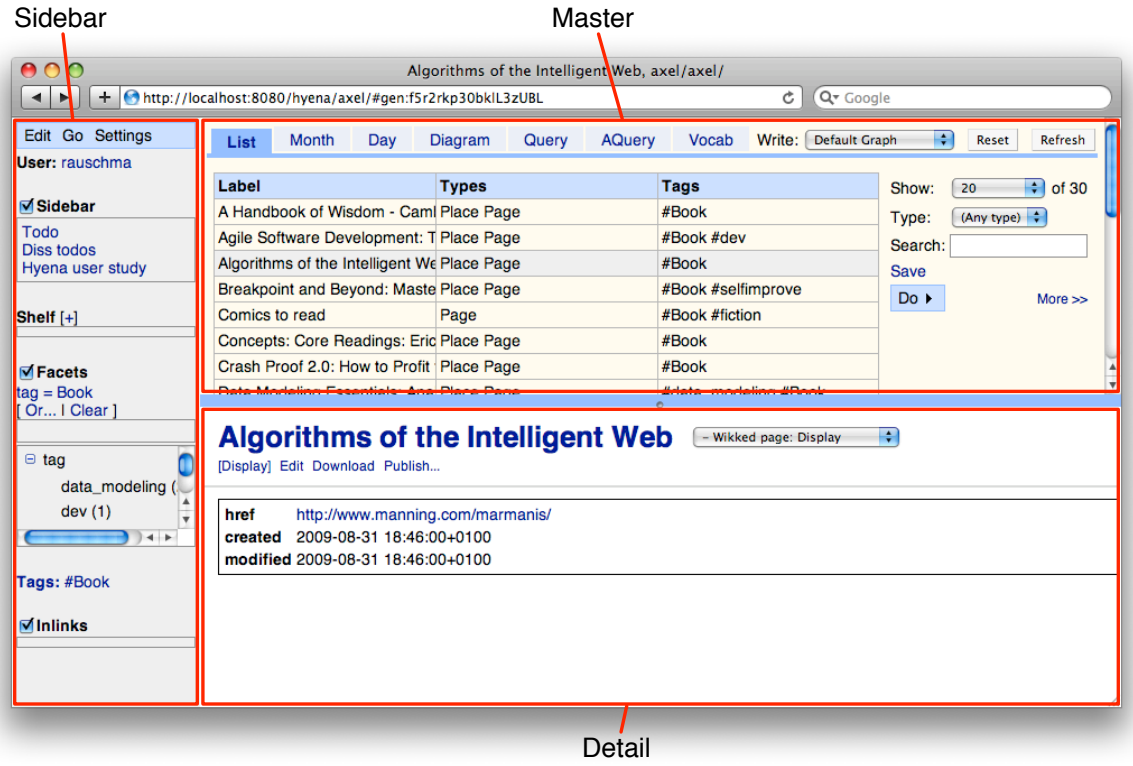


Figure 6.1: The user interface of HYENA/Web. The URL indicates that the name of the web application is `hyena` and that the name of the project is `axel`. No repository name is given, meaning that the root repository (with the same name as the project) is used. The master pane displays collections of resources, the detail pane displays the currently selected resource, and the sidebar displays context information for both master and detail.

6.2 Skill levels

Skill levels are an attempt to solve a problem inherent with generic tools: not every user needs all the features. Skill levels allow one to reveal HYENA's features in an incremental fashion.

- Wiki: HYENA works as a simple wiki.
- Wiki + Tags: shows the resources in the repository; enables tags and faceted navigation.
- Wiki + Tags + Database: Lens-based editing is enabled.
- Wiki + Tags + DB + Semantic web: enables a tab for vocabulary exploration.

6.3 Master tabs

The master pane handles sets of resources. It contains a tabbed folder where each tab corresponds to a different way of generating and visualizing such sets.

6.3.1 Resource set management

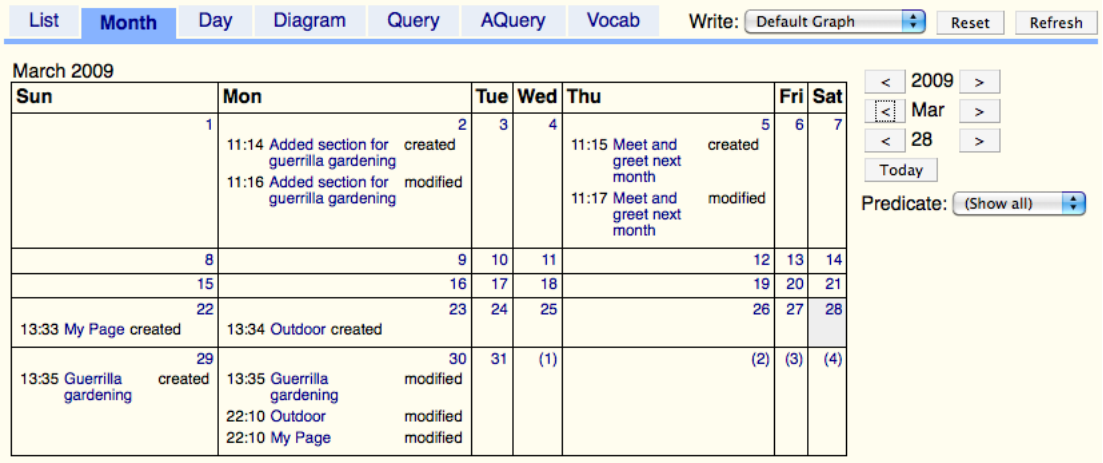


Figure 6.2: Month tab: Shows the current resource set (search results) in a month grid.

The first four tabs represent different ways of editing the same resource set:

- List: This tab shows the resource set in a table. The table can be sorted by any of its columns or filtered by text or by type. The columns are either determined by a lens definition or a default is used (label, types and tags of each resource).
- Month (Fig. 29.3): The same resource set that is edited by the List tab can also be displayed in a month grid. Each set element is displayed according to its *occurrences*: for each of its properties that contains date information, a hyperlink to the resource is shown in the corresponding day, together with the key of the property. The properties (such as `lastModified`, `created`, `due`) can also be used to filter the occurrences.
- Day: This tab allows one to zoom in on the day that is currently selected in the month view. A table displays the occurrences in three columns that mirror RDF statements: The first column displays the label of the resource, the second column displays the label of the property that holds the time information, the

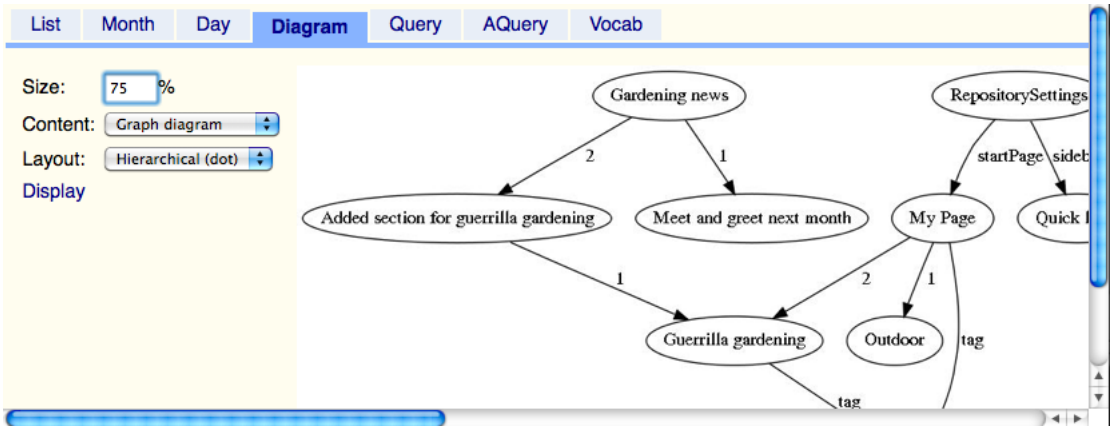


Figure 6.3: Diagram tab: Show the current resource set (search results) as a graph diagram which visualizes how resources are connected.

third column displays the date (and possibly time) itself. In addition to filtering by occurrence property (as in the month tab), one can also choose to show the past (everything before the current day), the present (the current day) and the future (everything after the current day). One example of using this feature is to choose "future" to show upcoming events.

- Diagram (Fig. 6.3): displays the resources in the resource set and their connections as a diagram.

6.3.2 Querying

There are two ways of entering queries: One either uses SPARQL directly or one uses the AQuery tab for *assisted querying* (Sect. 7.6), a form-based way for entering SPARQL.

6.3.3 Vocabulary management

The Vocab tab currently only shows a browsing interface for the URIs in the repository (which indicate the vocabularies that are currently in use). It shows the URIs in a table with two columns: The first column shows the label of the URI, the second column shows tag. Each URI is tagged with the statement component where it appears and its namespace. The statement component can be either one of graph, property, or resource: If a URI appears at least once in the named graph position, it is considered a graph URI. If a URI appears in the predicate position, it is considered a property URI. Otherwise, it is considered a resource URI. Additional free tags can be used to mark important URIs. By default, hyena uses the tag "data" for important data-level URIs such as common classes, the tag "meta" for meta-level URIs such properties that indicate the date of modification, and "config" for its configuration vocabulary. The table can be filtered by tag.

6.4 Detail pane and inspectors

The detail pane shows the currently selected resource and consists of the following elements:

- Title: when clicking, a dialog is displayed with information about the current resource (its URI, its human-readable label, how to embed it, etc.).
- Inspector combo box: allows one to change the inspector that is used for editing and/or displaying the resource. Inspectors are categorized by *presentation mode*: they either display, edit, or configure (meta-data, less common editing tasks).
- Mini-links: a bar that makes commonly used commands available. Examples are: switching the presentation mode, saving, etc.
- Content: as displayed by the currently active inspector.

6.4.1 Inspector selection

When selecting a new resource or when switching the presentation mode, HYENA has to figure out what inspector to show initially. This task is complicated by the fact that there are often many different inspectors; every lens leads to two inspectors, one for displaying data with the lens, one for editing data with the lens. If one selects a new resource, the current inspector should be used again. If that is not possible (because it does not apply to the new resource) then the "best" inspector for that resource should be chosen. If one switches the mode, say from displaying to editing, then the new inspector should correspond in some manner to the current one.

This leads to the following strategy for inspector selection in HYENA. When selecting a new resource, one searches all of its inspectors that have the same presentation mode as the current inspector. It is assumed that the user wants to continue displaying, editing, or configuring. HYENA keeps a history of inspectors that the user has selected manually. Supposedly, because she didn't like the current inspector and wanted to override the automatic choice. If any of the inspectors in the history applies to the new resource, it is used. Otherwise, the best ranked inspector is used. Inspectors rank themselves with regard to the current resource. This is mainly an integer that encodes the distance in the inheritance hierarchy between the type of the resource and the type that the inspector supports. Generic inspectors are registered as supporting `rdfs:Resource`.

For changing the mode, HYENA also relies on history and ranking. But both are overridden if the inspector specifies a related inspector. For example, the inspector for displaying wiki pages specifies the inspector for editing wiki pages as related.

6.5 Sidebar

The sidebar shows context information for the master pane and the detail pane and consists of the following elements:

- Menu bar: with commands that operate on the master pane or the detail pane.
- A login pane: Either the active user is shown or (when logged out) text boxes for logging in.

- Sidebar page: displays a wiki page to make things such as commonly used links available in the sidebar.
- Shelf: is a bookmark-like mechanism to temporarily hold links to important resources.
- Facets: are used for summarizing and filtering the current resource set.
- Tags: for showing and editing the tags of the current resource.
- Inlinks: show what resources refer to the currently selected resource.

6.6 Discussion

The need of managing sets of resources, with the option to closer examine single resources lead to the master-detail design of HYENA. With sets of resources, one faces the challenge of efficiently finding what one is looking for and of displaying search results meaningfully. The former is solved by faceted navigation, full text search, and other filter mechanisms. The latter is solved by several ways of displaying sets of resources: As a list, in a calendar, as a diagram, etc. With single resources, one faces the challenge of efficiently displaying and editing the same resource in multiple ways. Only those operations should be offered for application to a resource that make sense in the current context. This is solved by providing multiple inspectors per resource, by managing intelligently which one of them is selected when first showing a resource, and by context-sensitive *mini-links* for invoking operations that are shown next to the current inspector. Finally, *skill sets* help with hiding some of the user interface complexity when it is not needed.

Chapter 7

Information navigation

Contents

7.1 Overview

This chapter describes HYENA's means for information navigation. First, *faceted navigation* is explained, where one explores a list of entities via the values of their attributes. One sees a summary of the values and can restrict the list to only those entities that have a given value. For example, among a list of songs, one can only show those that were performed by a given artist. CoIM uses faceted navigation for RDF, with the entities being resources and the attributes being properties. Facet-aware RDF editing lets one edit the properties of one or more resources at the same time. When adding a new property to a resource, one is shown the values that the property has for other resources, which helps with quick entry and consistent use of the values. *Tags* are categories that can be attached resources; CoIM treats them as a special case of facets. *Meta-faceted navigation* is a way of navigation when a value set cross-cuts facets. For example, the meta-facet "time" comprises, among others, the facets "time of modification" and "time of creation". Assisted querying is a form-based way of entering a SPARQL query where suggestions help with formulating the query. *Multi-paradigm search* is the integration of several different kinds of searches (such as faceted navigation and keyword search).

7.2 Faceted navigation

Faceted navigation is an efficient way of exploring a set of entities via the values of their attributes. It combines a summary of those values with a way of selecting only those entities that have a given value. Faceted navigation is currently most prominently used in music programs. As an example, consider the following small database of songs:

Genre	**Artist**	**Album**	**Title**
Rock	Beatles	Revolver	Good Day Sunshine
Funk	James Brown	Cold Sweat	Cold Sweat
Rock	Beatles	White Album	I'm So Tired
Funk	Prince	Parade	Kiss

Any of the columns taken as a set is a *facet* of these songs. Thus, the facet "Genre" has the values "Funk" and "Rock". In faceted navigation, a facet plays two roles. First, it summarizes a set of faceted entities: For a set of songs, there are typically less genres than songs, making the list of genres a nice summary. Often a count of how often a given facet value appears is displayed in such a summary. This illustrates how the set of all entities is partitioned by a facet. Continuing the example, the facet-based summary below shows that the songs are half funk, half rock.

Genre	**Artist**	**Album**
Funk (2)	Beatles (2)	Cold Sweat (1)
Rock (2)	James Brown (1)	Parade (1)
	Prince (1)	Revolver (1)
		White Album (1)

Note that not all facets make good summaries: The song title was left out, because there normally is almost a one-to-one mapping between song and song title. The facet "album" does not work well for the example, either (obviously, this is different in real-world music databases).

The second role a facet plays is for navigation. It is based on the observation mentioned above that facets partition the set of faceted entities. For navigation, facet values are interpreted as *restrictions*. Selecting a value shows only those entities where the facet has that value. After restricting the result set, one can continue to refine it by repeating the last two steps: One first computes the facet values as a summary of the restricted, smaller, set and then further restricts it by selecting from these values. For music, this kind of navigation is often hierarchical: One starts with genre, continues with artist, and then selects from the albums of that artist. But it works non-hierarchically, too. Several artists might collaborate for an album. Then it makes sense to look upwards in the "hierarchy": Given an album, what are the artists?

In a slightly improved version of the navigation described above, facet boxes not only contain the current restrictions, but also values that would appear in the box, if it didn't contain any restrictions. The latter values and the restrictions are distinguished by highlighting the restrictions. Without these improvements, if a box contains restrictions, *only* those restrictions are shown. That is, clicking on "Cold Sweat" in "Album" makes all other choices disappear. With these refinements, alternative choices are still selectable. That is, clicking on "Cold Sweat" highlights that choice, but does not hide the other albums. Thus, the highlighted choices are the summary and remaining choices are possible alternatives. Before, all that a box contained was the summary.

Faceted navigation also works well as a complement to keyword text search. Searching for a keyword among the entries can be seen as a form of restriction. The computed summary now tells the user where the text appeared (in which albums etc.).

7.2.1 The facet summary as a tree

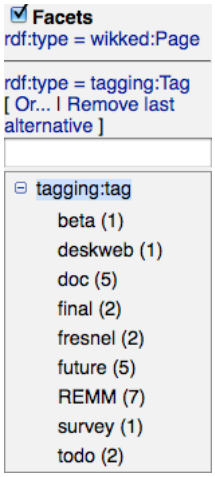


Figure 7.1: Facets as a tree: The `rdf:type` facet has completely disappeared. On top there is the trail of restrictions, a disjunction specifying that pages and tags should be displayed.

HYENA uses a compact version of faceted navigation: Facets are displayed as a forest of trees whose roots are the facet names (Fig. 7.1). The children of the facet names are the facet values. With facet boxes, you can highlight the current restrictions in the boxes, with a tree that is less practical. Instead, HYENA's tree only shows the summary values, without alternatives. The current restrictions are not part of the tree either, but shown separately, where they can be undone. They form a navigational trail, similar to paths in hierarchical navigation. Whereas until now, all restrictions were conjunctions ("and", an entity has to contain all restrictions to be shown), HYENA also supports disjunctions of conjunctions ("or", an entity has to contain either of several sets of restrictions). For example, when one wants to list all entities whose type is either "Page" or "Tag", a disjunction is needed (Fig. 7.1). Negating a restriction (selecting all entities that do *not* have it) is also supported.

An advantage of tree-based faceted navigation is compactness. Furthermore, the restriction trail gives a sense of location and makes it relatively easy to support disjunctions. An advantage of facet boxes is that one always sees the alternatives to the current restriction. With a tree, one has to remove a restriction to see alternatives. Accessing facet values is also simpler with boxes, as they have a more static layout and one does not need to expand a tree node to see them. HYENA remembers what facet keys are expanded to partially mitigate this disadvantage.

7.3 Defining and editing RDF facets

CoIM uses faceted navigation for RDF data. The entities to be navigated are resources, their attributes are their properties. As it is impossible to predict what properties make good facets, CoIM lets the user specify what properties should be considered by faceted navigation.

7.3.1 Simple facets

For resources, the simplest kind of facet is to use the values of a given property as facet values. For example, the values of `rdf:type` make a good facet. Simple facets are specified by the URI of the property. Editing facets works the same for a single resource and for sets of resources: Facet values can be removed and added. When adding a facet value, we can support the user by presenting her with a set of values to choose from. In case none of those values are appropriate, we need to give her the option to create and assign a new value. According to these requirements, a simple facet specification comprises the following components:

- Property URI: the name of the facet.
- Default values: for a user to choose from when he adds a facet value.
- Value types: If a facet value does not exist yet, it has to be created. The value types specify the class(es) to instantiate in such a case. If the facet values are literals, the facet type will be a subclass of `rdfs:Literal`. Instantiation then means entering the text of the new literal.

There are several ways of specifying default values and value types:

1. Not at all: If there are no explicit default values, HYENA uses the current values of the facet. If no value types are specified, then new values are created as untyped resources.
2. Range classes: given a set of classes, the default values are the instances of those classes, the value types are the classes.
3. Default value sets: HYENA defines IDs for sets of default values.
   - `remm:primaryClasses`: The primary classes, as defined by the currently active Fresnel group.
   - `remm:propertyValues`: The same as the default.

Note that the last two ways are not mutually exclusive.

7.3.2 Broad facets

The idea of broad facets is that for every assignment one makes, there is information *about* that assignment to be recorded. The most common use case is collaborative tagging (Sect. 7.4.4): If several users tag, it makes sense to record who did the tagging and when. Alas, RDF support for annotating edges in the graph is not very capable (contrary to, for example, topic maps [PH03]). To remedy the situation, HYENA uses the RDF pattern "single-target edge" (Sect. 12.3.1): A resource *manifesting* the edge is inserted between the source and the target. This indirection makes facet definitions more complicated. Data to be specified:

- The source predicate, which points from the source to the edge.
- The type of edge resource.
- Automatically generated properties of the edge resource. For example, adding the user that is currently logged in as the value of `dcterms:creator` or adding the current date and time as the value of `dcterms:created`.
- The target predicate connects the edge and the target. The default is `rdf:value`.
- Default values, value types: are used and specified like for simple facets.

7.4 Tagging

Tags are categories that are attached to entities and a popular way of organizing data on the web. For example, Flickr[1] uses tags for photos, Delicious[2] uses tags for bookmarks. CoIM treats tags as "facets without a key". The tags of a resource as well as its facets can be displayed and edited in title tag syntax (Chap. 8), which follows the pattern

```
#facetkey1=facetvalue1 #tag2 #tag3
```

When displaying the tags of a resource, one can click on a facet value or a tag to restrict the current resource set to similar resources. During editing, one can start typing a tag name and is presented with possible completions. When committing the changes, all tag names are looked up. If a tag with precisely the specified name exists (ignoring case), then it is attached to the resource, otherwise a new tag with this name is created.

7.4.1 Operations for tag editing

To support the above mentioned way of editing and navigation, the following operations are needed.

Operations on tags

- Find tag by name: either via its exact name or via prefix. The former is needed when entering a tag name, the latter is used to offer suggestions if the user has not yet finished typing in the name.
- Find tagged resources: Restrict the current resource set so that only resources with a given tag are shown.
- Create a tag, given a name: Used if there is no tag that already has that name.

Operations on tagged resources

- List tags: what tags are attached to a resource?
- Add tag: to a resource.
- Remove tag: from a resource.

[1] http://www.flickr.com/
[2] http://delicious.com/

7.4.2 Simple tags

The simplest way of handling tags is to attach them to a resource with a single standard property. A complete specification for this kind of tagging comprises the following values:

- Tagging predicate: the property with which to attach a tag to resource.
- Tag type: the type of tag resource.
- Tag name predicate: the property where the name is stored. The default is `rdfs:label`.

The above mentioned operations for tagging are implemented as follows. Finding a tag means finding tag resources whose name is equal to or starts with a given text. Finding tagged resources means finding resources that have a tag property whose value is the tag. Creating a tag means instantiating the given tag type. Tagged resource operations are handled similarly. While the tagging predicate is normally hidden, it is used as a key in the facet tree. The facet values are the tags, allowing one to use faceted navigation for tags.

7.4.3 Several tagging predicates and types

If several tagging vocabularies are to be mixed, things become more complicated. We now need to ask for the type of the tag when creating a tag and when adding a tag to a resource. This assumes that there is a unique mapping from a tag type to a tagging predicate and a tag name predicate (for labeling).

7.4.4 Broad tagging

Richard Newman's tag ontology [New], the archetype of RDF tag ontologies, introduces *taggings*, resources that denote a single assignment of a tag. The ontology description states:

> Taggings reify the n-ary [sic] relationship between a tagger, a tag, a resource, and a date.

One of the requirements for HYENA's broad facet mechanism (Sect. 7.3.2) was to support broad tagging. It is thus powerful enough to handle this scenario; tagger and date are edge properties.

7.4.5 Managing tags

When working with tags, there usually comes a time when the current set of tags has to be changed: Two tags mean the same thing and should be merged; the meaning of a single tag should be refined by splitting it into two tags; or a tag should be renamed, because its current name does not reflect its meaning any more. In HYENA, one does not need special operations for this, because the existing editing mechanisms can be used:

- Merging tag u into tag t. This is handled by first deleting u and then renaming its URI to the URI of t. As this changes every occurrence of URI u into URI t, it has the desired effect: each attached u is changed to t.

- Splitting a tag t into two tags u_1 und u_2: Facet changes can be applied to sets of resources. Thus, one first lists all resources that are tagged with t. Then one selects all those resource that should be tagged with u_1, removes t from those resources and assigns the new tag u_1. Finally, one repeats this procedure for u_2 or one simply renames t to u_2.
- Renaming a tag: is trivially simple. HYENA always refers to a tag by URI and looks up the name on the fly when resolving a tag name or displaying a tag. Thus, to change the name, one changes the name property and from then on, the tag will be found under the new name and displayed with it.

7.4.6 Grouping tags

Often, there is not one homogeneous group of tags, but several subgroups that are used orthogonally. For example, in collaborative tagging systems, one often has tags such as `todo` and `paris`. But these are different kinds of tags. The former refers to the status of a tagged entity, while the latter refers to a related location. Writing them as `status = todo` and `location = paris` brings us back to facets. And one definition of "facet" is in fact "group of tags". The migration from tags to facets is performed incrementally: A tag can be specified to have a *tag predicate*. This predicate becomes a facet and the tag is henceforth not added with the default predicate, but with its own tag predicate. More than one tag can use the same tag predicate. When listing the tags of a resource, the tag is displayed as

tag predicate = tag

This makes the grouping immediately obvious. With broad facets, the tag predicate becomes clickable and leads to the edge resource.

7.5 Meta-faceted navigation

Abbreviated and informally, faceted navigation can be written as

entities + restrictions $\rightarrow$ results + summary

Meta-facets are based on the idea that some data one is looking for cannot be found in a single facet, but in several ones. Two examples are "time" (facets "created", "modified") and location (facets "born", "residence"). Navigation for meta-facets extends the constructs for faceted navigation as follows: Each meta-facet restriction filters all property values. That is, values determine the result, property keys are completely ignored. For example, a time restriction is often an interval with informal semantics such as "all resources that have an attached date within December 2009". The results do not contain just resources any more, but rather *occurrences*. An occurrence is a triple consisting of subject, predicate and object. Two possible results for the above given time restriction are

```
(:r1, dcterms:created, "2009-12-6"^^xsd:date)
(:r2, dcterms:modified, "2009-12-30T21:32:52"^^xsd:dateTime)
```

The following extensions are necessary for meta-faceted navigation:

Restricting occurrence predicates: Occurrences can be restricted by occurrence predicate. For example, to only show the dates of last modification and not the dates of creation.

Broad facets: For broad facets, one can add the edge resource to the occurrence, allowing the user interface to display a link to it.

Keyword search: Text is also a meta-facet. Restricting by occurrence predicate (where the text has been found) is useful. The occurrence does not contain the complete object, but only an the text that has been found and a few words surrounding it to provide context.

Integration with faceted navigation: Meta-faceted search usually starts by listing all occurrences in the current search results. For example, to display all time information on a month grid. So the meta-facet "filter pipeline" starts by converting a set of resources to a set of occurrences. These occurrences can then be further filtered. If the final result is to be displayed at the same time by meta-faceted navigation and, say, faceted navigation, then one has to convert the occurrences back to a set of resources.

Further details on meta-faceted navigation are given in Sect. 9.6 on the search calculus.

7.6 Assisted querying

Assisted querying provides a form-based way for entering SPARQL queries. When it comes to filling in text boxes in a triple pattern, assisted querying can show possible values (*expansions*) for that box. This can be used to quickly perform small queries. For example, after filling in the subject box, showing values for the predicate box answers the question: "What properties does this resource have?" After entering the URI of a resource r into the object box, retrieving the values for the subject answers the question: "What resources refer to r?". Because SPARQL queries are used to find expansions, this feature works for an arbitrary amount of triples patterns and is useful for navigating long paths. The assistance being given helps with learning the structure of the data and with formulating the query.

7.7 Multi-paradigm search

The goal of multi-paradigm search is to combine several search paradigms into a single solution. Synergistic effects between paradigms usually make this solution more powerful than the sum of its parts. The paper "Collaborative multi-paradigm exploratory search" [TB08] lists the following search paradigms:

- Keyword-based search: for keywords in the content or the meta-data. Sometimes both results are combined, and meta-data results highlighted (because it is assumed that they are more interesting to the user).
- View-based search: interactively guides the user through the search process by visualizing the current result and offering next steps for refining the query. Faceted navigation is a kind of view-based search.
- Query by example: Apart from the classic relational query by example (where one enters values into the columns of a table), newer variants return results that are similar to a given instance. Note that this works both for images, textual data and structured data, after having defined appropriate similarity measures.

CoIM performs multi-paradigm search by integrating simple search criteria (such as "what graph is the resource in?") with keyword-based search and view-based search (facet-based, meta-facet-based). Query by example is not part of CoIM multi-paradigm search; assisted querying is similar, but handled separately.

7.8 Running example

Shenzi wants to add ideas as wiki pages. First, she prepares some meta-data to help with organizing and navigation, then all authors create the data and finally, she navigates the data. The next chapter explains how the meta-data mentioned below can be added quickly to a resource.

Create the meta-data. First, she creates resource for the tag `#idea`. Then she creates three resources representing the co-authors. Finally, she declares the property `event:agent` to be a facet.

Create the data. For each idea, a co-author creates a wiki page. It is marked as an idea by assigning it the tag `#idea`. The co-author records himself or herself as the creator of the page by adding his or her resource as the value of property `event:agent`.

Navigate the data. All ideas are listed by showing all resources with tag `#idea`. All ideas by a particular author are listed by additionally restricting the facet "agent". To see all ideas created or modified by a particular author in a particular month, one switches to that month in the month view. As all wiki pages are automatically annotated with the date of creation and last modification, they appear in the appropriate day of the month (similar to Fig. 29.3). Each calendar entry shows whether the wiki page has been created or modified on that date. A combo box that contains all the (time-related) predicates that appear in the month view can be used to just show when pages have been created.

7.9 Future research

Dynamic category sets. Dynamic category sets [Tun06] are the result of the observation that when searching faceted data, the query text might not be contained in a single facet value, but rather spread out across several facets. For example, a user might search for "20th Century Fiction", thinking of it as the value of a single facet (say, "title"). But a matching entity might be faceted as having a "time period" of "20th Century" and a "Genre" of "Fiction". This is the vocabulary problem [FLGD87] transferred to facets: end users might mentally partition the information space differently from the indexers that created the facets. The paper proposes to make keyword queries independent of how the facets are organized by demanding that the keywords be contained in the union of all facet values and not just in a single one.

Structuring facet values. Similar to faceted browsing in /facet (Sect. 27.4), HYENA should support hierarchical facet values. For example, the values of `rdf:type` could be displayed as the tree of the class hierarchy. If a facet has many (flat) values, grouping

enhances usability. That is, one introduces a tree structure. For example, years can be grouped into decades (1960-1969, 1970-1979, etc.). Often, the group members are not even enumerated in the facet summary.

Improved navigation. /facet can also relate two result sets. For example: "What pictures of Rome have been painted by a French artist?". This query relates the pictures of Rome and French artists. /facet's approach is relatively simple and type-based and would have to be refined. The relation browser [Mar06] can preview the facet tree that would result when selecting a restriction. This is shown as a tooltip and very useful.

Advanced search. Intelligent information retrieval algorithms could be used to find similar resources or related resources.

More facet visualizations. The calendar and day tabs are the first facet-specific visualizations. More should be added. For example, geographical maps or time-lines.

Facet inferencing. Inferred facets are especially useful for tags. For example, one might want to automatically tag each resource with "Programming language" that is tagged with "Java".

7.10 Discussion

This chapter described faceted navigation and brought several contributions. It has shown how tags and facets are related and how one can incrementally migrate from the latter to the former. Facet editing and the integration of broad facets are another contribution. As is meta-faceted navigation, which complements and extends faceted navigation. Combining several search paradigms is called *multi-paradigm search*. Sect. 9.6 provides a formal foundation for HYENA's multi-paradigm search, the so-called *search calculus*.

Chapter 8

Title tags

Contents

8.1 Overview

Title tags are used in three capacities in HYENA. First, when creating a new wiki page, one can enter the title and add meta-data to the new page at the same time. For example:

```
Feed the bird #todo
```

The word `#todo` at the end is a tag expressed in title tag syntax. This input leads to a new page being created that has the title "Feed the bird" and is tagged with *todo*. Second, the same syntaxs (without the page title) is used to later edit the tag. Third, the tags of a resource are displayed with title tags syntax. This chapter explains the title tags syntax and a complementary simple notation for date and time.

8.2 Basics

The grammar for title tags is as follows.

tag ::= #⟨category⟩ | @⟨person⟩ | >⟨location⟩
facet ::= #⟨facet-key⟩=⟨facet-value⟩

Tags can appear either inside the text of the title or after it. In the former case, tags are added to the title and less typing is needed. For example:

```
I will go to #Munich
```

The title becomes “I will go to Munich”, and a tag is added, too. This syntax is similar to conventions established by Twitter. More Examples:

- `Meet @Jack >home #time=tue`
- `Lunch with @Jill #loc=Ye Olde Tavern`
- `Call back Fineas #todo`
- `http://example.com/article.html #read`

The last example shows support for bookmarking: If a title tag text starts with a URL, a bookmark resource is created and the title filled in by downloading the web page and extracting the title. The tags and facets after the URL are added to the resource.

8.3 Attaching meta-data

The meta-data generated by title tags is based on the *Event Ontology*[1]. The following facets are predefined. Note how some tags are synonyms for facets.

Facet key	**Predicate**	**As tag**
#due=⟨time⟩	cal:due	
#time=⟨time⟩	event:time	
#loc=⟨location⟩	event:place	>
#who=⟨person⟩	event:agent	@

Names of categories, agents, and locations are resolved as follows. HYENA searches the labels of instances of `tagging:Tag` (categories), of `foaf:Agent` (agents), or of `geo:SpatialThing` (locations). If one of the labels exactly matches the name, the corresponding resource is used. In addition to labels, `foaf:nick` and `hyena:shortName` (a kind of nick for locations) are also considered. If no exact match is found, HYENA creates and uses a new appropriately typed resource that is labeled with the name.

8.4 Simple time notation

The following notation is used to make time easy to enter for humans. Alternatively, one can enter a keyboard shortcut to bring up a date chooser (a small calendar) and insert a date.

- 2008-10-22
- 10-22 (next October 22)
- oct-22 (next October 22)
- tue, tuesday, +tue (next Tuesday)
- -tue, -tuesday (last Tuesday)
- tod, tom, yes (today, tomorrow, yesterday)

[1] http://motools.sourceforge.net/event/event.html

- today, tomorrow, yesterday (today, tomorrow, yesterday)
- +1, -3 (tomorrow, three days ago)

Time (time symbols are disjoint with date symbols):

- 15, 2pm, 10am (15:00, 14:00, 10:00)
- 6pm:30, 16:30 (18:30, 16:30)

8.5 Running example

The previous chapter mentioned that the co-authors store ideas in wiki pages. Such wiki pages can be created quickly with title tags. The following is an idea added by Ed:

```
Add overview as mind map #idea @Ed
```

This creates a wiki page that is tagged with `#idea` and records Ed as a related person.

Shenzi can add bookmarks via title tags, too. A URL plus tags leads to a bookmark being created and tagged. Additionally, HYENA has an operation for caching the web page referred to by a bookmark in the file system. This ensures that the bookmarked content is always available, even when the Internet cannot be accessed, a server is temporarily offline, or the site structure has changed.

8.6 Discussion

Title tags are an important puzzle piece to enable efficient note taking. Chap. 29, "Integrating structured and unstructured data", explains all considerations in this regard.

Part III

Foundations

This part describes the foundations of connected information management (CoIM). First, a model for CoIM is presented. It is an informal model, a more abstract way of looking at the CoIM implementation HYENA. Its goal is to integrate various kinds of structured and unstructured data such as data entries, text, files, etc. The CoIM model builds on RDF and extends it with the following contributions: wiki markup as a formal language that integrates structured data; the means to keep proxies for external data up to date; and a *search calculus*, a formalization of information navigation.

Few comprehensive models of this kind exist in the literature. KiWi's model [SEG+09] is similar to HYENA's. It does not have HYENA's formalization of navigation or external references. KiWi's model is based on *content items*, a mixture of XML (wiki) and RDF (data). Content items are more powerful than CoIM markup (queries can refer to the structure of XML), but also less standard, making interaction with external data sources more complicated.

The next chapter introduces *Wikked*, the CoIM wiki markup language, in detail. Wikked is a hybrid of the line-based wiki markup standard Creole [wika] and LaTeX, whose constructs can be nested. This hybridization introduced challenges that lead to the adoption of staged parsing in HYENA. The result of parsing is an abstract syntax that needs to be evaluated and can then be translated to an external markup language for display or printing. Examples of external markup languages are HTML and LaTeX. The author is not aware of any other wiki syntax that has a formal definition. HYENA's use of LaTeX has proven useful when integrating external LaTeX-based work flows: Bullet lists (in wiki syntax) created for brain storming can be exported to LaTeX, fragments of LaTeX code can be used in HYENA. *Templates* are fragments of Wikked markup with blanks that are to be filled in by RDF data. Templates are used as a flexible means for displaying RDF data. Given the generality of Wikked markup, it was easy to extend it for this task.

Finally, RDF patterns are described. These patterns collect how HYENA uses RDF for configuration and works around some of RDF's limitations.

Chapter 9

A model for connected information management

Contents

9.1 Overview

Currently, the data one encounters in one's digital life is stored in disconnected islands: Data entries are kept in separate databases such as an address book application for contact data, a browser for bookmarks, a calendar for events, BibTeX files for publication data; unstructured text is stored in text files or wiki pages; files such as images or PDF files are stored in the file system. There is no way to consolidate this data or to organize and navigate it in a manner that crosses islands. The goal of connected information management is to build bridges. This chapter presents a model that allows one to manage all kinds of information in a centralized location. The data can be organized by uniformly attaching meta-data to it. A special *data calculus* complements the model to perform information retrieval.

The main unit of management of the CoIM model is the *project* (Fig. 9.1): It consolidates all data used for a particular purpose and is also the granularity of authorization. The foundation of a project is an RDF repository[1]. RDF is ideally suited to store most structured data, including the examples given above. The CoIM model extends RDF to integrate the remaining kinds of data. The first extension is to allow the remaining

[1]The current implementation of HYENA still supports several repositories per project. This chapter describes the simpler and more elegant approach of having one repository per project.

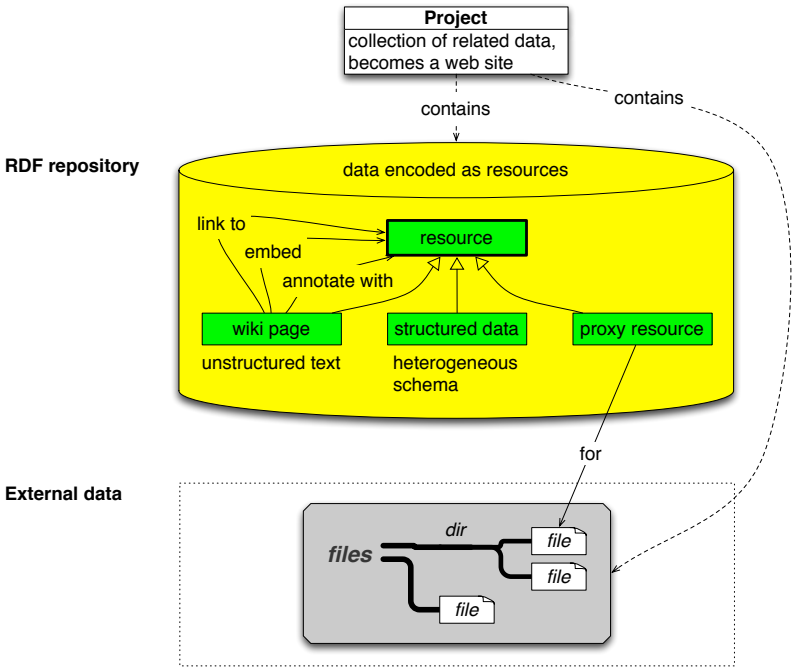


Figure 9.1: A project contains an RDF repository for managing all of the data and some external data that does not fit into the repository, but is instead represented by proxy resources.

kinds of data to become RDF. The relevant entities are *manifested*[2] as RDF resources: Each wiki page is a resource and the wiki markup is stored in a property. Data that cannot be efficiently stored in RDF is called *external*. External entities are manifested as *proxy resources* pointing to those entities. The second RDF extension is a set of events which can be reacted to, ensuring the integration and consistency of the manifested resources:

- Resource post-processing: is performed after saving a resource to RDF. It is used for tasks such as updating the time of last modification and to make references in wiki markup visible in RDF.
- URI renaming: is a user interface operation that is invoked by the end user. In addition to changing the RDF, the URIs in wiki markup have to be updated.
- Derivation update: The content of some named graphs is derived from other data. Examples are graphs with proxy resources which are derived from external data or graphs with inferred data which is derived from RDF data. A derivation update is explicitly invoked by the user and recomputes the derived data. The CoIM model assigns *kinds* to named graphs which determine the semantics of their data and thus how to perform the derivation update.

Projects achieve closure regarding their most important kind of external data, files, by including sets of files in addition to RDF data. Synchronizing projects between installations synchronizes both files and RDF data.

[2]The verb *to manifest* is used instead of *to reify*, to avoid confusion with RDF reification.

The *search calculus* has been created to support navigation of all CoIM data according to the principles of *multi-paradigm search* (Sect. 7.7) which combines keyword-based search, faceted navigation, meta-faceted navigation and more.

9.2 Requirements

The data managed by the CoIM model falls into three categories:

- Structured data: includes object data (such as contacts, bookmarks, events), meta-data (the time of creation of a contact) and configuration data (the wiki page to be shown after a CoIM system has started). Note that meta-data is cross-cutting, it exists in addition to other data. Some of it should be added automatically; conceivable are timestamps and location.
- Wiki pages: are text with references to data. Examples are notes and outlines.
- External data: everything that is not stored in its entirety in the RDF repository. An example is files.

The following principles must hold when managing this data:

- Uniform annotation: All annotations, including meta-data, should be attached to entities in the same way and be compatible with each other. For example, all entities should attach tags in the same manner and use the same tag vocabulary.
- Uniform semantic references: Whenever an entity managed by CoIM refers to another element, it should use the same mechanism to do so. Each reference should be labeled, to indicate its semantics. Knowing who references a given entity is important for navigating the data. The label is crucial for wiki pages which can, for example, embed another entity or link to it. In the former case, the referenced entity becomes part of the page, in the latter, it is just linked.
- Consistency of unstructured content: Whenever annotations or other data is derived from the content of an unstructured entity, measures should be taken so that derivation and content stay consistent. For example, if the date of last modification of a file has been made available to a CoIM system as meta-data and the file changes, then the meta-data has to be updated accordingly.
- Multi-paradigm search (Sect. 7.7) for the data has to be supported. In CoIM, this kind of search integrates keyword-based search, faceted navigation, meta-faceted navigation and more.
- Uniform navigation: When it comes to browsing and searching the data, all entities should be treated the same. This requirement is relatively simple to fulfill if annotations are uniform, as they will be used for organizing entities.

Wiki pages are a special kind of unstructured data, because in addition to being referred to, they can also refer to other data. Three kinds of references are predefined and lead to different ways of presenting the referenced entity: a *link* is shown as a hyperlink whose text is the name of the referenced entity; an *embedding* leads to the referenced entity being displayed in place; an *annotation* is displayed as the icon of a post-it note that shows the entity in a pop-up window if clicked on. References can appear anywhere in styled text which makes wiki pages ideal for collating and presenting data.

The requirement of "uniform semantic references" applied to wiki markup means that references need to be translated to the same labeled format used by all entities. Similarly, "consistency of unstructured content" demands that the result of the translation has to change if the wiki page changes and that the wiki page has to change if the translated references change.

9.2.1 Notation

The following notation is used in this chapter:

- Node denotes the set of RDF nodes.
- Res denotes the set of RDF resources.
- Uri denotes the set of RDF URIs.

9.3 Projects and repositories

The main packaging construct of CoIM is the *project*. It consolidates all the data mentioned above and allows users to "take it with them". Data access is authorized at the project level. A project is a tuple (R, g, F) where

- $R \subset (\mathrm{Uri} \times \mathrm{Res} \times \mathrm{Uri} \times \mathrm{Node})$ is the RDF repository, a set of *quadruples* (*quads*). An alternative name for a quad is *statement*.
- $g : \mathrm{Uri} \to \mathrm{GraphKind}$ assigns kinds to graphs. The set $\mathrm{GraphKind}$ is defined in Sect. 9.3.1.
- F is a set of files.

The RDF repository is the core of a project, it is where all the data is integrated. This is done by manifesting each entity as an RDF resource. Event operations are invoked in reaction to certain events and extend RDF so that requirements such as consistency can be fulfilled. The next section describes named graphs which are used to assign semantics to sets of resources which is necessary for the operations to work correctly.

9.3.1 Named graphs

Named graphs group sets of triples and are assigned a *kind*. The kind defines the semantics of the content of a graph and thus how the content is processed by the standard operations. The grouping is done by adding a fourth component to each triple, the URI of the graph. This component is ignored when working with data, so one still processes triples as before, but it can be used to retrieve all triples in a given graph. Named graphs contain

- generated data (such as proxies for external data). encapsulated, so that it can easily be re-generated.
- imported data to facilitate re-importing.

- framework data. Extensions of the CoIM framework can provide their own RDF data that becomes part of each RDF repository. This is often meta-data such as classes or lenses. Apart from this data being read-only to users, no other distinction is made between framework data and data created by end users. Framework data is stored in separate named graphs.
- user data, allowing users to group their data; for example, by topic.

In the future, named graphs will be used

- to integrate external linked data sets (Chap. 3) and
- to separate inferred data from asserted data (Chap. B).

The purpose of a graph is defined via the function g which maps graph names to graph kinds. This set is defined as

$$\mathrm{GraphKind} = \{\mathsf{framework}, \mathsf{user}\} \cup \mathrm{ExtKind}$$

Thus, a graph can contain framework data, user data, or (proxies for) external data. Sect. 9.5.3 contains more on external data and the set $\mathrm{ExtKind}$. A graph can have the following properties:

- Writable: Only user graphs are writable. Framework data is managed by the framework and proxy resources for external data are generated automatically.
- Removable: User graphs and graphs with proxy resources can be removed.

9.4 Event operations

To guarantee the consistency of the data, extra work has to be performed during some events in the life cycle of the data. This sections describes the events and the work.

Resource post-processing. After a resource has been saved, it is post-processed to ensure its consistency. This includes updating the date of last modification and adding the references in wiki markup (see below). The former is different from similar data that is derived for files: If a non-external resource has the property `hyena:updateModified` then its time of last modification is continually updated. If that property is encountered and a resource does not have a time of creation, then that time is added.

URI renaming. This operation is explicitly invoked by end users. Renaming a URI t to a URI u means replacing all occurrences of t in RDF with u. If t appears in the content of unstructured data such as wiki pages, then more work is necessary. See below for details on wiki pages.

Derivation update. Some named graphs don't contain original data, their data is derived from other data. Proxies resources for external data are one kind of derived data. To be consistent, their meta-data, which includes their path in the file system and the date of last modification, should be updated from time to time. This update is triggered explicitly by the user and leads to each graph with proxies being brought up to date. In the future, when RDF inferencing is implemented (Chap. B), the derivation update can be used to update the inferred data.

9.5 Manifesting entities as resources

Manifesting the various entities as RDF resources is key to integrating them in the repository. This section describes how this is done.

9.5.1 Structured data

Structured data is usually represented as entities with attributes which is trivial to express in RDF [MAK+04]. RDF can accommodate heterogeneous data, because there is no central schema.

9.5.2 Wiki markup

In this chapter, wiki markup is treated as an opaque text string from which a set of references can be extracted. More details on wiki markup are given in Chap. 10. To manifest a wiki page as a resource, one creates a resource with type `wikked:Page` and stores the markup in a property `wikked:content`. The value of this property is a literal with the datatype `wikked:Markup`. The requirement of "uniform semantic references" is fulfilled by writing the references inside the markup to RDF. When it comes to the requirement of "consistency of unstructured content", two directions have to be considered: Changes in markup have to be propagated to RDF. And changes in RDF have to be propagated to the markup. The latter direction is only partially supported. Namely, when a resource URI is renamed. Changing the manifested references in RDF does not change the markup. Markup-to-RDF is handled during resource post-processing, RDF-to-markup is handled during URI renaming.

Resource post-processing. The references are extracted from the markup as a set of (property key, property value) pairs. First all properties whose keys appear in the set are removed and then the pairs are written to RDF as new properties. Note that the semantics of the references is largely ignored, it is expressed as the property keys. References are described in more detail in Sect. 10.4.1.

URI renaming. The markup contains URIs as the arguments of reference commands. When renaming a URI, one can use the manifestations in RDF of the references to find out what resources with wiki markup *might* be affected. Then one still has to find all occurrences of the URI to be renamed (if any) and update them.

9.5.3 External unstructured data

External unstructured data is everything that is not stored in an RDF repository: files, email messages[3], Java source code fragments, etc. CoIM refers to this kind of data by creating *proxy resources* for it. Proxy resources can be linked to and embedded, as a place holder for the external entity. They can also be used to associate meta-data such as tags with external entities. All proxies for a group of external entities (such as a directory in the file system or an email account) is kept in a single named graph. For each kind of external entity two things need to be defined:

- **Graph type:** An element of ExtKind defines what kind of data is stored in a given named graph.

[3] Mentioned for illustrative purposes only and not currently supported by HYENA.

- **Proxy update:** An operation that brings proxy resources up to date with the (meta-)data of the external entities.

Furthermore, support for external data often includes help with following proxies to the original and vice versa. For example, one can let the user search for an external entity and then go to the corresponding proxy resource; possibly after having automatically created it. Note that while files are included with a project, not all conceivable external data has to be. Proxy resources could, for example, refer to data on the Internet.

Choosing URIs for proxy resources

External entities usually have identifiers. Some of these IDs are *stable*, they don't change during the lifetime of the external entities. An example is the ID of an IMAP email message. Stable IDs are translated to URIs for proxy resources. Each entity with an unstable ID is turned into a resource with a generated, globally unique URI. The ID is stored in a property. If it changes, the property is updated during the next proxy update. The goal is to keep the URIs of the proxy resources stable. Then, during a proxy update, only the proxy resources need to be changed and to the statements that refer to them. For example, URIs for file proxies cannot include the (unstable) file system path, because if the file was to move then the statements mentioning the URI would have to be updated. With unstable IDs, extra care has to be taken not to create two proxies for the same external entity.

9.6 Search calculus

Result sets describe the results of multi-paradigm search (Sect. 7.7). This search integrates keyword-based search, faceted navigation, meta-faceted navigation and more. To formally describe how result sets are produced, this section presents a calculus for multi-paradigm queries. In other words, it contains a set of operations on which multi-paradigm search can be based.

9.6.1 Requirements

In general, the *result set* of a search are all resources that match the query. The query is expressed as a sequence of filters. Basic filtering is performed by enumeration (only keep the enumerated resources; used when the user selects resources and creates a new set with them), by property (only keep resources with a given property), and by graph (only keep resources in given named graphs). For faceted navigation, a summary of a set is needed, in addition to facet-based filtering.

Meta-faceted navigation has the following requirements. Results need to be returned as a set of *occurrences*, denoting not just what resources matched, but also where the match occurred. Various kinds of filtering operations need to be supported that are specific to the meta-facet. For example, one such operation, for the meta-facet "time", is "only occurrences within a given month". Finally, one needs to know the set of all predicates where matches occurred.

9.6.2 The calculus

In search calculus, a query is a sequence of filters. Filtering is performed by starting with all resources in the repository; each filter removes zero or more resources from

its input. A filter is a function and a query is produced by composing functions. The semicolon operator is used for *diagrammatic function composition* (from left to right). The last argument and the range of each filter function is always a set. Sometimes additional arguments are used to parameterize the filter. These arguments are curried (partially applied), resulting in a function from set to set that can be composed. The RDF repository is almost always an implicit argument; it is omitted for reasons of simplicity.

Sets

The following sets are used in function signatures:

- Date, Str: dates and text strings, the types of filter parameters.
- $\mathrm{Occ} = \mathrm{Res} \times \mathrm{Uri} \times \mathrm{Node}$: the set of occurrences. An occurrence is a triple consisting of the subject where something has been found, the property key and the property value (or an abbreviation thereof).

Constants

The following constants are used as filter arguments and to start a query expression.

$\mathsf{today} : \mathrm{Date}$
: returns the date of today.

$\mathsf{facets} : 2^{\mathrm{Uri}}$
: returns all declared facet predicates (which influence the summary).

$\mathsf{userGraphs} : 2^{\mathrm{Uri}}$
: returns the URIs of the *user graphs* which comprises the default graph and any graphs the user has created. Read-only graphs provided by the HYENA framework are non-user graphs.

$\mathsf{all} : 2^{\mathrm{Res}}$
: returns the set of all subjects in the RDF repository. Almost every query starts with this constant.

Basic filtering

The following filter functions complement (meta-)faceted search and keyword search.

$\mathsf{oneOf} : 2^{\mathrm{Res}} \times 2^{\mathrm{Res}} \to 2^{\mathrm{Res}}$
: $\mathsf{oneOf}(\mathit{enum}, \mathit{input})$ only keeps those resources of input that appear in enum.

$\mathsf{withProp} : \mathrm{Uri} \times 2^{\mathrm{Res}} \to 2^{\mathrm{Res}}$
: $\mathsf{withProp}(\mathit{prop}, \mathit{input})$ only keeps those resources of input that have the given property. For RDF search, a useful option is to only show resources that have a type. This hides many internal helper resources and thus reduces clutter.

$\mathsf{inGraph} : 2^{\mathrm{Uri}} \times 2^{\mathrm{Res}} \to 2^{\mathrm{Res}}$
: $\mathsf{inGraph}(\mathit{guris}, \mathit{input})$ only keeps the resources of graphs whose name appears in guris.

Faceted navigation

For faceted navigation, one needs to filter with the current restriction and to compute a summary of the current result.

$\mathsf{restrictFacets} : 2^{2^{\mathrm{Uri}\times\mathrm{Node}}} \times 2^{\mathrm{Res}} \to 2^{\mathrm{Res}}$
: $\mathsf{hasFacet}(disj, input)$ keeps those resources whose properties match $disj$. $disj$ is a disjunction of conjunctions of (key,value) pairs, encoded as a set of sets of pairs. If a resource matches the disjunction, it contains all property (key,value) pairs of at least one of the disjunction elements.

$\mathsf{facetSummary} : 2^{\mathrm{Uri}} \times 2^{\mathrm{Res}} \to (\mathrm{Uri} \times \mathrm{Node})^\star$
: $\mathsf{facetSummary}(keys, input)$ extracts the values of the facets $keys$ from $input$. Returns a sequence of pairs so that value counts can be performed (how often a given value appears in the input).

Meta-faceted navigation

When combining faceted and meta-faceted navigation, one first computes the result of simple filtering and facet restrictions. The resulting resource set is then translated to occurrences which can be further filtered. If afterwards, a summary for faceted navigation is needed, it must be extracted from the final set of occurrences via projSubj. Note that for the month view, it makes sense to show all time-related predicates even those that don't appear in the current month (but in another month). The rationale is that even "no results" means something, especially if browsing several months.

The following operations translate a set of resources to a set of occurrences. This translation is sometimes performed in conjunction with a filtering step.

$\mathsf{withTime} : 2^{\mathrm{Res}} \to 2^{\mathrm{Occ}}$
: $\mathsf{withTime}(ress)$ translates the set of resources $ress$ to a set of occurrences. The result contains (the occurrences of) all time-based property values.

$\mathsf{withText} : \mathrm{Str} \times 2^{\mathrm{Res}} \to 2^{\mathrm{Occ}}$
: $\mathsf{withText}(text, ress)$ returns all occurrences of property values that contain $text$. The values are abbreviated to only show the search text and a few surrounding words.

Projecting occurrence components:

$\mathsf{projSubj} : 2^{\mathrm{Occ}} \to 2^{\mathrm{Res}}$
: $\mathsf{projSubj}(occs)$ returns the subjects of the occurrences in $occs$.

$\mathsf{projPred} : 2^{\mathrm{Occ}} \to 2^{\mathrm{Uri}}$
: $\mathsf{projPred}(occs)$ returns the predicates of the occurrences in $occs$. Used to compute the list of all available predicates.

Filtering occurrences:

$\mathsf{inMonth} : \mathrm{Date} \times 2^{\mathrm{Occ}} \to 2^{\mathrm{Occ}}$
: $\mathsf{inMonth}(date, input)$ only keeps occurrences whose dates have the same month as $date$. The argument is a date (and not a month or a date-time), because a date denotes the "current position in time" in the user interface. This position determines the month to be displayed (month view) and the day to be displayed (day view).

beforeDay : $\mathrm{Date} \times 2^{\mathrm{Occ}} \to 2^{\mathrm{Occ}}$
: beforeDay$(date, input)$ only keeps occurrences with dates before the day of $date$.

onDay : $\mathrm{Date} \times 2^{\mathrm{Occ}} \to 2^{\mathrm{Occ}}$
: onDay$(date, input)$ only keeps occurrences whose dates have the same day as $date$.

afterDay : $\mathrm{Date} \times 2^{\mathrm{Occ}} \to 2^{\mathrm{Occ}}$
: afterDay$(date, input)$ only keeps occurrences with dates after the day of $date$.

withPred : $\mathrm{Uri} \times 2^{\mathrm{Occ}} \to 2^{\mathrm{Occ}}$
: withPred$(pred, input)$ only keeps occurrences whose predicate is $pred$. Used when the occurrence predicate is restricted.

9.6.3 Implementing the calculus

The search calculus is implemented by HYENA by translating a sequence of filter operations to a SPARQL query. SPARQL exhibits the following limitations:

- Specific blank nodes cannot be referenced. As a consequence, one cannot filter arbitrary resources by enumeration. As a work-around, HYENA filters the query result in Java.
- Query results in non-first normal form: One cannot list a resource with all of its properties, but always has to perform separate grouping.
- Statements that are not in any particular graph cannot be listed. Most RDF engines by now store their data as quads so that each triple is annotated with the named graph it is in. Some statements belong to the so-called default graph, they do not have a graph URI. While one can list all statements in a particular named graph in SPARQL, one cannot list all statements in the default graph. As a work-around, HYENA queries for triples that exist somewhere in the repository but not in any particular named graph. Another solution would be to assign a URI to the default graph and change all quads after loading and before saving.

9.7 Example

The RDF in this section is expressed in Trig syntax [C+] which is an extension of the Turtle syntax for triples that has been used so far. Trig allows one to specify named graphs by grouping triples with braces and prefixing a URI. In addition to standard namespaces and the HYENA namespaces `hyena` and `wikked`, two additional namespaces are used: `graph` for the URIs of named graphs and `res` for the URIs of resources. URIs in the latter namespace are usually automatically generated. To make the RDF data easier to read, descriptive local names were chosen.

In this example, Shenzi wants to create a wiki page with a single to-do item: To read the paper "As we may think" from Vannevar Bush. She has downloaded the paper as a PDF file and would like to manage both the file and the corresponding bibliographical data with HYENA. Initially, the project is a file directory that looks as follows:

```
shenzi/
    shenzi.trix
    think.pdf
```

`shenzi.trix` is the RDF repository, `think.pdf` is a PDF of "As we may think". The initial data in the repository is

```
graph:Framework {
    res:Lens rdf:type fresnel:Lens .
}
graph:Files {
    res:ThinkFile rdf:type hyena:File ;
        hyena:path "think.pdf" .
}
```

The bibliographical data of that paper is, in BibTeX format:

```
@article{bush:1945,
    title   = {As we may think},
    author  = {Vannevar Bush},
    year    = 1945,
}
```

This data is added to the repository as the following RDF.

```
graph:UserData {
    res:Think rdf:type bibo:Article ;
        dc:title "As we may think" ;
        dc:date "1945-07-01"^^xsd:date ;
        bibo:authorList ( res:Bush ) .
    res:Bush rdf:type foaf:Person ;
        foaf:name "Vannevar Bush" .
}
```

Next, she creates two wiki pages. The first page embeds the data of paper. To do so, the user only specifies the URI, HYENA automatically adds the title "As we may think" of `res:Think` to the `\embed` command, so that it becomes more readable for humans. Underneath the embedding, Shenzi adds a short comment.

As we may think (res:PageThink)

```
\embed{res:Think As we may think}
Still contains many relevant ideas.
```

The second page links to the first page. Shenzi adds the tag "Todo" to it.

Todos (res:PageTodos)

```
* Read \link{res:PageThink As we may think}
```

The two pages lead to the following RDF being added to the repository. Note how `\embed` and `\link` in the wiki markup lead to RDF properties being created. A resource for the "Todo" tag has also been automatically created.

```
graph:UserData {
    res:PageThink rdf:type wikked:Page ;
        rdfs:label "As we may think" ;
        wikked:content "..." ;
        dcterms:created "2009-11-07T09:54:33"^^xsd:datetime ;
        dcterms:modified "2009-11-07T09:54:33"^^xsd:datetime ;
        wikked:embed res:Think .
    res:PageTodos rdf:type wikked:Page ;
        rdfs:label "Todos" ;
```

```
            wikked:content "..." ;
            dcterms:created "2009-12-06T14:03:10"^^xsd:datetime ;
            dcterms:modified "2009-12-06T14:03:10"^^xsd:datetime ;
            tagging:tag res:Todo ;
            wikked:link res:PageThink .
        res:Todo rdf:type tagging:Tag ;
            rdfs:label "Todo" .
    }
```

9.7.1 Data calculus expressions

In this section, the namespace prefixes of the qnames are omitted (for example, `modified` is used instead of `dcterms:modified`).

Query: What resources are matching the facet disjunction $((\texttt{type} = \texttt{Page}) \wedge (\texttt{tag} = \texttt{Todo})) \vee (\texttt{type} = \texttt{Tag})$?

$$\mathsf{all}; \mathsf{restrictFacets}(\{\{(\texttt{type}, \texttt{Page}), (\texttt{tag}, \texttt{Todo})\}, \{(\texttt{type}, \texttt{Tag})\}\})$$

Result (resources): $\{\texttt{PageTodos}, \texttt{Todo}\}$

Query: Meta-facet summary: what time-related predicates appear in the repository?

$$\mathsf{all}; \mathsf{withTime}; \mathsf{projPred}$$

Result (predicates): $\{\texttt{created}, \texttt{modified}\}$

Query: When have resources been modified in November 2009? A complete date is used to specify the month.

$$\mathsf{all}; \mathsf{withTime}; \mathsf{inMonth}(2009\text{-}11\text{-}26); \mathsf{withPred}(\texttt{modified})$$

Result (occurrences): { (`res:PageThink`, `modified`, `"2009-11-07T09:54:33"^^datetime`) }

Query: Where does the text "Todo" appear in the repository?

$$\mathsf{all}; \mathsf{withText}(\text{“Todo”})$$

Result (occurrences): { (`PageTodos`, `label`,`"Todos"`), (`Todo`, `label`,`"Todo"`) }

Query: List the facets in the user graphs. Returns a sequence to be counted.

$$\mathsf{all}; \mathsf{inGraph}(\mathsf{userGraphs}); \mathsf{facetSummary}(\{\texttt{type}, \texttt{tag}\})$$

Result: ⟨ (`type`, `Article`), (`type`, `Person`), (`type`, `Page`), (`type`, `Page`), (`tag`, `Todo`), (`type`, `Tag`) ⟩
Thus, facet `type` has the values `Article`, `Person`, `Tag` that appear once and the value `Page` that appears twice. Facet `tag` has the value `Todo` that appears once.

9.8 Discussion

The CoIM model extends RDF, so that it is suitable for connected information management: One has to be able to manage both structured and unstructured data. This is achieved by manifesting every entry that is to be managed as an RDF resource. Some unstructured data, such as files, are not stored in the RDF repository, but still managed by CoIM. Wiki markup is unstructured data that is stored in the RDF repository, but can contain references to structured data. It is thus useful for annotating and combining structured and unstructured data. Further book-keeping is done when the user saves a resource to RDF, renames a URI or invokes an update of derived data, fulfilling the requirements mentioned at the beginning of this chapter:

- Uniform annotation: Every entity is (represented as) an RDF resource and uses RDF properties for annotations.
- Uniform semantic references: RDF properties are also used as labeled edges between entities.
- Consistency of unstructured content: The RDF representation of unstructured content is updated either when saving it (wiki pages) or during derivation update (files). URI renaming changes RDF and updates wiki pages as necessary.
- Multi-paradigm search: The search calculus presented in this chapter lays the foundations for a user interface.
- Uniform navigation: Navigation is mainly based on annotations such as tags and facets, which, thanks to the uniformity requirement, are attached to all entities in the same manner. Thus, the search calculus does not distinguish between different kinds of entities.

Looking at related work, only semantic wikis undertake the work of integrating unstructured data and structured data. The semantic wiki with the most powerful modeling abilities is KiWi [SEG+09]. It is similar to CoIM, but does not have CoIM's integration of external data or a formalization of multi-paradigm search. KiWi's model is based on *content items*, a mixture of XML (wiki) and RDF (data). Content items are more powerful than CoIM wiki markup (queries can refer to the structure of XML). But where KiWi introduces a completely new infrastructure, CoIM minimizes this kind of effort and is based on a standard RDF repository.

The paper "Extending Faceted Navigation for RDF Data" [ODD06] has a calculus that is similar to the CoIM search calculus, but only for faceted navigation (albeit a more powerful variant than HYENA's).

Future research will be concerned with making more internals of unstructured data available to CoIM. For example, a derivation update could extract text from unstructured entities, enabling a full text search via the data calculus.

Chapter 10

Wikked: A wiki markup language

Contents

10.1 Overview

Wiki markup is an important element of the CoIM integration strategies, because it can hold both unstructured text and (references to) structured data. This makes it useful for anything from small notes to presenting and annotating data. The breadth of connected information management places several requirements on wiki markup: In addition to wiki syntax for simple content, more complex content should be supported by a formal markup language. Plugins supporting new vocabularies have to be able to extend the markup language. Several display media have to be supported (among others, a graphical user interface and printing). For collaboration, editing conflicts have to be handled and an editing history is needed.

These requirements lead to wiki markup being evaluated in stages: An initial hybrid of wiki syntax and LaTeX syntax is translated to pure LaTeX, parsed and evaluated to a core abstract syntax. This abstract syntax is then *rendered*, turned into an external markup language such as HTML or LaTeX. Plugins can contribute new commands that are converted to the core syntax during evaluation. The CoIM file management is used to keep a history of editing. Each history entry is addressed using a globally unique version ID that is assigned to a page when it is saved. This ID is also used to detect editing conflicts which are resolved either manually or automatically, via a simple diff algorithm [HM76].

10.2 Requirements

Wiki markup plays the important role of being a glue for structured data, by annotating and collating it. Simple things should be easy to enter, in traditional wiki syntax. But one should also be able to express complex things, for which a more formal markup language is better suited. This markup language should be extensible so that framework plugins can add new constructs. The final "rendering" of the markup has to be configurable: Being able to export LaTeX is useful to make the conversion to printable documents, HTML is necessary for display in a browser. HTML output itself has two variants: On one hand, HTML should contain user interface elements such as links to go to embedded content. On the other hand, simpler HTML is needed for printable output. Other parts of the framework also have the need for configurable output, not just wiki markup evaluation. That means that an abstract wiki syntax should be available to framework plugins. An example are embedders, which use the abstract wiki syntax to render their content.

For collaborative editing, two more features are important: First, one should be able to visit older versions of a wiki page, to understand its history and so that inappropriate changes can be undone. Second, editing conflicts have to be handled: If a wiki page is to be saved and its content in RDF has changed since editing began, there has to be a way to reconcile the changes in RDF and the changes in the editor.

10.3 The markup language and its processing

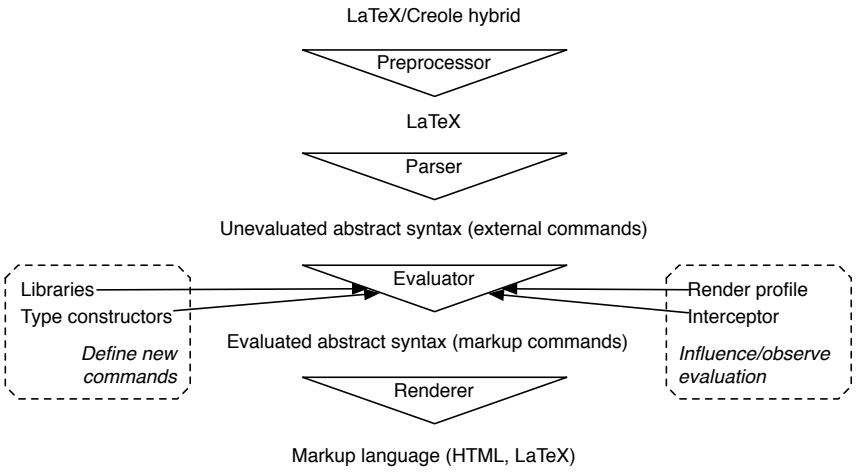


Figure 10.1: Content processing comprises the following phases: preprocessing, parsing, evaluation, and rendering.

The CoIM wiki markup language is called *Wikked*. In order to make Wikked as flexible as required, it goes through a number of transformation steps (Fig. 10.1) where each step transforms one formal language to another:

- The initial input is a mixture of LaTeX syntax (for example, `\textbf{bold}` for bold text) and Creole wiki syntax (for example, `**bold**` for bold text). This is the markup language that the end user sees and uses.

- A *preprocessor* transforms this hybrid into pure LaTeX, mainly by searching and replacing via regular expressions. For example, all occurrences of `**bold**` become `\textbf{bold}`.
- *Parsing* leads to an internal *unevaluated abstract syntax*. Various *external* commands, some of them quite powerful, still need to be evaluated. For example, `\embed{res}` is a command to *embed* resource `res` (to insert the content of it).
- *Evaluation* leads to an abstract syntax with a core set of so-called *markup commands*. These comprise the most important construct for displaying text such as tables, bullet lists, text styles, etc. Thus, all external commands are evaluated to markup commands. For example, `embed` is replaced with the (evaluated) commands of the resource it refers to.
- *Rendering* is the final step where Wikked's syntax is translated to an external markup language such as HTML or LaTeX.

The following sections explain all transformation steps and the formal languages that are involved.

10.3.1 Input: Creole/LaTeX hybrid syntax

```
| this | is |
| a | table |
//italics// and **bold**
* A list with
* two bullets
* and a \href{http://nytimes.com}{link}
```

---

this	is
a	table

italics and **bold**

- A list with
- two bullets
- and a link

Figure 10.2: Wiki markup: On top, you see how the text is entered, on the bottom, how it is rendered. Note how it is a mixture of traditional wiki markup and a LaTeX command (last line, starting with a backslash).

Wikked, the syntax entered by the user, is a hybrid of the Creole wiki syntax and LaTeX (Fig. 10.2). Creole [wika] is a standard for wiki syntax and uses visual symbols such as `**` for bold text, `//` for italic text, `|` for tables, etc. LaTeX [Pro] has a more formal syntax where commands are written as

```
\command[option1=value2, o2=v2]{argument1}{a2}
```

Any argument or value can contain line breaks and commands can be arbitrarily nested. Alternatively, one can bracket several lines with `\begin{env}` before the first line

and `\end{env}` after the last line. `env` is the environment, a name for a multi-line command, if you will. Thus, where the dominant construct in Creole is the text line (the character that starts a text line often determines how it is interpreted, as, for example, with bullet lists), it is the command or the environment in LaTeX. This makes the two syntaxes hard to mix in a formal grammar and necessitates the following processing steps. A detailed description of Wikked markup is given in App. A.

10.3.2 Preprocessing: translate to pure LaTeX

Preprocessing translates the end user syntax to pure LaTeX. This is done by iterating over the lines and doing a regular expression based search and replace from Creole characters to LaTeX commands. One such rule is:

`[^:]//(.+?)//` → `\textit{$1}`

Explanation: The left hand side of the rule is a *condition*, a regular expression that specifies when the rule is applicable. If it matches, the right hand side of the rule, the *replacement*, takes the place of the matched text. If there is a URL such as `http://example.com` in the text, the double slash it contains should not be treated as starting text in italics. Thus, the condition forbids colons before the initial double slash. The double slash is followed by a sequence of one or more arbitrary characters. This sequence is minimally (lazily) matched and finished by a second double slash. The replacement wraps the text between the slashes in a LaTeX `\textit` command. Multi-line Creole markup such as bullet lists are handled by keeping the current line prefix as context during parsing. If it changes, one might have to open or close LaTeX environments. Triple braces are used in both Wikked syntaxes to escape markup; special characters inside them are interpreted as plain text and do not change the text style. This means that an extra parsing step is needed before applying regular expressions: The text is translated into a stream of parsed and unparsed (escaped) segments. Only the parsed segments are changed by regular expressions. The last step of preprocessing is to combine the segments into a single text string again.

10.3.3 Parsing: convert to unevaluated abstract syntax

Expression	::=	Block | Command | NoWikiText | PlainText .
Block	::=	Expression* .
Command	::=	"\" Name ("[" Options "]")? PositionalArgument* .
Options	::=	Key "=" Expression ("," Options)? .
PositionalArgument	::=	"{" Expression "}" .
NoWikiText	::=	"{{{" Character+ "}}}" .
PlainText	::=	Character+ .

Figure 10.3: Grammar of concrete LaTeX syntax.

During parsing, text in LaTeX syntax is translated to the abstract Wikked syntax. The concrete (LaTeX) syntax is defined by the grammar in Fig. 10.3.

In the abstract syntax (Fig. 10.4), Expression, Block and Command are direct equivalents of the concrete syntactic constructs. Evaluation is often directly delegated to methods defined in the host language Java. To make implementing these methods simpler, instances of class JavaObject are part of the abstract syntax and wrap native

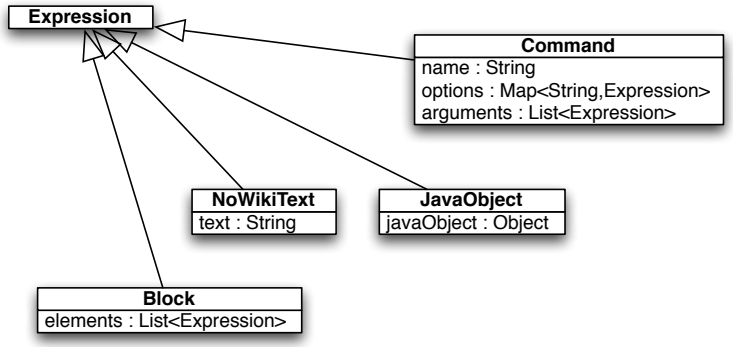


Figure 10.4: Wikked abstract syntax

Java objects. JavaObjects can directly hold complex intermediate results without some kind of encoding scheme. There are two kinds of text. The concrete PlainText is stored as a String inside a JavaObject. The concrete NoWikiText becomes the abstract NoWikiText. Note that for functions working with text, the two kinds of storing it make no difference, as they are mainly different ways of *parsing* text; special characters in plain text have to be escaped, while nowiki text is a single escaped block. A common text API abstracts the differences. This API also involves Block, because adjacent plain text and nowiki text might have to be joined into a single text string. But Creole also defines NoWikiText to change the appearance of text: text does not wrap, line breaks show up as line breaks, and a mono-spaced font is used. Thus, while one could wrap plain text with a special nowiki command, using an extra syntactic construct helps with making printed syntax trees look the same as the parsed input. Furthermore, two kinds of nowiki text are distinguished: inline nowiki text (no newlines) and multi-line nowiki text (with newlines). They are usually rendered differently[1].

10.3.4 Evaluation: convert to evaluated abstract syntax

In evaluation, a broad set of commands is translated to a core set of *markup commands*. The latter is a subset of the former. Both source and target of this translation are in abstract syntax. The following rules define evaluation:

$$
\begin{array}{lcl}
\text{eval(t : NoWikiText)} & = & \text{t} \\
\text{eval(c : Command)} & = & \text{evalFunction(c.name,} \\
 & & \quad \{\ \text{key} \rightarrow \text{eval(expr)} \mid (\text{key} \rightarrow \text{expr}) \in \text{c.options}\ \}, \\
 & & \quad [\ \text{eval(expr)} \mid \text{expr} \in \text{c.arguments}\ ]\ ) \\
\text{eval(b : Block)} & = & \text{block([eval(expr)} \mid \text{expr} \in \text{b.elements])} \\
\text{eval(j : JavaObject)} & = & \text{j}
\end{array}
$$

Evaluating a command invokes `evalFunction`, after evaluating the map of options and the sequence of arguments. Evaluating a block means creating a new block with evaluated elements. `evalFunction` translates markup commands to the same command, but with evaluated options and arguments. For other commands, it looks for a definition and signals an error if there isn't one. The next section describes ways of defining new commands.

[1] LaTeX: `verb` command versus `verbatim` environment.

Evaluation is controlled by the so-called *evaluator*. It holds various kinds of data that influences evaluation. Because the constructs to be evaluated are often nested (e.g. when embedding content), the existence of the data it contains can be scoped: There is not a single evaluator, but a chain of evaluators where the last (top-most) is current. One "pushes" a new evaluator onto the evaluator stack when, for example, a new embedded page is to be evaluated. To enable commands to store any kind of scoped data, each evaluator holds a mapping from a key to an expression. Examples of data stored in the evaluator:

- Shared: *local settings* (with user interface settings such as the current write graph); *heading counter* (keeps track of nested numbered headings); *render profile* (parameterizes output, see below); *evaluation interceptor* (tracks evaluation, see below). This data is not scoped and thus the same in every member of the evaluator chain.
- Scoped: *heading level depth* (How deeply nested is the current heading? Needed, because if a page is deeply nested, then so are its top-level headings); *TOC in progress* (in bindings, a boolean that indicates that the current evaluation of the data is for the table of contents, see Sect. 10.3.4); the *current resource* (in bindings, needed mainly by wiki pages, so that the RDF data can be accessed via commands).

Note that management of headings keeps two separate pieces of data: the heading counter is compound and keeps one number per level of *numbered* nesting. To make sure that unnumbered headings also have an effect on nested headings, the heading level depth is used.

Defining new commands

There are two ways of defining new commands: Libraries and type constructors. Libraries are all objects in a core container that implement a marker interface. Each method that is annotated with `@Function` defines a Wikked command. `evalFunction` uses name and arity to find a suitable match.

Type constructors define a bidirectional translation between a Java object and a Wikked expression. They are used so that complex Java objects such as lists can be introduced into evaluation and to print an expression with a JavaObject as something parsable. Type constructors are subclasses of `AbstractTypeConstructor` and define

- the type that is constructed.
- name and arity of a Wikked command and a method implementing that command that returns an instance of the constructed type.
- a method that translates an instance of the constructed type to a Wikked expression.

Wikked currently has constructors for lists, booleans, integers, URIs, and literals. Due to type constructors, Wikked can be used for serializing simple Java objects; in a syntax that is (moderately) human-readable.

Influence and observe evaluation

There are two ways to influence and observe evaluation.

Render profiles. RenderProfiles are necessary, because the rendered HTML in HYENA contains so-called *chrome*, user face elements that go beyond simple hyperlinks: going to an embedded resource to display or edit it, creating a new resource, etc. Chrome depends on the platform: it is different on HYENA/Eclipse and HYENA/Web and does not show up at all when printing. This requirement necessitated to a way of parameterizing evaluation, the so-called *render profiles*. Render profiles get their name from being tightly coupled with rendering. HYENA has one render profile for printing and two render profiles (Eclipse and Web) for chrome. The print profile can be used for LaTeX, too, but the chrome profiles only work with HTML rendering.

- Platform characteristics: Is CSS to be included (needed by HTML renderer)? Is chrome to be displayed (used when evaluating an embedded page and GUI invocations)?
- Table of contents: post-processing a numbered heading, appending a TOC entry. Both methods help with platform-specific ways of scrolling to a location in a document (in HYENA/Eclipse, one can use normal anchors, in HYENA/Web, URL fragments are not available).

Interceptors. An interceptor is attached to an evaluator and notified by it of certain events during evaluation. Currently, interceptors are only used for assembling the table of contents: Before evaluating a possibly compound page, one has to create the table of contents. To do so, the page is already completely evaluated, but one is not interested in the result, but in finding all headings. Accordingly, the interceptor for the table of contents is notified of three events: the beginning and end of a page (so that dotted frames can be drawn around embedded content in the TOC) and the appearance of a numbered heading. Triggered by these notifications, it constructs the markup for the table of contents.

User interface scope versus Wikked scope

HYENA is always split into a back-end (runs on the server in HYENA/Web) and a front-end (the user interface, runs on the browser in HYENA/Web). The back-end manages the RDF and thus Wikked processing is performed in the back-end. If one of the Wikked commands needs to contact the user interface (for example to select a resource when a link has been clicked) then that is handled in a platform-specific way via rendering profiles.

10.3.5 Rendering: convert to output markup language

The final step before displaying the evaluation result is to *render it*, to translate abstract wiki syntax with only markup commands to an output markup language. Currently, HYENA supports HTML and LaTeX. This step is relatively straightforward, because the markup commands contain very simple constructs such as headings, text styles, bullet lists and tables. And these constructs are easily translated to a markup language.

10.4 Structure and wiki markup

By adding structure to a wiki text, it becomes semantically richer. There are two common ways of introducing structure: references and page fragments.

10.4.1 References

Traditional wikis have a single kind of reference: *links* refer to pages. Wikked goes beyond these capabilities by providing three kinds of references that can refer to both data and wiki pages (as there is no fundamental difference between the two in the CoIM model). References point to resources via their URI. By manifesting references in RDF, they become structured data and can be queried for.

- `\link` is rendered as a hyperlink to another resource. It is manifested in RDF as a `wikked:link` property.
- `\embed` displays the complete content of another resource inside the current resource. How the content is displayed can be configured, but the default rarely needs to be changed. Embeddings are manifested in RDF as `wikked:embed` properties.
- `\annotate` is rendered as an icon of a post-it note. Clicking on the icon displays a pop-up with the content of the annotation resource. Annotations manifested in RDF as `wikked:annotate` properties.

10.4.2 Page fragments

The main use cases for page fragments are:

- Annotation: Some content management systems allow content fragments to be annotated with comments, corrections, etc. Similarly, some specifications are published with a unique number for every paragraph, to enable annotations.
- Editing page fragments: MediaWiki (which powers Wikipedia) allows one to "zoom in" on part of a page and edit it. This reduces editing conflicts and makes editing more focused.
- Refined querying: If the querying mechanism is aware of the text structure, one can perform queries such as "in what paragraph appears both the word 'wiki' and the word 'semantic'"?

Some wikis store their content in XML and permit queries that take the XML structure into consideration. In contrast, Wikked markup currently is treated as flat text when it comes to querying (only references are exported to RDF). A limited kind of annotation can be performed via annotation references. Otherwise, a page fragment can be emulated by creating a sub-page for it and embedding it in the parent page.

10.5 History and editing conflict management

Past versions of a wiki page are stored in the file system. There is one directory per resource and one text file per history entry. Each history entry has a globally unique ID (automatically generated), a time stamp and a list of entries it is derived from (usually one; more than one when merging entries). Per page, there is a single history root (the very first version) and a single *current version* which is stored in the wiki page resource. The history thus forms a directed acyclic graph (DAG). Three aspects combine to make distributed synchronization simple: First, the DAG handles editing branches well. Second, data is stored using standard CoIM file management which means that

file synchronization can be used for the history. Third, the data grows monotonically; existing data is never changed.

10.5.1 Editing conflicts during saving

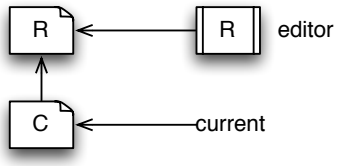


Figure 10.5: Editing conflict during saving. The version R recorded in the editor is not current any more. Instead, a newer version C has been saved since the editor was opened.

An editing conflict can happen during saving (Fig. 10.5): The version R recorded when reading the page is not the same as the current version C of the page in the repository (meaning that it has been changed elsewhere since). HYENA presents the user with the following choices:

- Automatic merge: Perform an automatic merge of C and the current content, using a simple diff algorithm [HM76]. The merged page gets a new version and is marked as being derived from both R and C.
- Manual merge: Perform an automatic merge, but don't save it yet. Instead the user can continue editing to fine-tune the result.
- Overwrite and save: The current content is forced to be the current version and marked as being derived from both R and C.
- Continue editing: postpone the decision about how to solve the conflict.
- Don't save: completely discard the changes that have been made.

10.5.2 Editing conflicts during synchronization

The second kind of editing conflict happens during synchronization, if a common older version `v0` was continued differently on two installations (Fig. 10.6). The different history entries `v1` and `v2` are both considered current. Thus, the user has to decide which of the two resources to keep. She can also choose to merge the two resources. Then a merged history entry is created whose fathers are `v1` and `v2`. In either case, the effect is that there is again a single current entry in the history.

10.6 Future research

The mechanism used by the back-end to contact the user interface should be made explicit so that end users can use it, too. This would allow them to automate tasks, program custom functionality etc. Then three different formal languages are used in

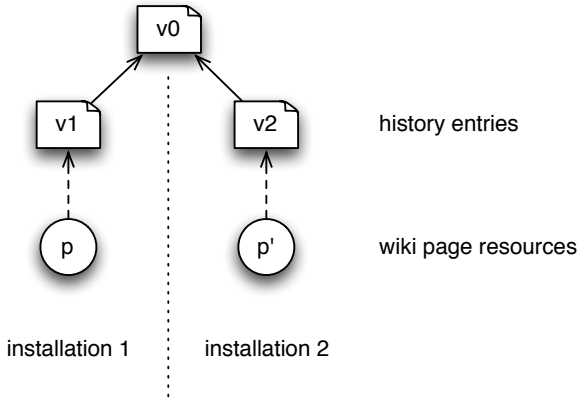


Figure 10.6: Editing conflict during synchronization. Two installations have continued the history differently. The same resource also exists in different versions p and p'.

Wikked: Wikked for wiki markup, Java to extend wiki markup, and some other language (probably JavaScript) to control the user interface.

Having an editing history is very useful. Thus, HYENA should also have one for RDF data, not only for the content of wiki pages. Most of the effort of devising such a mechanism will be spent on making it as flexible as the current scheme, while handling any RDF.

10.7 Discussion

The requirements stated at the beginning of this chapter are fulfilled as follows. The complexity of the markup language scales well: Simple things such as bold text or bullet lists can be quickly entered using the Creole wiki syntax. Complex, possibly nested, commands can be entered using LaTeX syntax. Additionally, the latter syntax can be extended by plugins, so that support for new kinds of data can also be based on Wikked. Staged parsing ensures that this flexible concrete syntax is translated to a compact abstract syntax. This simplifies the targetting of different output media such as a web browser or a printer, because this abstract syntax can be easily translated to external markup languages such as HTML or LaTeX. Any client or part of the framework can access this capability by using the core syntax to produce its output. With branches and non-conflicting merging of version DAGs, the wiki page history is very flexible in distributed settings. Editing conflicts are handled in a manner that is typical for this kind of problem. Because the history uses the standard CoIM file management, it is easy for users to fix problems. History management is reminiscent of distributed version control [YCM06].

No other wiki (semantic or otherwise) has a markup language that is as sophisticated as Wikked. Compared to other wikis, moving from simple to complex markup is easier for end users and automatic processing of markup is helped by a concise abstract syntax. A predecessor of Wikked has been discussed in [RK06].

In the running example, researcher Shenzi uses the LaTeX interoperability of Wikked to support her external LaTeX-based work flow that she has established for writing papers. During brain storming with her coauthors, bullet lists in Creole syntax are created

as wiki pages. When writing the paper, those are exported to LaTeX. When a part of the paper needs to be collaboratively edited, the relevant fragment is put on a wiki page and later reintegrated into the paper.

Chapter 11

Templates: A presentation language for RDF

Contents

11.1 Overview

Lenses (Chap. 13) are a convenient way of defining a way to both display and edit a resource. Alas, when it comes to displaying, they are relatively complicated, but not very flexible. Their output is always a table with one property per row. *Templates* were created to provide more flexibility for displaying than lenses while being simpler to use at the same time. A template is a function from RDF to Wikked markup. Using a standard technique, a template is expressed in the object syntax Wikked, interspersed with commands in a meta-syntax for hiding or repeating blocks of Wikked or for inserting input data. The meta-syntax chosen for templates is also Wikked which allows one to re-use Wikked parsing. In order to display properties and lists, the commands of the meta-syntax must be able to handle multiple values. Each value can be compound, resulting in the need for recursion. Furthermore, illegal values and the *skeleton* (text between values) must be handled properly. The result are three groups of meta-commands: Display commands present the current node, iteration commands loop over multiple values and recurse into them, conditional commands are used for correctness checks and the skeleton (where they check the position of the current value within a sequence).

11.2 Requirements for RDF templating

The classic way of transforming a record-like data structure to text is via string interpolation. For example, in Unix shells, one can access environment variables by prefixing a dollar sign:

```
echo "Home directory: $HOME"
```

For RDF, CoIM uses a similar approach, but has to take into consideration the peculiarities of RDF. A node can be compound: lists and properties can have multiple values or no value at all. Additionally, one has to choose an order for property values, which are unordered. Displaying them in random order is confusing for end users, choosing a meaningful fixed order is complicated to do generically. The difference between no values, one value, and several values is especially obvious when it comes to the *skeleton*, the text surrounding the values. For example, mathematical sets are written as "$\emptyset$" for the empty set, as "$\{e_1\}$" for a single element e_1, as "$\{e_1, e_2\}$" for two elements, etc. The skeleton consists of $\emptyset$, the braces and the colon. Enumerated items are omitted if there is no item, written as "i_1." for a single item i_1, as "i_1; i_2." for two items etc. The skeleton consists of the semicolon and the dot. If a list or property value is compound one has to recurse into it and display its parts. One might also need to skip a compound value if it is ill-formed. For example, one might not display the resource of a person if no name has been assigned. Extra care has to be taken that the skeleton is properly handled in case of a rejection.

11.3 Syntax and meta-syntax

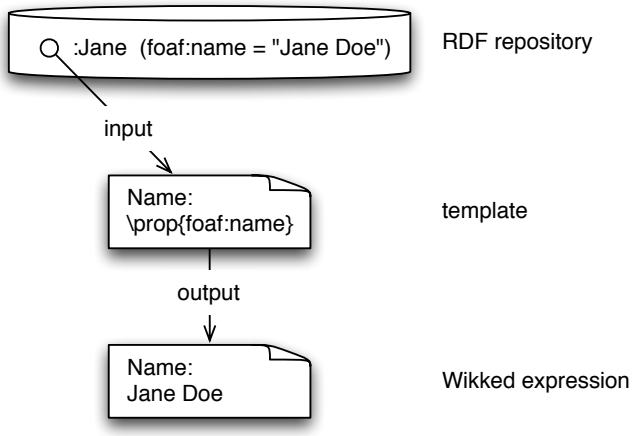


Figure 11.1: A template is a function that translates a resource in the RDF repository to a Wikked expression.

A template is a function that is applied to a resource and produces Wikked markup (which can be integrated with the standard CoIM display work flow). Thus, the object syntax of a template is Wikked. A meta-syntax is used to define the template-as-a-function. It specifies blanks inside the markup that are to be filled in by RDF data and

blocks of markup that might be hidden or repeated. This meta-syntax is also Wikked (Fig. 11.1).

An important concept is the *current node*. When starting template evaluation, the current node is the resource it is applied to. If a template command recurses into a property value, that value becomes current. Thus, both resources and literals can become current. The following island grammar sketches the template syntax:

```
template   ::=  block .
block      ::=  ( objectSyntax | display | iteration | conditional )* .
iteration  ::=  \iterCmd{ source }{ block } .
```

The template is a block which is a sequence whose elements are either target syntax, commands displaying the current node, an iteration (a loop, if you will) over values, or a conditional. An iteration command optionally specifies what to iterate over, the property whose values are to be used. This property is implicit (and the argument omitted) when iterating over a list or a sequence. Display, iteration, and conditionals are explained in the following sections.

11.3.1 Displaying the current node

Formatting commands display the current node in different ways. They are derived from the lens display hints (Sect. 15.6.3 provides more details).

- `\DisplayDateTime[format]` (time-related data)
- `\DisplayExternalLink[text]` (link to a web site)
- `\DisplayImage`
- `\DisplayPlainText[mode]`
- `\DisplayResourceLink[mode,text]` (link to a resource in the RDF repository)
- `\DisplayWikiMarkup` (treat plain text as wiki markup)

The optional parameter `format` provides a date format, a time format or a date time format such as "yyyy-MM-dd HH:mm:ss". The optional parameter `mode` specifies how to display the current node. Possible values are `ResourceLabel`, `ResourceQName`, `ResourceUri`, `LiteralText`, `LiteralParsable` (Sect. 15.6.3). The optional parameter `text` overrides the default link text.

11.3.2 Iteration commands

Iteration commands loop over a sequence of values. If they provide a body then the values becomes the current node in that body. If not, resource values are displayed as links and literals are displayed as plain text.

- `\prop{propUri}{body}` is for iterating over the values of property `propUri`.
- `\seq{body}` is for iterating over the elements of an RDF container (type `rdf:Seq`).
- `\list{body}` is for iterating over the elements of an RDF collection (type `rdf:List`).

The following options can be used for these commands:

- `ifempty`: an expression to be shown if one iterates over the empty sequence.
- `before`: show this expression before all of the result if there is at least one iteration value.
- `after`: show this expression after all of the result if there is at least one iteration value.
- `ifCard` is a sequence of semicolon-separated expressions such as `rdfs:label = 1`, meaning that iteration values are ignored that don't have properties with the given cardinalities. Comparison operators are =, !=, <, <=, >, >=. This option (which also has a corresponding command) is necessary so that the conditionals `\ifLast` and `\ifNotLast` work.
- `range`: an interval such as `1`, `2-`, or `-3` specifying the indices of the values to iterate over.

11.3.3 Conditionals

The body of conditionals is only shown if their condition holds.

- `\ifCard{condition}{body}`: `body` is only shown if the current resource has the property cardinalities specified in `condition` (in the same way as for the iteration option `ifCard` above).
- `\ifFirst{body}`: `body` is only shown if the current iteration element is the first one.
- `\ifLast{body}`: `body` is only shown if the current iteration element is the last one.
- `\ifNotLast{body}`: `body` is only shown if the current iteration element is not the last one.
- `\ifRegex{regex}{then}{else}`: compares a regular expression `regex` with the text of the current resource (that is, the URI or the literal value). If it matches, a `then` text is displayed which can contain the command `\group{i}` to access the value of match group i. An optional `else` text is shown if the regular expression does not match.

11.4 Example

Given the following RDF of a workshop with participants:

```
:wsh2001 rdf:type :Workshop ;
    ex:participant [
        foaf:name "Jane Bond" ;
        foaf:homepage <http://www.sis.gov.uk/~jb/>
    ] ;
    ex:participant [
        foaf:name "Lars Croft" ;
        foaf:homepage <http://www.croft.co.uk/~lc/>
    ] .
```

We can apply the following template to `:wsh2001`. The meta-syntax is printed in bold.

```
Participants of the workshop are:
\prop[ifCard=foaf:homepage=1;foaf:name=1,
      ifEmpty=None]{ex:participant}{
    \href{\prop{foaf:homepage}}
         {\prop{foaf:name}}\ifNotLast{,}\ifLast{.}
}
```

The result is a bullet list of the participants:

```
Participants of the workshop are:
\href{http://www.sis.gov.uk/~jb/}{Jane Bond},
\href{http://www.croft.co.uk/~lc/}{Lars Croft}.
```

11.5 Discussion and future research

Templates provide a simple, mostly declarative, means for displaying RDF. They are an application of proven techniques from, among many others, web application frameworks and generative programming, to the new realm of RDF. The object syntax and result of applying a template to an RDF node is a Wikked expression. Adaptation to the RDF input is handled by three groups of commands: Display commands present the current RDF node as a link, an image, plain text, etc. Iteration commands are used for recursing into complex values and for displaying multiple values coming from properties, containers, and collections. Conditional commands handle illegal or varying input.

Currently, the meta-syntax is Wikked, the same as the object syntax. On the positive side, this allows one to use Wikked parsing and evaluation for handling templates. On the negative side, expressions in object syntax must always be well-formed and closed. One cannot, for example first repeat `\textbf{` several times and then `}` several times. As the core ideas of templates are universal, that suggests using them to generate a wider variety of artifacts. This might mean a different meta-syntax, but certainly a more tolerant parser that ignores the structure of the object language. Naturally, Wikked can still be an object language.

An unsolved problem is in what order to display multiple values of a property. HYENA currently sorts the URIs to have at least a stable order, but a more sophisticated mechanism clearly is necessary. For example, if a property's values are resources, one could provide an argument to `\prop` that states what property of the resources to use as a sort key.

Chapter 12

RDF patterns

Contents

12.1 Overview

This chapter describes useful modeling patterns for RDF. The first group of patterns concerns encapsulating sets and sequences of resources as resources. Part of the SKOS [MB] vocabulary has been defined for this purpose and is explored: Sets are supported by a type for set resources and a property for set members. Problems of RDF collections regarding editing and querying are solved by introducing anchor resources for collections and inferencing for membership. The second group of patterns covers ways for going beyond the labeled binary edges between nodes that RDF offers. By turning the edge into an intermediary resource, it can be annotated with its own properties and/or refer to several targets. The third group of patterns demonstrates how RDF can provide extension points for data contributed by plugins or end users. The scenarios given are the contribution being a single simple value or several complex values.

12.2 Encapsulating multiple resources as resources

12.2.1 Encapsulating a set as a resource

Problem. RDF provides collections and containers for encapsulating sequences of resources, but has no standard construct to encapsulate a set of resources as a resource.

Example. Participants in an event can be either individuals or groups of people. In the latter case, they should be seated close to each other. As an example, let us assume that Fiona is having a party and invites Joe Public and the couple Jane and John Doe. The couple should be added as a single participant.

Solution. The SKOS [MB] vocabulary defines the necessary constructs: a class `skos:Collection` for sets of resources and a property `skos:member` for set members. Storing sets in properties is an obvious approach, with properties being set-valued. The contribution of the SKOS vocabulary is thus having a standard property for doing so and a type that marks this practice. Using the SKOS vocabulary results in the following RDF:

```
:FionasParty rdf:type ex:Event ;
    ex:participant :JoePublic ;
    ex:participant [
        rdf:type skos:Collection ;
        skos:member :JaneDoe ;
        skos:member :JohnDoe
    ] .
```

12.2.2 RDF collections

Stable identity

Problem. Instances of `rdf:List` sometimes have no identity of their own: If they are empty, they are always the URI `rdf:nil`. In this case, no individual label can be given or instance-specific properties added. Furthermore, this fact prevents list identities from being stable during editing. With all other resources, editing a compound resource does not affect the statement whose object it is. With collections, this is not the case.

Example. If a task "put up Christmas tree" initially has no subtasks, expressed as an empty list. Then one cannot select this empty value and add more subtasks to it:

```
:PutUpChristmasTree rdf:type ex:Task ;
    ex:subtasks rdf:nil .
```

Solution. SKOS gives collections stable identities, by introducing an indirection. A resource of type `skos:OrderedCollection` becomes the permanent anchor and holds the actual list in the property `skos:memberList`. If the above example uses this vocabulary, the value of property `ex:subtasks` is a resource that is stable during editing.

```
:PutUpChristmasTree rdf:type ex:Task ;
    ex:subtasks :Subtasks .
:Subtasks rdf:type skos:OrderedCollection ;
    skos:memberList rdf:nil .
```

Inferred membership

Problem. There is no way to use SPARQL for list elements, for example, to retrieve all lists that have a given element.

Example. Continuing the example of the task of putting up a Christmas tree, let us define this task to have the subtasks "buy, "install", and "decorate".

```
:Subtasks rdf:type skos:OrderedCollection ;
    skos:memberList ( :buy :install :decorate ) .
```

There is no easy way to find the list of which, for example, `:install` is a member, as the above collection is stored as a chain of resources and this kind of construct is not supported by SPARQL.

Solution. The SKOS reference [MB] suggests the following (custom) inference rule:

> **(S36)** For any resource, every item in the list given as the value of the `skos:memberList` property is also a value of the `skos:member` property.

That means that an inferencer [1] that supports this assertion would infer the following RDF from the example above:

```
:Subtasks
    skos:member :buy ;
    skos:member :install ;
    skos:member :decorate .
```

This makes it easy to find the list, given an element of it.

12.2.3 RDF containers

Note that containers, instances of `rdf:Seq` don't even exhibit the problems that are solved for `rdf:List` here. They always have a stable identity. Additionally, all container properties are subproperties of `rdfs:member`. Thus, given a container

```
:MySeq a rdf:Seq ;
    rdf:_1 "a" .
    rdf:_2 "b" .
```

a standard RDFS inferencer can perform inferences similar to the ones above:

```
:MySeq rdfs:member "a", "b" .
```

An additional advantage of a container is that its elements are direct properties which simplifies parsing and removing a container.

12.3 N-ary relations

In RDF, normal relations, as defined via statements, are binary: Their elements are edges connecting subjects and objects. The predicates are the labels of those edges. While binary relations are fine for most applications, sometimes, one needs to relate more than two participants. In this section, we describe the three most common use cases. We also refer to the corresponding use cases in the note "Defining N-ary Relations on the Semantic Web" (DNRSW, [NR]) which has been published by the Semantic Web Best Practices and Deployment Working Group.

[1] OWL is not powerful enough to make this kind of inference.

What all use cases have in common is that the edge is modeled as an explicit resource that is connected by properties to all participants of the edge. Properties pointing to the edge resource are called *source properties*, the resources doing the pointing are called *source resources*. The edge resource consists of two kinds of properties: *edge properties* contain data that is considered to belong to the edge. One or more *target properties* are the arrows of the edge and point to the so-called *target resources*.

12.3.1 Single-target edge

Problem. The atomic unit of data in RDF is the statement. Especially in multi-user applications being able to annotate the smallest possible unit is useful. It allows one, for example, to assert who created the statement. RDF has a standard mechanism for this kind of annotation which is called *reification*, because it reifies a statement as a resource. Alas, reification is brittle and takes extra time to look up. It also wastes storage space, especially if long literals are involved.

Example. Joe has tagged the book "Resurrection" with "interesting". The usual way of expressing this is as the statement

```
:Resurrection ex:tag :interesting .
```

For multi-user applications, one wants to annotate this statement with who added the tag and when.

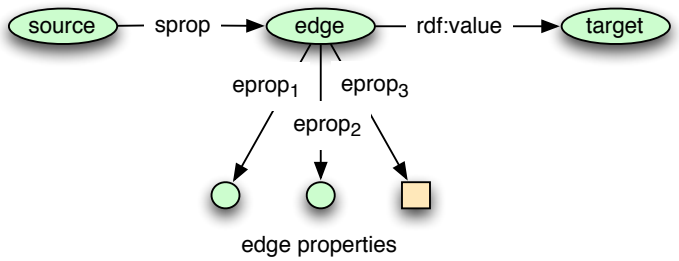


Figure 12.1: Single-target edge: a binary edge with edge properties.

Solution. [DNRSW use case 1] A single-target edge has a single source property of cardinality one and a single target property of cardinality one. Thus, one has a traditional binary edge, but with the ability to add edge properties to it. RDFS provides the standard property `rdf:value` to mark the main property of a compound value (Sect. 5.2.1). So this property can either be the target property or be made a superproperty of the target property (in which case RDFS inference adds it to the edge).

This pattern is used for *broad facets* (Sect. 7.3.2), the solution to the above mentioned problem. The following RDF records that Joe assigned the tag and when. It uses the Common Tag[2] RDF vocabulary.

```
:Resurrection ctag:tagged [
    a ctag:ReaderTag ;
    ctag:means :Interesting ;
```

[2] http://commontag.org/

```
    foaf:maker :Joe ;
    ctag:taggingDate "2009-02-13"^^xsd:date
] .
```

12.3.2 Multi-target edge

Problem. A property has a compound value, with one part of the compound value considered to be more important than the rest.

Example. A product item weighs 2.4kg. Thus, the value of property "weighs" has two parts, first the amount "2.4" and second the unit "kg". The amount is more important, as the unit might be standardized and then mentioning it only serves descriptive purposes.

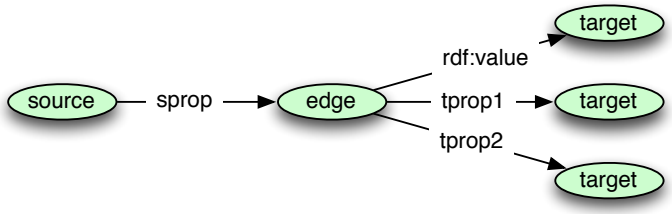


Figure 12.2: Multi-target edge. If one of the target properties is considered primary, one can use (or infer) `rdf:value`.

Solution. [SWBP use case 2] A multi-target edge has a single source property and several target properties. It can be interpreted as a property that has a compound value. `rdf:value` can be used for the main property or inferred as a superproperty of the main property, similarly to the single-target edge. The following RDF [MM] expresses the fact that product item 10245 weighs 2.4kg.

```
exproduct:item10245 exterms:weight [
    rdf:value "2.4"^^xsd:decimal ;
    exterms:units exunits:kilograms
]
```

12.3.3 Assignment of roles

Problem. An event happened that binds together several role players. This event should be manifested as a resource.

Example. Tracy has purchased the book "Where the wild things are" as a birthday gift. She used the online bookseller books.example.com and paid 15 Euros.

Solution. [SWBP use case 3] In this case the function of the edge is still to relate several resources, but none of these resources stands out, so the *instance* of relating (the edge) is conceptually very close to a "normal" resource. The edge is a mapping

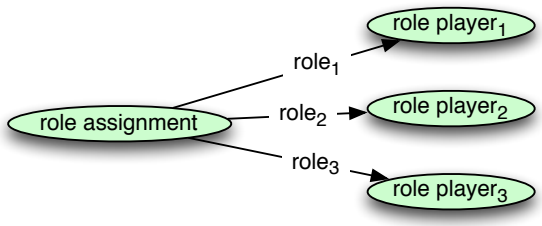


Figure 12.3: Role assignment

from role to role players. The following RDF (adapted from [NR]) expresses Tracy's purchase.

```
:Purchase_1 rdf:type ex:Purchase ;
    ex:buyer :Tracy ;
    ex:object :Wild_things ;
    ex:purpose :Birthday_gift ;
    ex:amount 15 ;
    ex:seller :books.example.com .
```

12.4 Configuration

In CoIM, components (both standard components and plugin components) can consume RDF configuration data which is provided by either components or an end user. The challenge is to define public means for contributing such data; this usually involves defining a public URI and the role it plays.

12.4.1 Configuring a single value

Problem. An end user can set a preference that is honored by the framework.

Example. As part of HYENA's *repository settings*, one can define a resource to select when a repository is first displayed in a web browser.

Solution. One defines a public URI and a property where the configuration value is to be stored. An instance lens can help end users with editing. Applied to the example, `hyena:RepositorySettings` is a resource for end user configuration. The property `hyena:startPage` specifies the resource to show when starting up. This leads to RDF similar to the following.

```
hyena:RepositorySettings hyena:startPage page:3 .
```

12.4.2 Contributing a set of complex values

Problem. A component needs a set of configuration entities that each comprise several values.

Example. Both components and end users can contribute namespaces. A namespace has a URI and a prefix (a short alias for the URI).

Solution. One defines a public type. To retrieve the configuration values, one looks for all instances of that type. A class lens can support the user with editing the instances. Applied to the example, `remm:Namespace` is the class of namespace definitions. Defining two namespaces `dcterms` and `foaf` leads to the following RDF:

```
[] rdf:type remm:Namespace ;
    remm:prefixName "dcterms" ;
    remm:uriref "http://purl.org/dc/terms/" .
[] rdf:type remm:Namespace ;
    remm:prefixName "foaf" ;
    remm:uriref "http://xmlns.com/foaf/0.1/" .
```

12.5 Discussion

The SKOS vocabulary for encapsulating sets as resources and for fixing some of the RDF collection problems are not yet supported by HYENA. Mentioning this vocabulary in this chapter has been motivated by the difficulties the author encountered when implementing collection editing. The pattern for a single-target manifested edge is used to add context information to broad facets (Sect. 7.3.2). The configuration patterns were derived from HYENA's handling of data extension points.

Part IV

The RDF editing meta-model

The RDF editing meta-model (REMM) is a comprehensive solution for RDF editing. The challenge was to bring form-based editing as implemented by, say, XML schema based editors, to RDF. For RDF, more work is required, because the data of a resource is usually unordered and a few of its data structures are complicated to edit. REMM is based on the Fresnel display vocabulary, which it extends with support for editing.

This part starts with a chapter giving a longer introduction and then continues with chapter describing the components of REMM: A few aspects of OWL make it unsuited for simple, intuitive editing (Sect. 5.4.5). As an answer, *REMM schema* has been created. It is a different way of interpreting OWL and contains formal definitions for operations that are needed by RDF editing (such as computing the schema of a class). *REMM presentation* specifies what parts of a resource are to be displayed or edited: What properties, in what order, how deep to descend into nested resources, etc. It also specifies how to display or edit: what widgets, explanatory text, etc. *REMM editing* describes how the specifications made in REMM presentation are turned into trees of RDF data that are then changed and written back to RDF.

For RDF presentation and editing, there is no solution that is as comprehensive as REMM and encoded in RDF. Furthermore, REMM is based on the Fresnel vocabulary that is already widely in use. Its type system avoids many complications of OWL, while being compatible with it. With REMM, RDF feels similar to frame-based databases. The author is not aware of any simple type systems for RDF.

Chapter 13

Introduction: The RDF editing meta-model (REMM)

Contents

13.1 Overview

This chapter serves an an introduction to the following four chapters on the *RDF editing meta-model* (REMM): "REMM schema" (Chap. 14), "REMM presentation" (Chap. 15, "REMM editing" (Chap. 16), and "Configuration in RDF" (Chap. 17). REMM provides standards and techniques for implementing RDF editing: It defines an RDF vocabulary for editing and clearly specifies the semantics of this vocabulary. It also sketches user interface mechanisms to illustrate how the vocabulary would be used in practice. A previous version of REMM has been published as a book chapter [Rau08a].

13.2 RDF vocabularies that REMM is based on

RDF has many existing standards that are more or less related to editing and presentation:

1. Schema: The *RDF Schema Language* RDFS [BG04] and the *Web Ontology Language* OWL [B+a] are schema[1] languages for describing the structure of data.

[1] For this chapter, we consider ontologies to be a superset of schemas.

2. Presentation: The *Fresnel Display Vocabulary* [BPKL06] helps with presenting the data. It declaratively specifies in RDF, how RDF data should be formatted and laid out. This means that you can package both the data and instructions on how to display it in the same RDF graph.

3. Reversible embedding for publication: *RDFa* [AB06] extends XHTML so that RDF data can be embedded inside it. This means that the process of merging unstructured data (XHTML) and structured data (RDF) for publication becomes reversible; tools become feasible that, when given a web address, can extract the data embedded in it, in a clearly defined, unambiguous way.

4. Querying: The *SPARQL query language* [PS05] aids in flexible data retrieval and is used in advanced Fresnel applications.

5. Computed data: Not all information has to be explicitly stated, some of it can be derived from existing data. OWL inferencing is one mechanism for such derivation. More powerful, rule-based approaches are currently being standardized [Rul] for RDF.

REMM is partially based on (1), (2) and (3). It is a reflection on how these vocabularies can be used and extended to better support data modeling. Both OWL and Fresnel are lacking for this task: OWL is too complex for basic data editing, as stated by Hendler [Hen06]. By its very nature, OWL does not support editing-specific views on RDF data: For a class of resources, such a view would specify what properties to edit (while ignoring the rest), how to display them and in what order. Fresnel has been created to complement OWL in this regard and calls editing-specific views *lenses*. But Fresnel is only concerned with displaying data and lacks several crucial features for editing. The REMM is a combination of subsets and extensions of existing standards and is split into three parts:

- Schema (Sect. 14): We specify a simple type system that represents a simpler way of interpreting OWL and provides everything that is needed for most data editing applications. We also introduce some constructs that OWL is missing.

- Presentation (Sect. 15): We restrict and extend Fresnel to suit our purposes. Some parts of Fresnel are too advanced for the current iteration of the REMM, some things are naturally missing, as Fresnel was never intended to support editing.

- Editing (Sect. 16): Data structures and algorithms that are necessary during the actual editing of RDF. We never edit RDF directly, but change it in a three step process: First a lens is used to create a *projection* of the RDF to be edited. This projection is a tree-structured view on the RDF that reflects the definitions of the lens. Second, the user changes the data, rearranging the nodes of the projection as she pleases. Third, the projection is *applied* to RDF: The changes encoded inside the projection are committed to RDF.

In order to better illustrate how the REMM could be implemented, we also comment on challenges when implementing a graphical user interface for the REMM. We assume familiarity with RDF. Knowledge about OWL and Fresnel is helpful, but we try to explain most of the basic ideas.

13.3 Conventions used in this document

13.3.1 Compliance with standards

The following markers are used to indicate how we extend or restrict existing standards.

Ignored	A feature of a standard is ignored, because it is not (nor will it ever be) useful for editing.
Future	We postpone implementing a feature, but it is likely that it is useful for editing.
Extension	A standard lacks a feature that we need. We thus have to define it ourselves.

13.3.2 Roles of REMM users

There are three kinds of REMM users:

- REMM implementors: are programmers that implement the REMM specification.
- Experts: are assumed to have in-depth knowledge of RDF and REMM. They create lenses etc. for *end users*.
- End users: work with what the experts have created. They should be shielded from the complexities of RDF as much as possible.

Subsequently, when explaining a REMM feature, these roles will sometimes be mentioned to make the intended audience and the requirements of that feature clear.

13.4 Building blocks for data modeling in RDF

We try to shield the user from some of the complexities of RDF and never edit RDF data directly, but through an intermediate tree-structured view, a so-called *projection*. A projection is built from a few basic pieces, it has its own data-building vocabulary, if you will. This vocabulary is reminiscent of object-oriented data modeling and knows of the following categories of RDF nodes:

- Atomic data: Sometimes we are only interested in a single RDF node which then forms *atomic* data. Literals are always atomic, but resources are sometimes, too, if they are used as symbols and encode something, for example a country or a web site address. Atomic data constitutes the leaves in the projection tree.
  - Enumerations: Enumerations are classes that explicitly enumerate all of their instances. OWL distinguishes between an enumerations of literals (“enumerated datatypes”) and enumerations of resources (“enumerated classes”); we seldom make that distinction. An enumeration mainly provides options to choose from. The data itself, the “currently selected node”, is always atomic.
- Compound data: Resources can contain data, making them compound; they are the inner nodes of the projection tree. The REMM knows of two compound data structures:

- Records: Resources whose properties are seen as a set of (key,value) pairs.
- Assemblies: A sequence of RDF nodes, encoded as either an RDF collection or an RDF container.

13.5 The main REMM constructs

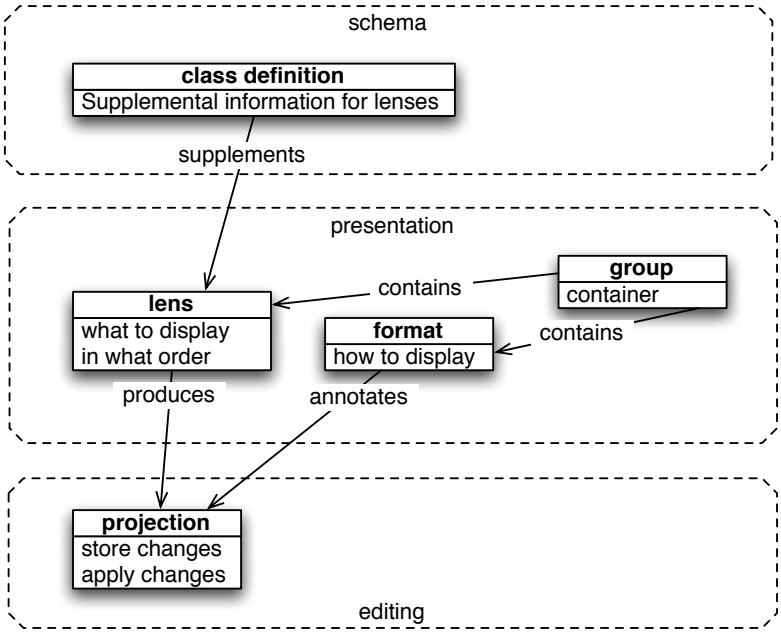


Figure 13.1: The REMM is partitioned into three layers: schema, presentation and editing. Except for projection (which is purely a programming language data structure) each of the constructs in the layers is defined by an RDF resource.

The REMM has three layers: schema, presentation and projection (Fig. 13.1). The schema defines the structure of the data. The presentation both selects what data to display (it acts much like a database view this way) and how to display it (style information, if you will): The so-called *lens* is the view, the *format* contains the style information and *groups* are a package mechanism for both lenses and formats. The editing layer uses *projections* to encode, visualize and apply changes to RDF data. Note that the projection is purely a programming language data structure, while all other constructs are defined in RDF.

13.5.1 Presenting the projection

We suggest that implementors of the REMM use two presentation modes: In *edit mode*, the user has the typical widgets for changing data, such as text fields and combo boxes. In *display mode*, we mainly show read-only text and make that text easy to copy. In display mode, the REMM behaves like Fresnel. Sect. 15.6.3 explains how one specifies the widgets to use for editing and display.

13.5.2 Lens-based tables

Using lenses to define tables is currently an experimental REMM feature.HYENA can display such tables, but their cells are not editable. A lens table has one column per property. Challenges include what to display in a cell if a property has multiple values and/or a compound value such as a collection.

13.6 The user interface: REMM in use

In this section, we want to give a feeling what the "finished product" should look like. There are established standards for editing record-based data, as embodied by database programs such as Microsoft Access or Filemaker. We want RDF editing to feel much the same way. We distinguish two kinds of editing: Meta-editing is performed by experts and involves creating schema, lenses, formats, and groups. Normal editing is performed by end users and involves creating instances and using lenses. We first take a look at normal editing.

13.6.1 Normal editing

The following scenarios require user interface answers:

- Creating new instances: How does the user create new data?
- Editing an instance: How does the user edit newly created or existing data?
- Batch editing: We want to give the user the opportunity to edit not only single instances, but also *sets* of instances. For example, if she wants to set a property of several instances to the same value.

Creating new instances

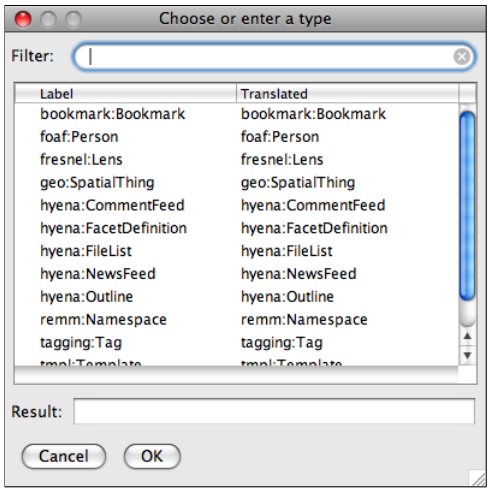


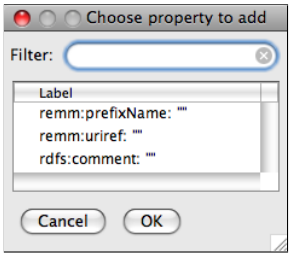
Figure 13.2: Creating a new instance.

When the user decides to create a new instance, she will usually know what kind of instance it should be. Thus we present her with a list of classes which comprises all instances of `rdfs:Class` (Fig. 13.6.1). Obviously, it is the responsibility of an expert to create these beforehand. We also give the opportunity to create untyped resources or to enter a type manually. Note that we currently do not support a new instance being a member of multiple classes. We consider this an advanced operation that could be implemented as adding more class memberships to an instance after instantiation. Whether to make this operation available to the end user or not and how is left to the REMM implementor.

Displaying versus editing an instance

(a) display instance

(b) edit instance



(c) add property

Figure 13.3: Instance editing.

Instance editing happens either after the user has created a new instance or if he encounters existing data. One needs a lens in order to edit a resource, but REMM implementations may provide a default lens for editing any resource. As editing controls add considerable visual clutter to a user interface, we have two modes for accessing instances:

- In *display mode* (Fig. 13.3(a)), data is read-only and can easily be copied. We simply display the currently selected resource. This is exactly what a web browser gives you when used with normal Fresnel.
- In *editing mode* (Fig. 13.3(b)) we provide the usual editing controls such as text fields and combo boxes. One can change property values, remove properties and add new properties. This last operation brings up a list of predicates (Fig. 13.3(c)) from which the user can choose the one she wants to add. If the range of a property comprises several classes, we display the same predicate several times, each time with a different class.

The expert may define constraints on the data, the main representative being *cardinality constraints* which state how often a given property can exist. Examples are: "there must be exactly one last name", "there must be at least one phone number". We allow these constraints to be violated at any time. For example, a new instance is always created empty, even if there are properties whose minimum cardinality is greater than 0. But we indicate violations with error messages.

Batch editing

Whereas in single-instance editing, one sees a lot of parsed data, batch editing usually starts out with a display that is mostly empty. On can then specify what property values are to be added and what properties are to be removed. The latter operation is an explicitly displayed "remove all" that can be mixed with the former operation. Thus, in addition to a plus icon for adding properties, there is also a minus icon for removing properties. For example: Remove all first names from a set of resources and add the single first name "Joe" to all of them.

13.6.2 Meta-editing

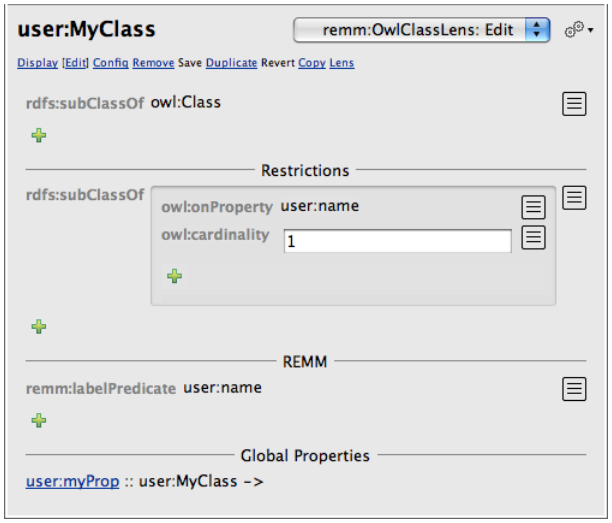


Figure 13.4: Class editing. The identifier of the class is not shown in this figure, because it is the same as the currently selected node which is displayed in a user interface element that is external to this pane.

Meta-editing refers to the expert editing classes and lenses. For class editing (Fig. 13.4), we display many properties as you would for instance editing. Two kinds of data associated with a class require special treatment, though: restrictions make more sense if they are inlined with the class, as they are the closest thing that OWL has to declaring what properties a class contains. Lens editing with REMM is self-hosting: You can define a lens to edit lenses (Sect. 15.8.3).

Chapter 14

REMM schema

Contents

14.1 Overview

With the schema, the expert defines the shape of the data: What properties can an instance have? What is the cardinality of a property? What values can it be assigned? And so on. Ideally, one would use RDF schema and OWL for this purpose, but RDFS is not powerful enough and OWL has deficiencies when it comes to intuitive modeling (Sect. 5.4.5). Thus, this chapter presents a simple, intuitive type system that provides several services crucial to editing:

- Checking whether a resource is an instance of a given class. This allows one to use a class to state what resources can be edited with a lens.
- Creating a *default instance* for a class. This allows one to add new properties to a resource by just picking a property key, a preliminary property value can be automatically created from the property range.
- Computing the schema of a class. For editing, the schema is used to determine what resource are *valid*. A resource is valid if all of its properties have the correct cardinality and if each property value is in the range of the property.

Because the semantics of OWL and the type system are so different, there is no direct translation from OWL. Instead, this chapter defines a loose translation that honors the meaning that is usually intuitively (and not always entirely correctly) given to OWL constructs.

14.2 A type system for lightweight RDF editing

REMM schema borrows OWL's class-centric approach (as opposed to the property-centric approach of RDFS). To make modeling intuitive, three assumptions are different from OWL: First, a closed-world perspective is taken. For example, missing properties cause a warning. Second, names are assumed unique unless specified equivalent. Thus, if a property has a maximum cardinality of one and two resource values, they are not inferred to be equivalent; instead, a warning is given. Third, classes are assumed to be disjoint unless declared equivalent.

In REMM schema, A *class* is a set of RDF nodes. A class is identified by a resource, its identifier. Every node is a member of one or more classes which are called the *types* of the node. The node is an *instance* of each of its types. A class can contain two kinds of schema information: the keys, ranges and cardinalities of properties and an enumeration of the instances (then it is called an *enumerated class*). The properties are used for consistency checks and for guided creation of instances, where a user is shown what properties she can add to an instance and what values they are allowed to have. Classes can also be used for *matching*, to check whether an RDF node is an instance or not, and for instance creation.

Kinds of classes

Abstract versus concrete classes. Classes can be arranged in a directed acyclic graph denoting inheritance. Every class inherits the schema of its ancestors. A class can be declared *abstract* meaning that it cannot be instantiated, only (possibly) its subclasses. It can still be used for matching instances of its subclasses. For example, if a property can have values that are either instances of `Man` or of `Woman`, its range is `Human`, an abstract superclass of both classes. When we check if a property value is valid, we match against `Human` which accepts all instances of `Man` and `Woman`. When it comes to instantiating new property values, `Human` itself is never directly mentioned; we only offer to instantiate either `Man` or `Woman`.

Internal versus external types. The types of a node that can be determined by examining just the node are called *internal*. Resources can also have *external* types, which are attached to them via the `rdf:type` property. All literals have internal types. Among the resources, `rdf:nil` has the internal type `rdf:List`. Elements of an enumerated class are also considered to have that class as an internal type, even if they don't have an `rdf:type` property. Instantiating a node with internal types means creating a node that has a certain structure. Instantiating a node with external types means creating a new resource and attaching those types to it.

REMM uses the following special classes, to cover several aspects of nodes that are missing from RDF schema.

- `remm:ResourceNode` $\subset$ `rdfs:Resource`: The class of all resources. Mainly used for matching. Note that `rdfs:Resource` is the class of *all* RDF nodes, not just of resources. Specifically, `rdfs:Literal` is a subclass of `rdfs:Resource`.
- `remm:UriNode` $\subset$ `remm:ResourceNode`: The class of all URI nodes.
- `remm:BlankNode` $\subset$ `remm:ResourceNode`: The class of blank nodes.

- `rdf:PlainLiteral` $\subset$ `rdfs:Literal`: The class of plain literals[1].

The following sections express the previous explanations more formally.

14.2.1 Basic sets

The following sets are taken from RDF:

Node	The set of all RDF nodes
Literal $\subset$ Node	RDF literals
Res $\subset$ Node	RDF Resources
URI $\subset$ Res	RDF URI nodes

14.2.2 Properties

The set Prop of properties comprises quadruples

$$\text{prop}(uri, range, minCard, maxCard)$$

with the components

$uri \in \text{URI}$	is the *predicate*, a URI.
$range \in 2^{\text{Res}}$	is the *range*, the resources of classes.
$minCard \in \mathbb{N}$	is the *minimum cardinality*.
$maxCard \in (\mathbb{N} \cup \{\infty\})$	is the *maximum cardinality*, which is either an integer or unconstrained.

Combining properties

When collecting all properties (inherited and original) of a class, several properties might have the same URI. Then those properties need to be combined into a single one. The operations shown below perform this duty. They use the subset relation $\sqsubseteq$ on the URI of classes which is defined later.

Property intersection. Property intersection is symmetric and defined as follows:

$$\text{prop}(p, r_1, a_1, b_1) \sqcap \text{prop}(p, r_2, a_2, b_2) = \begin{cases} \text{prop}(p, \texttt{min}(r_1, r_2), \texttt{max}(a_1, a_2), \texttt{min}(b_1, b_2)) \\ \qquad \text{if } \texttt{min}(r_1, r_2) \text{ is defined and } \texttt{max}(a_1, a_2) \leq \texttt{min}(b_1, b_2) \\ \textit{undefined} \qquad \text{otherwise} \end{cases}$$

$\texttt{min} : \text{URI} \times \text{URI} \to \text{URI}$ is defined as

$$\texttt{min}(t, u) = \begin{cases} t & \text{if } t \sqsubseteq u \\ u & \text{if } u \sqsubseteq t \\ \textit{undefined} & \text{otherwise} \end{cases}$$

Property restriction. For property restriction, the restricting property must be more specific than the restricted property:

$$\text{prop}(p, r_1, a_1, b_1) \circledcirc \text{prop}(p, r_2, a_2, b_2) = \begin{cases} \text{prop}(p, r_1, a_1, b_1) & \text{if } r_1 \sqsubseteq r_2 \text{ and } a_1 \geq a_2 \text{ and } b_1 \leq b_2 \\ \textit{undefined} & \text{otherwise} \end{cases}$$

[1] http://www.w3.org/TR/rdf-text/

14.2.3 Class hierarchies

A class hierarchy $\mathcal{H}$ is a triple $(\mathrm{Class}, \mathrm{Abstr}, \mathrm{Sub})$ where

- Class is the set of classes. A class is a triple

  $$\mathrm{class}(id, \{p_1, \ldots, p_m\}, \{i_1, \ldots, i_n\})$$

  with an $id \in \mathrm{Res}$, properties p_k, and enumerated instances $\{i_1, \ldots, i_n\} \subset \mathrm{Res}$. If the instances are not relevant or empty, the class can be abbreviated to a pair.

- $\mathrm{Abstr} \subset \mathrm{Res}$ contains references to all *abstract classes*.

- $\mathrm{Sub} \subset \mathrm{Res} \times \mathrm{Res}$ is the subclass relation, a directed acyclic graph.

Class contains all classes defined in RDF schema (such as `rdfs:Resource` and `rdfs:Literal`). For a Resource r the element relationship with Class is defined as:

$$r \in \mathrm{Class} :\Leftrightarrow \exists P, I : \mathrm{class}(r, P, I) \in \mathrm{Class}$$

Transitive subclass relation. The transitive subclass relation is defined as

$$d \sqsubseteq c :\Leftrightarrow (d, c) \in \texttt{transrefl}(\mathrm{Sub})$$

where

> `transrefl` : $D \times R \to D \times R$ computes the transitive reflexive closure of a binary relation.

Sets of resources are treated as intersections and related as follows.

$$D \sqsubseteq C :\Leftrightarrow \forall d \in D : \forall c \in C : d \sqsubseteq c$$

14.3 Operations on class hierarchies

RDF editing needs several operations on the above defined structures. These are described below.

14.3.1 Determining the types of a node

This section defines how the types of a node are determined. This definition is needed by later operations. It consists of the function

$$\texttt{types} : \mathrm{Node} \to 2^{\mathrm{Class}}$$

which is defined as

$$\texttt{types}(n) := \begin{cases} \{\texttt{ltype}(n)\} \cup \texttt{etypes}(n) & \text{if } n \text{ is a literal} \\ \{\texttt{rdf:List}, \texttt{remm:UriNode}\} & \text{if } n \text{ is } \texttt{rdf:nil} \\ \texttt{rtypes}(n) \cup \texttt{etypes}(n) & \text{otherwise} \end{cases}$$

Helper functions:

- `ltype` : Literal $\to$ URI determines the type of a literal. A plain literal has the type `remm:PlainLiteral`, a language tag literal has the type `remm:LanguageTagLiteral`, and the type of a typed literal is its datatype.

- `rtypes` : Res $\to 2^{\text{URI}}$ computes the external types of a resource, the types attached via the property `rdf:type`. If a resource has no external types, {`rdfs:Resource`} is returned. To this result, one adds `remm:UriNode` if the resource is a URI, and `remm:BlankNode` if it is a blank node.

- `etypes` : Node $\to 2^{\text{URI}}$ finds the enumerated types of which a node is part of.

$$\texttt{etypes}(n) = \{u \mid \text{class}(u, P, I) \in \text{Class} \wedge n \in I\}$$

Theorem 1 (Uniqueness of types)**.** *Every node has a unique set of types.*

Proof. For each node n, the enumerated types $\texttt{etypes}(n)$ are uniquely defined. If n is a literal, it is always a member of exactly one kind of literal, which determines the value of $\texttt{ltype}(n)$. Thus, $\{\texttt{ltype}(n)\} \cup \texttt{etypes}(n)$ is uniquely defined.

If n is a resource, it is either `rdf:nil` or not. In the former case, it has two uniquely defined types, in the latter case, $\texttt{rtypes}(n)$ is uniquely defined and thus also $\texttt{rtypes}(n) \cup \texttt{etypes}(n)$. □

14.3.2 Instance-of checking

The check n `instof` c determines whether a node n is an instance of a class c. Instance-of checking is used to tell if a lens applies to a node (by means of *selectors*, Sect. 15.3). The check is defined as

$$n \texttt{ instof } c :\Leftrightarrow \exists d \in \texttt{types}(n) \,.\, d \sqsubseteq c$$

Theorem 2. *All instances of a class are also instances of its superclasses.*

$$n \texttt{ instof } c \wedge c \sqsubseteq b \Rightarrow n \texttt{ instof } b$$

Proof. According to the definition of `instof`, there is a $d \in \texttt{types}(n)$ such that $d \sqsubseteq c$. Due to the transitivity of $\sqsubseteq$, the assertion $d \sqsubseteq b$ also holds and thus n `instof` b □

14.3.3 Default instances of classes

When adding a property to an instance, the REMM user interface gives it a default value. This section defines how, given a class, one creates an instance for it. To create a default value for a property, one applies this operation to a randomly chosen element of its range. Creating an instance distinguishes between external and internal classes: External classes are all instantiated the same way: one creates a new resource and attaches the type via the `rdf:type` property. For internal types, one can use a *default instance* as a preliminary place holder that might later be changed.

Default instances for internal types

The function `dflt` : Res $\to$ Node maps (the resources of) classes to default instances:

- Literal classes are mapped to literals. The default is to map `rdf:PlainLiteral` to `""` and subclasses d of `rdfs:Datatype` to `""^^`d. `rdfs:Datatype` itself and `rdfs:Literal` are abstract. A few datatypes are mapped differently from this rule. The result is chosen so that if it is the value of a property, it counts as "first time", "true", or "switched on", as opposed to the same property not being there meaning "never", "false", or "switched off". Examples of such properties are `ex:numberOfCars` and `hyena:updateModified`. Accordingly, the default instance is `"true"` for booleans, `"1"` for integers, and the current date or time for time-related datatypes.
- `rdf:List` is mapped to `rdf:nil`.
- Enumerated classes are mapped to the first enumeration member.
- All other URIs are mapped to $\bot$.

Theorem 3. *The default instance of a class is an instance of that class.*

$$\forall u \in \mathrm{URI} : \mathtt{dflt}(u) \neq \bot \rightarrow \mathtt{dflt}(u)\ \mathtt{instof}\ u$$

Proof. Trivial, because the result n of $\mathtt{dflt}(u)$ has been chosen so that $\mathtt{types}(n)$ contains u. □

Construction sets

A *construction set* answers the question how a given type t can be instantiated. Depending on t, one or more alternatives for instantiation are offered. Each alternative is one element of the construction set.

- If t is an internal type, a single instantiation alternative is offered—the default instance n of t. This is written as $\{\mathsf{cinst}(n)\}$.
- If t is a concrete class, it can be directly added as the type of a new resource. This is written as $\{\mathsf{cclass}(t)\}$.
- Abstract types cannot be instantiated themselves, one has to instantiate one of their concrete subclasses. Thus, each concrete subclass becomes one alternative.

These three cases cannot be mixed. If that happens, then, as a fall-back, one assumes that all elements of T are concrete and external. Formally, the function `constrs` maps a URI t to a construction set:

$$\mathtt{constrs}(t) = \begin{cases} \{\mathsf{cinst}(\mathtt{dflt}(t))\} & \text{if } \mathtt{dflt}(t) \neq \bot \\ \{\mathsf{cclass}(t)\} & \text{if } t \in \mathrm{Class} \wedge t \notin \mathrm{Abstr} \\ \bigcup_{c \in \mathtt{concr}(t)} \mathtt{constrs}(c) & \text{if } t \in \mathrm{Abstr} \\ \emptyset & \text{otherwise } (t \notin \mathrm{Class}) \end{cases}$$

Helper function:

- `concr` computes the concrete subclasses of an abstract class.

Converting an element of a construction set to an instance is defined as follows. The result is a pair of a node and a set of triples. To actually create the node that is returned, the triples have to be added to the RDF repository.

$$\begin{array}{lcl} \mathtt{newinst}(\mathsf{cinst}(n)) & = & (n, \emptyset) \\ \mathtt{newinst}(\mathsf{cclass}(c)) & = & (r, \{(r, \mathtt{rdf:type}, c)\}) \\ & & \text{where } r \text{ is a fresh resource} \end{array}$$

Note that the instances created might not be valid (see definition below), because the schema might demand properties to have a cardinality greater than 0. This is by design, as creating instances with correct cardinalities can become quite complex and editing must allow ill-formed instances, to cope with inconsistent data. Then the errors are pointed out and the user can fix them.

Theorem 4.

$$\forall t \in \text{Uri} : \forall e \in \texttt{constrs}(t) : \texttt{newinst}(e)\ \texttt{instof}\ t$$

Proof. Looking at the cases of the definition of `constrs`:

- t has a default instance: A default instance of a class is an instance of that class (see previous theorem).
- t is a concrete class: The special cases `rdf:nil`, enumerated types and literals are handled as default instances. Thus, the result will be a resource with an external type. This type will be returned by `types` (last case of function definition), which is why the assertion holds.
- t is an abstract class: Each concrete subclass c of t leads to result elements e such that $\texttt{newinst}(e)\ \texttt{instof}\ c$. As $c \sqsubseteq t$, $\texttt{newinst}(e)\ \texttt{instof}\ t$ also holds.
- t is not the URI of a class: The assertion holds trivially.

□

14.3.4 Schema computation

The algorithm for computing the schema $\texttt{sch}(c)$ of a class c is only sketched here. The result is a set R of properties.

1. Combine the properties of the superclasses of c to a single set P: The result is the union of the individual properties. If two properties have the same predicate, they are merged via property intersection $\sqcap$. If the result of merging is undefined the result of schema computation is also undefined. If there is no superclass, P is the empty set. Intuitively, P is the intersection of the schemas of all superclasses.
2. Add the properties Q of c to the superclass properties P. The result R is the union of both sets; if a property q of Q has the same predicate as a property p of P, then the property put in R is the restriction $q \circledcirc p$. If any of the restrictions are undefined, then the final result is also undefined. Intuitively, R is P restricted by c.

Valid instances

A resource r is *valid* if its properties comply with the ranges and cardinalities in the schema.

$$\begin{aligned} &\mathit{valid}(r) :\Leftrightarrow \forall c \in \texttt{types}(c) : \forall (u, r, a, b) \in \texttt{sch}(c) : \\ &\quad a \leq |\,\texttt{values}(r, u)| \leq b \wedge (\forall v \in \texttt{values}(r, u) : \texttt{types}(v) \sqsubseteq r) \end{aligned}$$

The function

$$\texttt{values} : \text{Res} \times \text{URI} \to 2^{\text{Prop}}$$

is defined as $\texttt{values}(\mathit{res}, \mathit{prop})$ returning all values of property prop of resource res.

14.4 Translation from OWL

By translating from OWL to REMM schema, existing OWL ontologies can be used for data modeling with REMM. Note that the semantics of OWL is not preserved, but rather the translation is a different way of interpreting OWL, similar to how OWL plus constraints [MHS09] interprets OWL definitions as constraints.

In this section, we abbreviate the components of a class hierarchy as follows: Class becomes C, Abstr becomes A, Sub becomes S. Translation from OWL to the type system happens in three steps, each step transforms a class hierarchy (C, A, S) to another one (C', A', S'), beginning with the first step whose input is $(\emptyset, \emptyset, \emptyset)$. The steps are:

1. Translate OWL's class definitions to type system classes.
2. Add global property definitions to the type system classes.
3. Supplement type system classes so that OWL's class equivalence is honored.

The OWL constructs are always listed in the same order as in the OWL reference [B+a].

14.4.1 Class definitions

To translate class definitions, we assume that one can iterate over class definitions (even embedded ones) and that each class definition has an ID. Both of which is the case when OWL is encoded in RDF. The translation is specified by rules, one per OWL construct. Each rule describes the effects on the class hierarchy triple. The right side of the arrow is the new triple (without parentheses), unchanged trailing components are not shown. Neither is the old triple, which is always implied. Thus a rule

$$(C, A, S), \langle\text{owl construct}\rangle \mapsto (C \cup X, A \cup Y, S \cup Z)$$

is abbreviated as

$$\langle\text{owl construct}\rangle \mapsto C \cup X, A \cup Y, S \cup Z$$

Embedded constructs

Ignored: $\mathrm{complementOf}$

$$u@\mathrm{unionOf}(d_1, d_2) \mapsto \\ C \cup \{\mathrm{class}(u)\}, A \cup \{u\}, S \cup \{(d_1, u), (d_2, u)\}$$

Explanation: assuming the class definition $\mathrm{unionOf}(d_1, d_2)$ has the ID u, it is added as a new class to C, the new class is marked as abstract and becomes a superclass of d_1 and d_2. The following rules work similarly.

$$u@\mathrm{intersection}(d_1, d_2) \mapsto \\ C \cup \{\mathrm{class}(u)\}, A, S \cup \{(u, d_1), (u, d_2)\}$$

$$u@\mathrm{oneOf}(x_1, x_2) \mapsto \\ C \cup \{\mathrm{class}(u, \emptyset, \{x_1, x_2\})\}$$

where the x_i are either individuals or literals.

$$u@\text{restriction}(p\ \text{allValuesFrom}(d)) \mapsto \\ C \cup \{\text{class}(u, \{\text{prop}(p, d, 0, \infty)\})\}$$

Ignored: someValuesFrom, value

$$u@\text{restriction}(p\ \text{minCardinality}(n)) \mapsto \\ C \cup \{\text{class}(u, \{\text{prop}(p, \emptyset, n, \infty)\})\}$$

$$u@\text{restriction}(p\ \text{maxCardinality}(n)) \mapsto \\ C \cup \{\text{class}(u, \{\text{prop}(p, \emptyset, 0, n)\})\}$$

Normalizing properties: if more than one property of a class has the same URI, they are merged into a single property definition via property intersection $\sqcap$. If the result is ever undefined, the whole translation is undefined.

Axioms and facts

Ignored: complete versus partial classes, annotations, individuals.

$$\text{Class}(u\ \mathit{embedded}) \mapsto u@\mathit{embedded}$$

If there are several embedded constructs, they are transformed into a new class that is the intersection of those constructs

$$\text{EnumeratedClass}(u\ i_1, i_2) \mapsto u@\text{OneOf}(i_1, i_2)$$

Ignored: Datatype, DisjointClasses (disjointness is default)

$$\text{SubClassOf}(d_1, d_2) \mapsto C, A, S \cup \{(d_1, d_2)\}$$

14.4.2 Global property definitions

Global property definitions are collected and added to the corresponding class.

$$\text{ObjectProperty}(p\ \text{domain}(d)\ \text{range}(r) \mapsto \\ \texttt{addprop}(C, d, \text{prop}(p, r))$$

Where

$$\begin{array}{lcl} \texttt{addprop}(\{\text{class}(d, P), c_2, \ldots, c_n\}, d, p) & = & \{\text{class}(d, P \cup \{p\}), c_2, \ldots, c_n\} \\ \texttt{addprop}(\{c_1, \ldots, c_n\}, d, p) & = & \{\text{class}(d, \{p\}), c_1, \ldots, c_n\} \end{array}$$

If the class already has a property q with the same predicate as the global property p, the class property restricts the global property and $q \Subset p$ replaces q in the class. If the result of this operation is undefined, the complete translation process is undefined.

Comments:

- The declaration of `super` properties is ignored (could be translated to SubPropertyOf).
- Multiple domains or ranges lead to the creation of new classes (intersections of the range or domain classes).
- Modifier `Functional` inside the ObjectProperty assigns a cardinality of [0,1] to the property.
- Ignored modifiers: `Inverse`, `Symmetric`, `InverseFunctional`, `Transitive`.

Other property definitions are handled as follows:

- `DatatypeProperty` is handled analogously to ObjectProperty.
- `AnnotationProperty` and `OntologyProperty` are ignored.
- `EquivalentProperties`, `SubPropertyOf`: Mainly important for inference, with no simple translation to a schema-based mechanism.

14.4.3 Equivalent classes

Equivalence is an OWL-only construct. If (the URIs of) two or more classes are declared as equivalent, it is as if one had used the same URI for all of them. As REMM schema does not have the notion of class equivalence, the effect has to be simulated: Once all classes have been translated and all global property definitions have been added to them, equivalence is added to the class hierarchy. For each set E of equivalent classes

- C: The union of all properties of E is assigned to each member of E (after appropriate merging).
- A: If one of the classes of E is abstract, all classes are.
- S: For each class ID $e \in E$, one looks for all pairs that contain e and for each different class ID $f \in E$, one adds a copy of the pair to S where e is replaced with f.

14.5 Discussion

This chapter presented an intuitive type system for RDF editing. It stands between the data modeling type systems of database systems and the logic-based type system of OWL: Its notions of consistency and correctness are looser than those of the former, to cope with inconsistent data. OWL makes several assumptions that are necessary for distributed reasoning, but either counter-intuitive or too complicated for data modeling. REMM changes those assumptions. The operations that are defined for the type system reflect the unconventional needs of editing. For example, creating an arbitrary instance of a given class supports REMM's approach to editing, where new properties are initialized with default values.

The translation from OWL to REMM tries to preserve as much of the intuitive meaning of the OWL constructs. Alas, the completely different semantics make any kind of correctness proof impossible.

Chapter 15

REMM presentation: Select, order and style the data to be edited

Contents

15.1 Overview

The schema is used for controlling the structure of the data. The presentation layer of the REMM takes care of displaying the (generally multi-dimensional) data on a two-dimensional medium. The editing process happens as follows (Fig. 15.1): The user selects one or more resources he wants to edit and then selects a *lens* (Sect. 15.5) to edit it with. A lens specifies which properties are shown and which ones are hidden. They thus provide the user with a limited view on the data, showing only those details that are relevant in the current situation. Lenses also define an order in which to display property values; a necessity for editing, because humans derive meaning from such ordering. Lastly, lenses specify via *selectors* (Sect. 15.3) what resources they can be applied to. Given a set of resources, we can thus automatically generate a list of applicable lenses for the user. In some situations, the most appropriate lens might be automatically selected for the user. This process is called *conflict resolution* and explained in Sect. 15.5.3.

Lenses are supported by *formats* (Sect. 15.6) and *groups* (Sect. 15.4). Formats contain more detailed information about how to present the data: What widget to display

it in, what font to use, etc. Format applicability is also specified via selectors. Groups define sets of lenses and formats that should be used in conjunction. As lens and format selection is constrained by the currently active group, groups determine the "look and feel" of the data. The active group is changed manually by the user.

The presentation layer is largely based on Fresnel. The reader can thus consult the Fresnel manual [BLP] for further details.

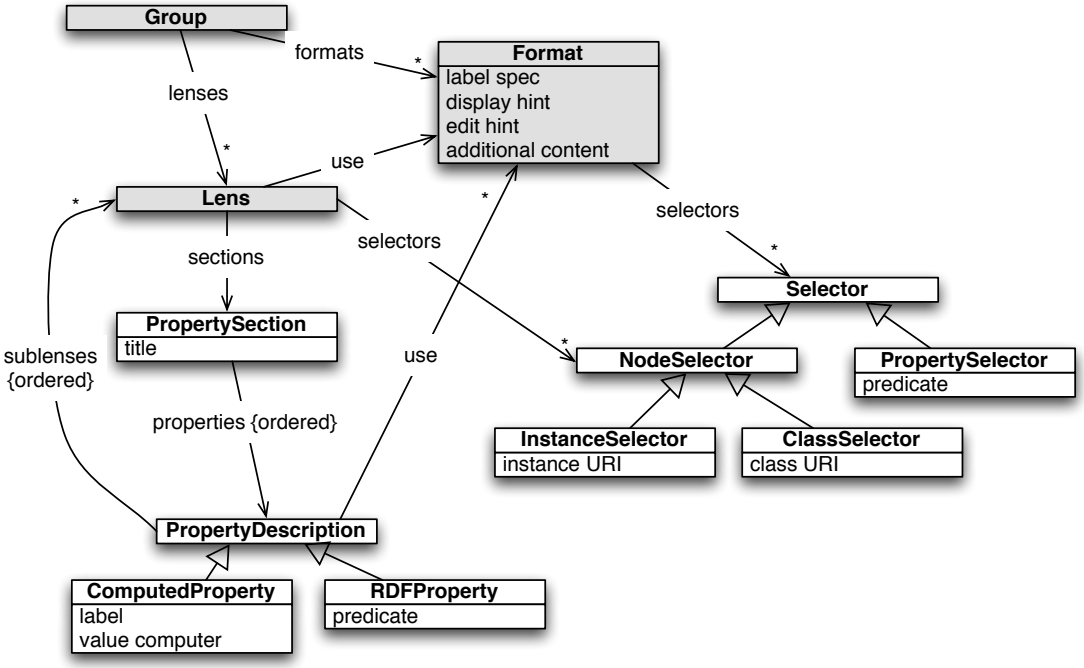


Figure 15.1: The main REMM presentation constructs are the group, the lens, and the format. A group contains lenses and formats. A lens contains properties partitioned into sections. Both formats and lenses use selectors to declare their applicability, but only the former can apply to properties.

Running example. For this chapter, we assume that Shenzi wants to edit movies with REMM and that each movie has a title, a genre, and a description.

15.2 The abstract box model: Laying out RDF data

The Fresnel abstract box model defines a layout grid for RDF data. This grid consists of nested boxes that form a tree (Fig. 15.2). The root of this tree is the so-called *container*, a top-level display element such as a GUI window or a printed page that holds all of the data. Inside the container, there are RDF resources. Each resource contains itself *sections*, groups of properties. Each property is a set of (label,value) pairs[1]. Apart from

[1]The Fresnel manual suggests only displaying the label the first time when a property has several values. It is our experience that it makes sense to display it each time, as values can take up quite a lot of (especially vertical) room, when nested lenses are involved.

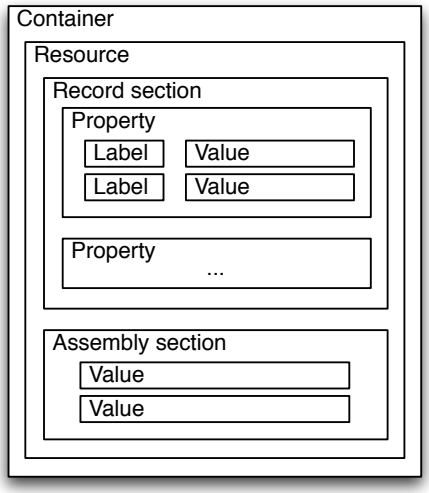


Figure 15.2: The Fresnel abstract box model defines how elements of the RDF data are to be nested when laying them out. The section box is REMM-specific and groups data that is edited similarly (such as all record properties or all members of an assembly); see Sect. 16 for details.

layout, the abstract box model is also referred to when defining styling and inheritance of format data.

15.3 Selectors: Matching resources and properties

Selectors are the matching mechanism that is used by both lenses and formats to specify what instances (and/or properties) they apply to. Selectors are mostly implicit in Fresnel, but we found it helpful to reify them.

15.3.1 Node selectors

Given a resource or a literal and a node selector, we can determine whether the selector *matches* the node or not. This is used to specify whether a lens or format applies to a node. Node selectors are either class selectors or instance selectors.

- Instance selectors: match nodes that are the same as a given node. They are mainly used for lenses that apply to one particular public resource (identified by a URI). One attaches them to lenses and formats via `fresnel:instanceLensDomain` and `fresnel:instanceFormatDomain`, respectively. There are the following kinds of instance selectors:
  - SPARQL selectors: A SPARQL query defines what nodes match a given selector. Not currently used in the REMM.
  - FSL (Fresnel Selector Language [Pie]) selectors: Not currently used in the REMM.
  - Simple selector: Directly states the applicable node. resource.

- Class selectors: Match by looking at the types of a node. Attached to lenses and formats via `fresnel:classLensDomain` and `fresnel:classFormatDomain`, respectively. A class domain of `rdfs:Resource` matches any node.
  - Simple selectors: Directly state the ID of the relevant class. Can optionally be expanded into a resource of type `remm:ClassSelector` with properties `remm:acceptClass` and `remm:rejectClass`. This kind of resource is attached via `remm:lensDomain` and `remm:formatDomain`. Sect. 15.8.2 shows an example of such a selector.

Note that via classes such as `rdf:PlainLiteral`, class selectors can also match literals.

15.3.2 Property selectors

Property selectors are only used in formats, via the property `fresnel:propertyFormatDomain`. There are two kinds of property selectors:

1. FSL selector: An FSL expression that returns properties. Not currently used in the REMM.
2. Simple selector: Directly state the predicate.

15.3.3 Selector specificity

The order in which we have listed the selectors above determines their *specificity*. That is, an instance selector is *more specific* than a class selector. When comparing two class selectors, whichever has a class that is a subclass of the other is considered more specific.

In situations where several lenses or formats apply, we have to decide which one to use. This process is called *conflict resolution* (Sect. 15.5.3) and selector specificity plays a large role in it. For example, if two lenses apply to a resource and one of them has an instance selector, while the other one has a class selector, then the former lens wins.

15.4 Groups: Context-specific containers for REMM constructs

Sets of lenses and formats can define a mode of operation or a different look. A *Group* is the construct to package such a set. Thus, depending on how a user currently wants to work, she chooses the appropriate group. This group is then called the *active group*. Furthermore, a property description in a lens can "use" a group (see below). Then starting with the property value, that group becomes temporarily active, influencing the selection of lenses and formats. There is an implicit group called *group union* which, as its name suggests, is the union of all groups. If it is active, it is as if all groups were active at the same time. A group specifies the following things:

- A set of lenses[2]. When a program shows the lenses that can be used for a given resource, it only considers lenses that are members of the active group. That

[2] In Fresnel, things are actually stated the other way around: lenses and formats say what group they belong to. Conceptually, we found it easier to think about a group as containing lenses and formats.

means that lenses that are not part of any group stay invisible, but might still be used as sublenses. As there are many small helper lenses, this reduces user interface clutter.

- A set of formats.
- Style information for containers, resources, properties, labels and values. This information is used to complement the same information that is stored in formats. So, in a way, a group is also a format.
- Additional content to be inserted before and/or after: resources, properties, labels and values. Again, this content is supplemental to similar format information.
- Primary classes: what classes are most important? This allows one to filter when displaying a list of instances so that less important ones are not shown. It also can be used to produce a set of classes to choose from when creating a new instance. See Sect. 17.2 for more information.

15.5 Lenses: Selecting trees of RDF data

A lens is a way for the expert to state what part of the data she wants the end user to look at. Thus, a lens encodes what properties to edit and in what order to display them. A lens has the following properties:

- Selectors: what instances can this lens be used with (Sect. 15.3)?
- `remm:peerRank` has a value between -9 and $+9$ to rank applicable lenses in conflict resolution (Sect. 15.5.3). 0 is the default, higher values are better. As an example, not all universal lenses are equally important, the peer rank ensures that more useful lenses are preferred during automatic selection.
- `fresnel:showProperties`: a list of *property descriptions* (see below for details). Each property description defines how to display a single property. When laying out the property data, we honor the order in which the properties are listed here. There are several pseudo-predicates (Sect. 15.5.1) for specifying sets of properties that should match a given description. Some of these sets are not even known in advance (such as `fresnel:allProperties` for all properties one encounters in a resource).
- `fresnel:use`: a format directly. Normally, formats function completely separate from lenses. This mechanism allows one to directly specify what format to use, without depending on a format selector to match.
- Lens purposes (Sect. 15.5.4): describe a lens. Fresnel supports `defaultLens` so that some lenses are preferred over others and `labelLens` for lenses that generate labels for resources. The latter is not supported by REMM.
- Projection purposes (Sect. 15.5.5): Lenses and properties can be enabled and disabled depending on the context in which they are used. For example, some properties can be shown in a form, but hidden in a table.

15.5.1 PropertyDescription

A property description provides further information about a specific property. It is a resource of type `fresnel:PropertyDescription` with various properties. As a shortcut, one can also use the predicate URI p directly (instead of a resource where p is specified via `fresnel:property`). A property description contains:

- `fresnel:property`: the URI of the property.
- `fresnel:use`: a format or a group. In the case of a format, it works like the "use" property of a lens. In the case of a group, the given group temporarily becomes the active group. Thus, this group is responsible for all data we encounter from here on down in the tree that we project from the RDF data.
- `fresnel:sublens`: a set of lenses that determines on a case-by-case basis how the data tree should continue underneath a property. For example, depending on the type of an object, we might either display just its URI or several of its properties. The selectors in the sublenses allow us to make this distinction.
- `remm:orderedSublenses`: has a collection domain and is needed when the order in which we check the sublenses matter. For example, one needs ordered sublenses in order to distinguish instances of `fresnel:PropertyDescription` from shortcuts: (1) If it is a resource (which might be a URI!) whose type is `fresnel:PropertyDescription`, then it is an instance. (2) Otherwise, if it is a URI, then it is a shortcut. (3) All other cases are ignored.
- `remm:propertyRange` assigns a property range. Instead of having to make this definition via OWL, it can be included with the lens and overrides schema information. Another way of looking at it is that the schema is adapted for editing needs. For example, if a lens is used for editing German content, one might require and create literals with the language tag "de".

Sections

In a projection, a *section* is a group of properties. It can optionally have a title. Some properties such as `rdfs:member` (for elements of an `rdf:Seq`) automatically create a new section, because they need different editing controls. One can also manually start a new section by adding a literal to `fresnel:showProperties` (where so far only resources were allowed). This leads to a new section being started whose title is the value of the literal. Sect. 15.8.2 demonstrates this way of starting sections.

Pseudo-properties

A few URIs have special meaning when specifying a property. They are not matched literally, but instead match sets of properties or are used to add special sections to a lens.

- `remm:globalProperties`: Add a section listing global properties whose domain is the current resource. Used for class lenses.
- `rdf:first`, `rdfs:member`, `fresnel:member`: The resource to be edited is a collection, a container, or either (see below). Create the corresponding sections when projecting.

- `fresnel:allProperties`: can be placed anywhere in the list and stands for *all* properties, even the ones we do not know about in advance.
  - `fresnel:hideProperties`: is a lens property and contains the URIs of properties that should be ignored by `fresnel:allProperties`.
- `remm:labelProperties`: Edit the label properties of a resource. They are usually the single property `rdfs:label`, but one can define custom labels (Sect. 17.3.2).

Property descriptions for assemblies (collections and containers)

The pseudo-predicate `rdf:first` is for editing collections (type `rdf:List`), `rdfs:member` is for editing containers (type `rdf:Seq`). `fresnel:member` is for editing either assembly, depending on the content of a resource.

Sometimes we use lenses to create new instances (e.g. when adding a new property with a sublens for the range). In that case, we must determine whether to create an empty collection or an empty container. That is, we must disambiguate `fresnel:member`. This happens according to the following rules.

- If there are lens selectors with a class name, we can examine the inheritance chain of that class to determine whether we have a collection lens, a container lens or both.
- Otherwise, there is no schema information to help us decide either way. If there are no additional properties, we assume `rdf:first` (as `rdf:List` properties cannot coexist with other properties). If there are additional properties, we assume `rdfs:member`.

If a lens is a collection lens, it pushes aside all other sections and properties and the collection becomes the only section. As mentioned above, this is due to the fact that collections do not have stable anchors. Sometimes they do not have their own anchor at all, if you consider `rdf:nil` an empty anchor.

15.5.2 ComputedPropertyDescription and node transfomers

Computed properties are a way of adding information to a projection that does not exist in RDF, but is computed using the projected resource. The result of the computation is always an RDF node so that it can be formatted using the normal mechanisms, as if the result had been directly parsed from RDF. Computed property descriptions have the following properties:

- `remm:computedPropertyLabel`: The label to use (computed properties have no predicate).
- `remm:nodeTransformer`: one of the following two URIs. Node transformers are functions $f : Resource \to Literal \cup Resource$
  - `remm:Subject`: maps a resource to itself. Useful for tables, where we might want to display the resources of the rows in a column.
  - `remm:TitleTags`: maps a resource to a literal containing the *title tags*, a string showing the tags and facets of the resource (Chap. 8).

15.5.3 Conflict resolution for lenses and formats

Whenever several lenses apply and the user does not choose one manually, we have to automatically determine which lens to use. This decision is made by finding the *most specific* lens, which depends on the specificity of its selectors. The same algorithm applies to formats as well, with minor modifications, so we describe both algorithms here. We successively try the following options until we have a matching lens or format:

1. (Format-only: the format attached to a lens via `fresnel:use`.)
2. Active group:
   (a) (Lens-only: most specific matching default lens of a group, `fresnel:purpose` = `fresnel:defaultLens`.)
   (b) The lens or format with the most specific (Sect. 15.3.3) matching selector in the active group.
   (c) If several lenses or formats are equally specific, the one with the highest `fresnel:peerRank` wins.

15.5.4 Lens purposes

Lens purposes come from Fresnel and are used less in REMM. Fresnel defines the following lens purposes, which are assigned to a lens by the property `fresnel:purpose`.

- defaultLens: Whenever looking for a matching lens inside a group, check lenses marked as default first.
- labelLens: When creating the display label of a resource (see Sect. 17.3.2), Fresnel considers all matching label lenses of the active group. Not supported by REMM.

15.5.5 Projection purposes

To make a lens more universal, we allow context-dependent hiding of lenses and properties. The context is a set of *projection purposes* which cover two areas (elements within each area are mutually exclusive):

- What is the output of the lens? `remm:tableLens`, `remm:formLens`, `remm:textLens`
- Is the projection for editing or for displaying? `remm:editLens`, `remm:displayLens`

The following two properties are used in lenses and property descriptions to adapt them to the projection purposes:

- `remm:showIfPurpose`: show the entity (lens or property) only if the given URIs intersect with the projection purposes. If the property is missing, we always show the entity.
- `remm:hideIfPurpose`: hide the entity if the given URIs intersect with the projection purposes.

15.5.6 Running example

The lens of the running example looks as follows:

```
:MovieLens rdf:type fresnel:Lens ;
    fresnel:classLensDomain ex:Movie ;
    fresnel:showProperties (
        [ a fresnel:PropertyDescription ;
          fresnel:property ex:title ;
          remm:propertyRange xsd:string ]
        [ a fresnel:PropertyDescription ;
          fresnel:property ex:genre ;
          remm:propertyRange ex:Genre ]
        [ a fresnel:PropertyDescription ;
          fresnel:property ex:description ;
          remm:propertyRange wikked:Markup ]
    ) .
```

This RDF specifies that resource `:MovieLens` is a lens that applies to instances of `ex:Movie` and that such instances have three properties `ex:title`, `ex:genre`, and `ex:description`. More examples of lenses encoded in RDF are given in Sect. 15.8.

15.6 Formats: Styling RDF data

Formats work in parallel with lenses to determine how the data stored in an RDF resource should be displayed. "How" means: What widget should be used to display the data? What text style should it be displayed in? Should additional text be appended before and after the data? Formats have the following categories of parameters (see also table 15.1).

- Selectors: Formats can be applied to both resources and properties, but some format mechanisms are specific to either resources or properties. Thus, the selectors of a format are in general a mix of resource and property selectors.

- Label configuration: Two things can be specified in standard Fresnel. First, whether the label should be shown at all. Second, whether a fixed text should be displayed as the label instead of the predicate.

- Styles: apply to elements in the abstract box model, namely resources, properties, labels and values. Styles depend on the HTML-specific Cascading Style Sheets standard and are currently ignored in the REMM. Ignored

- Presentation hint: How should a property value be displayed or edited? For example, even if the value is a URI node, it still might make sense to let the end user edit it inside a text field.

- Additional content: specifies text that should be prepended or appended to the content of resources, properties, labels or values. For example, one might want to start and end a list with square brackets and to separate the list elements with commas.

Format parameter	Group	Compound	Property	Value
Label and presentation hint				
fresnel:label			○	●
remm:displayHint			○	● (atomic)
remm:editHint			○	● (atomic)
Additional content				
fresnel:resourceFormat	○	●		
fresnel:propertyFormat	○		●	
fresnel:labelFormat	○		○	●
fresnel:valueFormat	○		○	●
Style				
fresnel:containerStyle	●			
fresnel:resourceStyle	○	●		
fresnel:propertyStyle	○		●	
fresnel:labelStyle	○		○	●
fresnel:valueStyle	○		○	●

Table 15.1: Inheritance of format parameters. Circles indicate where the parameters can occur (remember that groups also contain format parameters), filled circles indicate where they are used. Values to the right override (less specific) values to the left.

15.6.1 Inheritance of format parameters

The *projected format* is computed dynamically for each projection and a combination of several static formats contained in groups and lenses. The most specific of the static formats is called the *current format* and applies directly to the location where one is projecting. It is either explicitly specified via `fresnel:use`, has a matching selector, or has been omitted (resulting in an empty current format).

Tab. 15.1 shows how per-parameter inheritance is used to compute the projected format. Inheritance works along the projection tree: The group is seen as the root which contains nested compound projections that hold property projections which in turn contain either compound or atomic projections. In the table, each parameter has its own row. A circle indicates that the value influences inheritance, a filled circle indicates where the parameter is actually used. Columns to the right are more specific than columns to the left. For example, if a format is projected for the value and the labelFormat is specified in both the group and the property, the latter labelFormat is used. Other considerations:

- A format explicitly attached via `fresnel:use` overrides one with a matching selector which in turn overrides inherited parameters. Thus you can give defaults for properties via a group or a compound projection and override them with `fresnel:use` or a format that matches a specific property.
- Label and value format parameters are picked from the value, because this allows us to change the label depending on what sublens matched.
- Resource format, resource style: apply only to compound resource projections (not to atomic resource projections). Rationale: That way, literals and atomic resources are handled similarly.
- One always combines the current formats and never the projected formats of ancestors in the projection tree.

15.6.2 Additional content: Adding text

Additional content is specified for resources, properties, labels and values as `fresnel:contentBefore`, `fresnel:contentAfter`, `fresnel:contentFirst`, `fresnel:contentLast` and `fresnel:contentNoValue`. An atomic lens is only considered a value, not a resource. Additional content serves two purposes.

First, it can be used to add text data to the display boxes (Fig. 15.2). With "no value", the content *is* the box, while, with the other positions, it is placed before or after the box. In a sequence of boxes, the last `fresnel:contentAfter` is overridden by `fresnel:contentLast` (if it is there). Similarly, `fresnel:contentFirst` overrides the first `fresnel:contentBefore`. Note that each section has its own sequence of properties. When it comes to first and last values (and labels), we also start counting inside the section box and not inside the property box (the box model is shown in Fig. 15.2). Assemblies are considered a section with a single property.

Second, additional content is also used to translate a resource to text. If no additional content has been specified, a resource is translated to a sequence of values and labels without any space or separator. As this is undesirable, because it is hard to read, we supply a default that inserts an equal sign after the label and a space after the value. Sect. 15.8.4 gives an example of an additional content specification.

15.6.3 Presentation hints: What widgets to use?

Extension Presentation hints are a fundamental way of styling: they determine what graphical widget displays the relevant information. For example, instead of referring to a resource, a URI node could also be the address of a web site. Then, for editing, we would like to change it via a text field. During browsing, we want to have a clickable hyperlink. Note that we ignore the presentation hint for compound lenses, because there is only a single widget for compound projections. That is, presentation hints only make sense for atomic projections. There are two kinds of presentation hints: display hints and edit hints. The Fresnel property `fresnel:value` (whose range includes `fresnel:image`, `fresnel:externalLink`, and `fresnel:uri`) is translated to a display hint.

Display hints

Assigned to a format via `remm:displayHint`. The following classes exist:

- `remm:DisplayDateTime`. Display literals from the datatypes `xsd:date`, `xsd:datetime`, `xsd:time` as date and/or time. This usually results only in a slight change to make such data more readable for humans.
  - Property: `remm:dateTimeFormatPattern`. A pattern[3] that specifies how to format the date. Example: "yyyy-MM-dd".
- `remm:DisplayExternalLink`. Interpret the URI or literal as the URL of a web site and display it as a hyperlink. hyperlink to a web site.
  - Property: `remm:linkText` overrides the default link text.

[3] See http://joda-time.sourceforge.net/api-release/org/joda/time/format/DateTimeFormat.html

- `remm:DisplayImage`. Interpret the URI or literal as the URL of an image and display it as an embedded image.
- `remm:DisplayPlainText`. Display the URI or literal as plain text. How is determined via a display mode (see below).
- `remm:DisplayResourceLink`. Interpret the URI or literal as referring to a resource and link to that resource.
  - Property: `remm:linkText` overrides the default link text.
- `remm:DisplayWikiMarkup` interprets the literal as wiki markup and renders it accordingly.

Edit hints

Assigned to a format via `remm:editHint`. The following classes exist:

- `remm:EditCheckBox`: Edit boolean literals as check boxes.
- `remm:EditInstanceCollection`: Assign a value by choosing from a combo box with a list of instances. If instances are not enumerated by the class, they are looked up in RDF. Both literals and resources can be edited this way.
- `remm:EditResource`: Interpret a resource as referring to a resource and provide a suitable picker.
- `remm:EditTextBox`: Edit a literal or a resource with a text box.
- `remm:EditDateTime`: Edit a time-related literal such as an instance of `xsd:dateTime` with a date and/or time picker.

Presentation modes

Presentation modes determine how RDF nodes are translated to text.
(1) *Display modes* are assigned via the property `remm:displayMode` to instances of `remm:DisplayPlainText`, `remm:DisplayResourceLink`, and `remm:EditInstanceCollection` (where it determines how the entries in the combo box are printed).

- `remm:ResourceLabel`: Display the label of a resource.
- `remm:ResourceQName`: Display the qname of a resource.
- `remm:ResourceUri`: Display the raw URI of a resource.
- `remm:LiteralText`: Display just the text value of a literal.
- `remm:LiteralParsable`: Display a literal in a parsable way (Turtle syntax).

(2) *Edit modes* determine how things are edited. This means that the text to edit must be parsable. Edit modes are assigned via the property `remm:editMode` to instances of `remm:EditResource`, `remm:EditTextBox`.

- `remm:ResourceQName`: Edit the qname of a resource.

- `remm:ResourceUri`: Edit the raw URI of a resource.
- `remm:LiteralText`: Edit just the text of a literal.
- `remm:LiteralEditAnnotation`: Edit the text and the annotation (datatype or language tag) of a literal.
- `remm:LiteralDisplayAnnotation`: Edit the text of a literal, display its annotation.
- `remm:LiteralParsable`: Edit the literal in Turtle syntax.

15.6.4 Running example

We can now improve the property descriptions of the movie lens, so that better widgets are used. The defaults are acceptable for `ex:title` and `ex:description`. The range of `ex:title` is `xsd:string` and leads to a text box for editing and plain text for display. The range of `ex:description` is `wikked:Markup` and leads to editing via a text area for wiki markup and to displaying as rendered markup. One can provide improved editing of `ex:genre`, by using a combo box that lists all instances of `ex:Genre`. This is done by assigning a format via `fresnel:use` to the property description:

```
[ a fresnel:PropertyDescription ;
  fresnel:property ex:genre ;
  remm:propertyRange ex:Genre ;
  fresnel:use [
      a fresnel:Format ;
      remm:editHint [
          a remm:EditInstanceCollection
      ]
  ]
]
```

15.7 Documenting lenses

HYENA provides the following mechanisms to make lenses self-documenting:

- A lens documentation embedder: shows the domain of the lens, its properties and their ranges. If one of the properties has an `rdfs:comment`, it is displayed as its documentation.
- `remm:helpText`: Leads to a "note" link being shown during editing that displays the given text when clicked on.
- `remm:helpId`: Leads to a "help" link being show during editing that jumps to a section of the manual when clicked on.

The examples shown in Sect. 15.8 contain help texts.

15.8 Example lenses

The following examples are written in the RDF syntax *Turtle* [BBL08].

15.8.1 Lens for namespaces

The following is a lens for namespaces.

```
remm:Namespace a owl:Class ;
    remm:labelPredicate remm:prefixName ;
    rdfs:comment "For defining a namespace in RDF itself." .

remm:NamespaceLens a fresnel:Lens ;
    rdfs:label "Namespace lens" ;
    fresnel:group hyena:MetaGroup ;
    fresnel:classLensDomain remm:Namespace ;
    fresnel:showProperties (
        [ a fresnel:PropertyDescription ;
          fresnel:property remm:prefixName ;
          remm:propertyRange xsd:string ]
        [ a fresnel:PropertyDescription ;
          fresnel:property remm:uriref ;
          remm:propertyRange xsd:string ]
        [ a fresnel:PropertyDescription ;
          fresnel:property rdfs:comment ;
          remm:propertyRange xsd:string ]
    ) ;
    remm:helpText "URI: Should end with a slash or a hash" .
```

The lens begins with a declaration of its type `fresnel:Lens` and a human readable label. It is member of a group which is necessary for it to be visible. The lens applies to classes of type `remm:Namespace`. It shows the properties `remm:PrefixName` (the prefix of the namespace), `remm:uriref` (the URI of the namespace), and `rdfs:comment`. The last property of the lens definition provides a short help text.

15.8.2 A lens for classes

```
remm:OwlClassLens a fresnel:Lens ;
    fresnel:classLensDomain rdfs:Class ;
    fresnel:group hyena:MetaGroup ;
    fresnel:showProperties (
        [ a fresnel:PropertyDescription ;
          fresnel:property rdfs:comment ;
          remm:propertyRange xsd:string ]
        [ a fresnel:PropertyDescription ;
          fresnel:property rdfs:subClassOf ;
          fresnel:sublens [ # atomic
              a fresnel:Lens ;
              remm:lensDomain [
                  a remm:ClassSelector ;
                  remm:acceptClass rdfs:Class ;
                  remm:rejectClass owl:Restriction ]]]
        owl:unionOf
        "Restrictions"
        [ a fresnel:PropertyDescription ;
          fresnel:property rdfs:subClassOf ;
          fresnel:sublens remm:OwlRestrictionLens ]
```

```
            "REMM"
            [ a fresnel:PropertyDescription ;
              fresnel:property remm:labelPredicate ;
              remm:propertyRange remm:UriNode ]
            [ a fresnel:PropertyDescription ;
              fresnel:property remm:vocabularyCategory ;
              remm:propertyRange remm:VocabularyCategory ]
            "Global Properties"
            remm:globalProperties
        ) .

    remm:OwlRestrictionLens a fresnel:Lens ;
        fresnel:classLensDomain owl:Restriction ;
        fresnel:group hyena:MetaGroup ;
        fresnel:showProperties (
            owl:onProperty
            owl:allValuesFrom
            owl:minCardinality
            owl:maxCardinality
            owl:cardinality
        ) .
```

This lens for classes demonstrates three advanced features (underlined above): First, the lens domain is defined by a `remm:ClassSelector`. It explicitly rejects instances of `owl:Restriction`. This is necessary, because we want to handle restrictions later, but they are a subclass of `rdfs:Class`. Second, there are two section headings, “Restrictions” and “Global Properties”. These headings are different from additional content, because they actually start a new section with the consequence that restrictions are added separately from other class properties. Lastly, `remm:globalProperties` introduces a special class-specific section. This section collects all global properties whose domains match the anchor of the current projection. In the user interface, we allow the user to jump to them, but not to edit them (there is a separate lens that serves this purpose). Note that because `owl:Class` is a subclass of `rdfs:Class`, it can be edited by this lens, too.

15.8.3 A lens for lenses

The following RDF defines a lens that can be used for editing lenses.

```
remm:LensLens a fresnel:Lens ;
    rdfs:label "Lens lens" ;
    fresnel:group hyena:MetaGroup ;
    fresnel:classLensDomain fresnel:Lens ;
    hyena:helpId "lenses" ;
    fresnel:showProperties (
        rdfs:label
        [ a fresnel:PropertyDescription ;
          fresnel:property fresnel:classLensDomain ;
          remm:propertyRange rdfs:Class ]
        [ a fresnel:PropertyDescription ;
          fresnel:property fresnel:instanceLensDomain ;
          remm:propertyRange rdfs:Resource ]
        [ a fresnel:PropertyDescription ;
```

```
            fresnel:property remm:lensDomain ;
            fresnel:sublens remm:ClassSelectorLens ]
          [ a fresnel:PropertyDescription ;
            fresnel:property fresnel:group ;
            remm:propertyRange fresnel:Group ]
          [ a fresnel:PropertyDescription ;
            fresnel:property remm:helpText ;
            remm:propertyRange xsd:string ]
          [ a fresnel:PropertyDescription ;
            fresnel:property hyena:helpId ;
            remm:propertyRange xsd:string ]
          [ a fresnel:PropertyDescription ;
            fresnel:property fresnel:showProperties ;
            fresnel:sublens remm:ShowPropertiesSublens ]
    ) ;
    remm:helpText "Lenses must have a group to be publicly visible." .

remm:ShowPropertiesSublens a fresnel:Lens ;
    fresnel:classLensDomain rdf:List ;
    fresnel:showProperties (
        [ a fresnel:PropertyDescription ;
          fresnel:property rdf:first ;
          remm:orderedSublenses (
              remm:ComputedPropertyDescriptionLens
              remm:PropertyDescriptionLens
              [ # Resource has none of the property description types:
                # interpret as predicate URI
                a fresnel:Lens ; # atomic lens, no showProperties
                fresnel:classLensDomain remm:UriNode
              ]
          )
        ]
    ) .
# Omitted: remm:ClassSelectorLens
# Omitted: remm:PropertyDescriptionLens
# Omitted: remm:ComputedPropertyDescriptionLens
```

Demonstrated features: The sublenses of `rdf:first` are ordered, so that we can first check whether we recognize the type of the resource and otherwise interpret it as a predicate URI. The last of the sublenses has the lens domain `remm:UriNode`. This means that—as a last, catch-all case—we accept *any* URI node as a collection element. A lens domain of `rdfs:Property` would have been more elegant, but then we can only refer to properties that have an explicit type in the RDF repository.

15.8.4 Additional content

We demonstrate how a lens can be used to translate a resource to text with the following lens.

```
ex:TagsLens rdf:type fresnel:Lens ;
    fresnel:classLensDomain rdf:List ;
    fresnel:showProperties (
        [ rdf:type fresnel:PropertyDescription ;
          fresnel:property rdf:first ;
```

```
            fresnel:use [
                rdf:type fresnel:Format ;
                fresnel:propertyFormat [
                    fresnel:contentBefore "[" ;
                    fresnel:contentAfter "]"
                ] ;
                fresnel:valueFormat [
                    fresnel:contentAfter ", " ;
                    fresnel:contentLast ""
                ]
            ]
        ]
    ) .
```

This is a lens for collections. We want to display square brackets before and after all liste elements (even if there are not any) and commas between list elements. If we use this lens to generate text for the list

```
( "Private" "Todo" )
```

we get the result

```
[Private, Todo]
```

15.9 Discussion

This chapter explained how editing and displaying of RDF data can be specified using lenses. It is largely based on Fresnel, with the following improvements:

- **Improved automatic selection and adaptation of lenses:** An integer for peer ranking is used to prioritize lenses whose selectors are equally specific. Conditions operating on the current set of projection purposes (table, form, edit, display, ...) allows one to enable or disable lenses and properties in a context-sensitive fashion.
- **More powerful parsing:** The introduction of selectors as an explicit concept laid the foundation for more complex conditions of applicability. With multiple sublenses, these conditions direct how parsing continues. By ordering the sublenses, conditions become simpler to specify.
- **Improvements for displaying and editing:** Sections are used for grouping properties and improve readability for humans if a lens had many properties. Additional pseudo-properties enable the class lens to edit global properties whose range is the current class and any meta-data lens to edit the label properties (as they have been defined for the class of the instance to edit). Computed properties are a mechanism similar to inference. Presentation hints are used for selecting from a larger collection of widgets for displaying and editing, presentation modes allow one to configure how an RDF node is translated to text (when edited in a text field, shown in a combo box, etc.).
- **Self-documentation:** REMM collects the values of `rdfs:comment` to explain properties and a lens can be assigned a help text with an explanation or a help ID the refers to external help content. All of this is used to better document the lens and to support the user during editing.

Chapter 16

REMM editing: Specify and apply changes to resources

Contents

16.1 Overview

This chapter describes what data structures are necessary to hold the data during editing. It turns out that there are many things one has to consider so that editing RDF works as one would expect in a database-like application. Editing is a process that comprises four steps: First, the user picks a resource he would like to display. Second, an applicable lens is chosen, either automatically or manually by the user. Third, this lens is used to *project* the resource, producing, as a result, a *projection*. A projection contains the data of the resource, but in a format that mirrors the structure of the lens. Compared to traditional parsing, the lens can be considered the grammar and the projection the abstract syntax tree. Similarly, the projection will reflect how deeply a lens is nested and will always be a tree. The inner nodes of this tree are *compound* projections, the leaves are *atomic* projections. Fourth and last, after the user has finished editing the projection, it plays its second role: the changes it contains are written back to the resource; the projection is *applied* to the resource. This is the basic scenario for using a projection. When designing the data structure, we also have to consider that a projection has to work in *batch mode* (one should be able to apply the encoded changes to several resources at once). Additionally, lenses can be used for translating a resource to a text string (Sect. 15.6.2). For example, to generate labels for a whole class of resources.

This chapter first explains how a projection is structured, how it is created, how it is modified during editing and how it is applied. It concludes by giving an example.

16.2 The structure of a projection

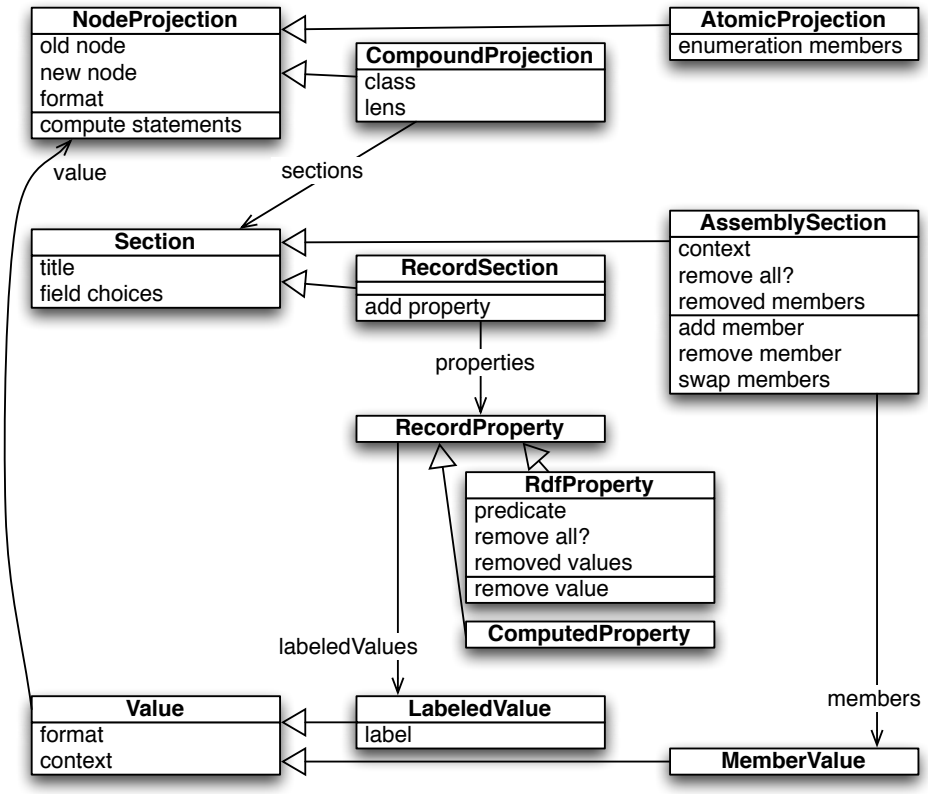


Figure 16.1: A UML class diagram for the projection data structures. A node projection is either atomic or compound. Compound projections contain sections which are either assembly sections or record sections. Each property of a record section either comes from RDF or is computed and contains a set of labeled values. Assembly sections contain member values. Note that even for the same record property, the labeled values can have different labels, because the label can be determined by the property value. The methods in the class diagram are the editing operations and are explained in Sect. 16.4.

Fig. 16.2 shows how the box model from Fig. 15.2 is reflected in projections: The root of a projection tree is always a *compound projection*, it holds *sections* that contain data that is edited in a similar way. For example, a projection of a resource could have two sections: one for container (`rdf:Seq`) data, one for normal properties. That is, it is both an assembly and a record. One can assign a section title, in the showProperties list of a lens, by putting a literal in front of the property description that starts the section. The projection classes have the following purposes:

- NodeProjection: is either compound or atomic. In both cases it contains a format and stores original and the current the value of the projected node. This is important when changing the RDF, because due to RDF statements being immutable,

we always have to remove the old statement and add a new statement.

- AtomicProjection: To avoid dependencies on schema and presentation data, we store the enumeration members with atomic projections that need them for editing (e.g., when they are displayed as a combo box widget).
- CompoundProjection: The lens is needed for adding new properties (whose values correspond to sublenses). The class is needed for adding a type to a newly created projection. To create atomic new properties, we need the property range and use the class as a fallback if there are no sublenses. Class and lens are both stored as resources, so that the projection does not have a dependency on schema or presentation data. Compound projections usually contain at least one section.
- Section: holds information that is related to the projected resource. Here we explain the assembly and the record section.
- AssemblySection: contains the elements of an ordered multi-set as encoded in an `rdf:List` or an `rdf:Seq`.
- RecordSection: contains properties which are a set of labeled values. A property can come from RDF or be computed.

Note how all constructs for data modeling that we have mentioned in Sect. 13.4 come up in projections: AtomicProjection is for atomic data and can contain enumeration members. CompoundProjection is for compound data and can contain a record and/or an assembly. The following sections explain how the projection data structure is actually used.

16.3 Creating a projection

Projecting a lens Creating a projection means that we get a set of statements (all the properties of a resource) and translate them to a CompoundProjection. To do that, we initially create an empty CompoundProjection instance and then iterate over all property descriptions in the lens. Each description consumes all statements that it matches and continues parsing by projecting the statement objects with its sublenses. While we usually ignore objects that do not match any sublens, we are more tolerant if there is only a single collection sublens, because collections are often untyped. Atomic sublenses lead to atomic projections and parsing stops. Otherwise, the process that we have just described starts again and new CompoundProjections are created.

Whenever the current section is not a fitting home for the data produced by the current property description or does not exist at all, a new one is created. For example, if a record section is current and a container property starts parsing, an assembly section must be created for its data. New sections can also be started by section titles, even if there is no direct necessity to do so. Even if a property description does not produce any data, an empty section is created for it, because there later need to be user interface elements for adding data to the section.

Property ranges and the schema of a compound projection Property ranges are used to create default values for properties and to display all fitting instances when letting the user choose a property value. The range can come from one of three sources: from the sublenses, from an explicitly assigned `remm:propertyRange`, or from the

class attribute of the compound projection. With sublenses, the range is the union of all classes used in selectors.

When filling in the class attribute of a compound projection, we face two problems: First, where should the schema information come from? From the resource that we are parsing (*dynamic schema*) or from the selectors of the lens (*static schema*)? Second, it greatly helps if we only have to deal with a single class, where in general, resources have several types and lenses have several selectors. REMM opts for picking the class of the most specific matching selector. That is, we use the static schema and have a single class.

Projecting a format So far, we have not explained how the format value of a projection is created. Table 15.1 shows how the format is created parameter by parameter: In each line, a filled circle indicates where the parameter value is stored, empty circles show that the value determines other values to the right unless it is overriden. An example: The value of `fresnel:propertyFormat` is stored in a property projection and a combination of the values from the group, the compound and the property format: If the parameter is specified in the property format, it is used, if not, we look at the compound projection and the currently active group.

In addition to that, we also track the position of the projection and consolidate the additional content parameters (Sect. 15.6.2): `fresnel:contentFirst` and `fresnel:contentLast` only have an effect for the first and last element, where they override `fresnel:contentBefore` and `fresnel:contentAfter`. Thus, we only keep these latter two parameters and change them during merging as appropriate. Note that finding the format *before* the merging involves selector matching and conflict resolution (see Sect. 15.5.3). The merged format that is stored in the projection is the result of merging the *unmerged* formats of the ancestors in the projection tree. Otherwise, format parameters would be propagated too far down the projection tree.

16.4 Editing a projection

This section explains the operations for editing a projection.

16.4.1 Adding a property to a record section

Providing the user with a good user experience when it comes to adding a property is surprisingly complex. Part of the problem is that we need to solve this problem in a generic way and consider all possible options:

- Predicate: What properties do we present the user with? What do we do if we have the pseudo-predicate `fresnel:allProperties`? What if the user wants to manually enter a predicate?
- Object: How does the user specify an object? Should she choose an existing node or create a new one? In the latter case, what type should it have? What if the object is a literal or an enumeration? Should she enter a value right away or do we provide a default and let her change that default later? How can we make sure that we always present an exhaustive list of options for the object?

There are obviously many solutions to this problem. We explain our preferred solution in more detail: We ask the user for how to create the predicate and the object at the

same time, instead of first letting him pick a predicate and then providing him with options for the object (in a more wizard-style fashion). All variations of this (object, predicate) request are encoded as so-called *property choices*. Property choices have to be created alongside the projection. They are a projection-specific list of all possibilities for adding a new property. Naturally, the end user will never see their structure, but rather a human-readable label. There are three kinds of property choices:

- typeset(predicate, mode, set of resource): Given a set of types, we either let the user pick an appropriately typed node or instantiate one for him.

  mode = `choose`: filter the choices by the given set of types. As an extended option, the user can manually enter a URI. The default for atomic sublenses.

  mode = `instantiate`: create a new blank node that has the specified types. The user can later rename it to a URI if she wishes so. The default for compound sublenses.

- constant(predicate, node): In the case of literals and enumerations (both of which are atomic), there is no real difference between choosing and instantiating. We give the property a default value that can later be changed.

- enterPredicate(choices): Used for the pseudo-property `fresnel:allProperties`. During parsing, it uses the same sublenses for any property it encounters. To create a property, we must first ask for a predicate before we can offer the choices derived from the sublenses. This results in a two-step process: First the user enters a predicate, then she decides what to do about the object. We prepare for the latter step by computing property choices with no predicate. After the user has chosen a predicate, these choices are copied and the predicate is filled in. Then the interaction with the user can continue like with normal property choices. In the future, we might include a default list derived from the (dynamic) schema. But there must always be the option to enter a predicate manually, because the schema might be incomplete. HYENA/Eclipse currently presents all instances of `rdf:Property`.

Property choices can be seen as instructions for creating new data. The user makes his choice, the projection is extended, then possibly edited and applied. The algorithm for creating the field choices is as follows. Creating the choices happens directly after creating the projection and is driven by property descriptions. That is, each property description produces a set of field choices. Obviously, their predicate is the predicate of the property description. What kind of choice it is, depends on the range of the property. We compute the construction set for the range (Sect. 14.3.3) and translate cinst elements to constants and cclass elements to typesets. For example, a literal class will produce a constant (say, the empty string for `xsd:string` or 0 for `xsd:integer`) and an abstract superclass will produce one typeset choice for each of its subclasses.

The property range is computed as follows. The selectors of the sublenses are translated to a set of property ranges. Each class selector is one property range, instance selectors are immediately turned into constant choices. If there are no sublenses, we use the schema to determine the range. If there is not even a schema, we assign two default field choices: one for creating an empty literal, one for choosing any RDF resource. The mode of the choices depends on whether or not a sublens is atomic. Atomic sublenses always produce atomic projections whose structure is not visible, thus choosing

existing instances is the only meaningful operation. For compound sublenses, we always instantiate a new projection, but the user can later swap the new instance for an existing one.

Note that adding a property to a record section actually means adding a labeled value to a property, because when "add property" is invoked on the section with a property choice, it looks for an `RdfProperty` with a fitting predicate

16.4.2 Other editing operations

- Compute statements: This operation is the foundation of applying changes to RDF (see below).
- Remove value from RDF property: To remove a value from an RDF property, one moves it from the labeled values to the removed values. Statement computation reacts accordingly.
- Add member to assembly section: Works the same as adding a property to a record section, only the predicate is never shown. Thus, member choices are the same data structure as property choices. A section can hold either one of them which is why they are called *field choices* in that class.
- Remove member from assembly section: To remove a member, it is moved to the removed members.
- Swap members: is used by the user interface when the user moves members via drag and drop.

16.5 Applying the projection: changing the data

The user always changes the projection and not directly the data. To actually perform the changes, she *applies* the projection. Before we can fully describe the application algorithm, we first have to explain another mode of projection application: Whereas until now, we have only talked about editing one resource at a time, *batch editing* is about editing *several* resources at once.

16.5.1 Batch editing

There are three kinds of manipulation operations that can be encoded in a projection: adding new data, removing existing data and changing existing data. Each of these operations faces new challenges with batch editing. Adding an atomic property value, such as a literal, to a set of resources is straightforward, but if the property value is compound (and thus not shared), we have to create a new instance for each resource. This kind of manipulation is encoded in a compound projection as the node being null. When removing data from a *single* resource, we are working with a definite set of data, so there are no problems. For multiple resources, we want to be able to remove both all properties with a given predicate and one given predicate-object combination. Especially the latter feature would be complicated to implement. REMM instead opts for a combination of the former feature and a simplified version of the latter: The projection is initially empty. Then one either adds atomic or compound data. To remove data, one invokes an explicit operation that removes all values of a

given property. This operation is "queued" and shows up as a user interface element, like the ones created when adding data. When going over all batch-processed resources, we create a projection for each one of them. Adding data is the same as in non-batch mode. Removing all properties means removing the projection trees of all values of a property. The "remove all" operation is encoded in class `RdfProperty` as the flag `removeAll`. That means that during batch editing, one can first remove all property values (if the flag is true) and then add new ones (like with non-batch projections).

Note that batch processing has two kinds of operations. On one hand, generic operations: For each batch-processed resource, new RDF is generated when adding a compound projection and different RDF is removed when removing all properties. On the other hand, RDF is directly specified when adding atomic projections or parsed compound projections.

16.5.2 Compute statements: what to remove, add and keep

If you look at Fig. 16.2, you see that the removed data is kept in or below the sections: the assembly section has a separate attribute for removed elements, the record section stores removed properties in the property projection. The "compute statements" algorithm produces three sets of statements: statements to remove, statements to add and unchanged statements. A few examples: null nodes of compound projections cause a new resource to be created for each invocation. An atomic projection with an old node and a new node produces one statement to be removed (with the old node) and one statement to be added (with the new node).

Afterwards, these three sets can be used to implement several operations: The change operation can be seen as a combination of addition and removal. When one changes a compound projection, the anchor does not change[1]. But changing an atomic projection, we have to first remove the old property and then add the new property. Thus, we store both the old and the new node with an atomic projection. To remove a projection (for example, during the change operation or when handling the "removed" data in an assembly section or a property projection), we compute the statements for the subtree, ignore the statements to be added and move the unchanged statements to the statements to be removed. Copying a projection means collecting all unchanged statements. We also offer a "duplication" operation which means that we have to post-process the unchanged statements and swap each existing resource with a freshly created one.

Immutable data Some named graphs in HYENA are immutable. When parsing a statement in such a graph, we don't allow editing. This is implemented in `AssemblySection` and `Value` via the `context` attribute that holds the graph URI and whether the graph is mutable (Fig. 16.2). The context data for `Value` is also used when expanding a collection from `rdf:nil` to one or more elements, to assign a graph URI to the newly created first element of the collection. Note that the context turns the `LabeledValue` into even more of a statement (minus the subject that is held by the compound projection).

When handling immutability during the computation of the statement sets, we need to split the set of unchanged statements into two disjoint sets: The set of mutable unchanged statements and the set of immutable unchanged statements. The algorithms for change, removal, and duplication are adapted correspondingly. For example, to

[1] The only exception are collections.

completely remove a projection from RDF, one removes all unchanged statements, but only if they are mutable.

RDF data versus projection-only data A projection is always a mix of data that has been newly created by the user and data that has been parsed from RDF. The most frequent case is that a parsed compound projection contains new data. But projection-only data can also contain RDF data if the user chooses to add a reference to an existing resource. The "old node" attribute of a node projection indicates whether a projection is parsed or created: if it is null, then the data does not exist in RDF.

Adding and removing types Compound lenses always make their `class` the type of newly created data. For removal, they have to find all equivalent types or subtypes of the `class` and remove them. This is due to our decision to use the static instead of the dynamic type (Sect. 16.3). Removing an atomic projection does not remove any types, as atomic projections are considered shared data.

Collections Collections are the only compound data structure where the anchor can change; they don't have stable identities. As a consequence, if there is a collection section in a compound projection, there can be no record or container sections. Furthermore, property projections must be ready for the fact that one of its compound values might change its anchor from a dedicated resource to `rdf:nil` or vice versa. Lastly, neither `rdf:nil` nor dedicated collection anchors have types; we neither add nor remove them.

It is a desirable feature that collections are handled like any other projection section. However, when creating an empty compound projection in RAM that does not yet exist in RDF, we create empty sections as place holders for properties or members to be added. Then we need to distinguish a collection that doesn't (yet) exist and a collection with no members, because the latter changes the compound projection: the anchor changes to `rdf:nil`. Contrarily, empty container sections and container sections with no members are the same. If a collection section is active, all other sections are cleared. If a non-collection section is active, the collection is reset to "does not exist". This way, the invariant is preserved that either the collection section or any other section is active.

To keep the computation of the changed and unchanged statements simple, we recreate the structure each time. That is, we remove the old structure which in general contains members that still exist and members that have been removed. Afterwards, we first create members that have been newly added, then we remove the statements of members that have been removed, and finally create a new structure that holds the members that still exist and the new members. To avoid recreating the structure when nothing has changed, we keep a flag informing us if the data has become "dirty". The algorithm for computing the statements of a container is similar.

Computed property projections Computed property projections are produced by computed property descriptions (Fig. 15.5.2). They are almost the same as RDF property projections, but are always immutable and don't store removed values or a `removeAll` flag. They do contain labeled values with usually atomic objects. Currently, one can only use one of a fixed set of *node transformers* (functions) to do the computation. In the future, JavaScript functions or rules could be additional ways of performing this task.

16.6 Example

For this section, the namespace ex is defined as http://example.com/. The RDF data contains a declaration of ex:Name as a datatype (for literals):

```
ex:Name a rdfs:Class ;
    rdfs:subClassOf rdfs:Datatype .
```

We use the following lens for humans and a sublens for a list of friends:

```
ex:HumanLens a fresnel:Lens ;
    fresnel:classLensDomain ex:Human ;
    fresnel:showProperties (
        [ a fresnel:PropertyDescription ;
          fresnel:property ex:name ;
          remm:propertyRange ex:Name ]
        [ a fresnel:PropertyDescription ;
          fresnel:property ex:friends ;
          fresnel:sublens ex:FriendsLens ]
    ) .

ex:FriendsLens a fresnel:Lens ;
    fresnel:classLensDomain rdf:List ;
    fresnel:showProperties ( rdf:first ) .
```

The first lens applies to humans and has the property ex:name with the range ex:Name and the property ex:friends whose range is a list. The second lens is used for editing the friend list. The fact that it edits lists is indicated by the pseudo-property rdf:first. The actual data to be projected is as follows.

```
ex:Adam a ex:Human ;
    ex:name "Adam"^^ex:Name ;
    ex:friends () .
```

Parsing this RDF data with ex:HumanLens results in the following projection, which is a data structure in RAM.

```
CompoundProjection: ex:Adam
    == RecordSection ==
    RdfProperty: predicate=ex:name
        LabeledValue: context=null
            AtomicProjection: "Adam"^^ex:Name
    RdfProperty: predicate=ex:friends
        LabeledValue: context=null
            CompoundProjection: rdf:nil
                == CollectionSection context=null ==
```

The root projection is compound and has the anchor ex:Adam it consists of a single section, a record section. This record section holds to properties. The first property ex:name has one value in the default graph (its context is null), an atomic projection of the literal "Adam"^^ex:Name. The second property ex:friends has a single value, a compound projection of an empty list. This compound projection comprises a single section for the collection.

Removing property values. One can remove property values which means that the LabeledValue is moved to the removed values. As a result, the labeled values become empty; empty properties are simply ignored.

Adding property values. The record section has the following field choices.

```
constant(ex:name, ""^^ex:Name)
constant(ex:friends, rdf:nil)
```

This means that there are two ways of adding a property to the projection. If one picks the first choice, the resulting projection looks as follows.

```
CompoundProjection: ex:Adam
    == RecordSection ==
    RdfProperty: predicate=ex:name
        LabeledValue: context=null
            AtomicProjection: "Adam"^^ex:Name
        LabeledValue: context=null
            AtomicProjection: ""^^ex:Name
    RdfProperty: predicate=ex:friends
        LabeledValue: context=null
            CompoundProjection: rdf:nil
                == CollectionSection context=null ==
```

Chapter 17

Configuration in RDF

Contents

17.1 Overview

This chapter explains several ways of configuring HYENA in RDF. The two most important topics are: First, how to designate *primary classes*. A primary class is preferred when filtering or creating instances. Second, how to name resources. The chapter concludes by summarizing all possible configuration data in a HYENA repository.

17.2 Designating primary classes

Not all available classes are equally interesting. This is why experts can designate more important classes as *primary*. Their special role comes into play in two situations: First, when displaying lists of resources, one can use the primary classes to filter it. Then only instances of those classes are shown. Second, when creating a new instance, only classes currently in use and primary classes are shown. Group-specific classes are added to a group as either list elements via `fresnel:primaryClasses` or as property values via `remm:primaryClass`. The latter property has the advantage of being compositional, where the former is a closed data structure. Global primary classes are instances of `rdfs:Class` (of which `owl:Class` is a subclass) that have a property `remm:vocabularyCategory` whose range is `remm:VocabularyCategory`. This allows us to group classes by purpose. Group-defined primary classes are part of a default category if they do not explicitly state one.

Example. The following two statements declare the two classes `wikked:Page` and `tmpl:Template` as primary. Note that the former is used by end users, while the

latter is only used by experts. The groups where the classes are primary have been chosen accordingly.

```
hyena:DataGroup remm:primaryClass wikked:Page .
hyena:MetaGroup remm:primaryClass tmpl:Template .
```

17.3 Naming resources

A resource has several names:

- The full URI: the lowest level of resource naming. Entered in HYENA in angle brackets: `<http://example.com/#Happiness>`
- The qualified name (qname): URIs can be abbreviated if they are part of a namespace (Sect. 2.3.1). Qnames are written without any brackets: `ex:Happiness`
- The label: the canonical way of adding a human readable label to a resource is via the `rdfs:label` property. Labels can only be used to display resources, not for entering them textually.
- The comment: is a longer description of a resource that is normally added via `rdfs:comment`.

17.3.1 Handling namespaces

Qnames exhibit two problems: First, while many RDF storage formats allow one to define namespaces, some don't (often if a relational database is involved). There also is no standard way of transferring such definitions. Second, the same prefix might be used by several namespaces, resulting in a clash. As an answer to the first problem, REMM reifies namespace definitions as resources of type `remm:Namespace` with the predicates `remm:prefixName` and `remm:uriref`. As a partial solution to the second problem, we regard qnames as a purely presentational mechanism and internally always use complete URIs. This solves internal disambiguities, but might still make explicit disambiguation necessary whenever the user enters a qname. Fresnel mentions qnames as having to be defined per group, but does not (to the author's knowledge) specify what vocabulary or mechanism should be used for doing so.

Example. The following RDF defines two namespaces `rdfs` and `gen`. The latter definition also contains an `rdfs:comment` that is shown as an explanation in the list of namespaces.

```
hyena:NamespaceRdf rdf:type remm:Namespace ;
    remm:prefixName "rdfs" ;
    remm:uriref "http://www.w3.org/2000/01/rdf-schema#" .
[] rdf:type remm:Namespace ;
    remm:prefixName "gen" ;
    remm:uriref "http://hypergraphs.de/generated#" ;
    rdfs:comment "Generated URIs" .
```

17.3.2 Labeling resources

The algorithm for labeling resources is as follows:

- If the resource has been assigned an explicit label via `rdfs:label`, that label is used.
- If the resource URI is part of a namespace, the local name is used (for example, `Resource` instead of `rdfs:Resource`). HYENA originally included the namespace prefix, but users found the colon in the middle of a name confusing and the local name still conveys all relevant information. The local name is URL-decoded. In some cases, when a user only provides a human readable label and HYENA needs a URI, it URL-encodes the label and adds it to a namespace URI to generate a URI. This works as a preliminary initial solution, because that makeshift URI can later be renamed to something cleaner.
- The raw URI is used.

Fine-tuning labels

Sometimes a vocabulary defines a property p that would make a good label and the resources that have that property do not define an `rdfs:label`. Then RDFS inference can be used to automatically add it: By making p a sub-property of `rdfs:label`, that property is added wherever p appears and has the same value.

The Simple Knowledge Organization System (SKOS) vocabulary provides[1] several ways of specifying labels: `skos:prefLabel` specifies the main label of a resource, `skos:altLabel` specifies alternative labels, and `skos:hiddenLabel` specifies invisible labels. This is currently not supported by HYENA.

Fresnel has the concept of a *label lens*, a lens that is used for labeling resources. Unfortunately, using lenses to translate a resource to text is complicated and not very performant. Thus, HYENA does not currently support this mechanism. In the future, a substitute (for projections) could be a property of atomic lenses that specifies how to translate a resource to text. This could be done via HYENA's template syntax (Chap. 11).

Example. The following RDF expresses the fact that the `remm:prefixName` of a namespace resources is a good label.

```
remm:prefixName rdfs:subPropertyOf rdfs:label .
```

17.4 Summary: all configuration data parsed from RDF

The following configuration data is parsed from RDF in HYENA:

- Repository settings: configure things such as the window title, the start page and the sidebar page.
- Brij settings: configures how RDF data is kept in sync with Java source code.

[1] http://www.w3.org/TR/skos-reference/#labels

- Public resources: Certain URIs can be declared public. *Public locations* appear in HYENA/Web in the "Go" menu, *public settings* appear in the "Settings" menu. An example for the former is a query for orphaned wiki pages, examples for the latter are the repository settings and the Brij settings.
- Short IDs: are assigned to resources so that the URL that links to them is cleaner. For example, if a resource has the qname `gen:rQeHbY2pyPJLsJzvuD`, its URL in HYENA/Web might be `http://localhost/hyena/proj/#gen:rQeHbY2pyPJLsJzvuD`, by assigning the short ID `todo` that URL becomes the much more readable `http://localhost/hyena/proj/#todo`.
- Sub-properties of `rdfs:label`: for resource labeling.
- Namespace definitions: for resource labeling.
- OWL, RDF: properties, classes (to provide schema support for REMM).
- REMM: groups, formats, lenses.
- Templates: are a more flexible display-only version of lenses.
- Facet definitions: are resources that define what facets to consider for faceted navigation.
- Tag predicates: are assigned on a per-tag basis to group tags (Sect. 7.4.6).

Part V

The extension framework

While HYENA's generic editing mechanisms such as lenses are very helpful with supporting new RDF vocabularies, not all new functionality can be supported in a generic manner. Accordingly, HYENA as a framework has been made as extensible as possible, so that it can be adapted to the editing requirements of new vocabularies. Even the core framework "eats its own dog food"; it is based on the same mechanisms that are available to extensions.

This part starts with giving an overview of HYENA's architecture, which is based on dependency injection (Sect. 18.2). Importing external data as RDF and exporting RDF as files is implemented as plugins. Several challenges must be overcome for the import and export translations and are described here. HYENA allows one to interpret the same resource from different angles: raw data, high-level interpretations, partial views, editing versus display, etc. These different interpretations must be kept in sync. HYENA supports multiple interpretations and their presentation via several constructs and maintains consistency by exchanging events between these constructs. HYENA projects can be synchronized between installations. As they comprise both files and RDF repositories, two different synchronization algorithms have to be implemented: On one hand, file synchronization is based on well-known approaches such as digests. On the other hand, RDF synchronization is implemented the simplest way possible and extends ideas from file synchronization to RDF. The granularity of file synchronization is the file, the granularity of RDF synchronization is the resource.

Creating a framework that involves RDF created new challenges such as handling multiple interpretations of the same resource. But it also simplified some issues such as configuration. In addition, HYENA strives for self-documentation and easy discoverability of services and provides several conventions to achieve these goals.

Chapter 18

Architecture: Hyena as an implementation framework

Contents

18.1 Overview

This chapter describes the foundations of the HYENA programming framework. The main architectural principle of the framework is *dependency injection*. Dependency injection has two major benefits: It allows the framework to scale to many interdependent components and it allows it to adapt to different scenarios by changing what components are in use and how they are connected. Dependency injection does this by letting components declaratively specify abstract dependencies, while a core injector object chooses concrete implementations, instantiates them, and connects them. The container API is HYENA's implementation of dependency injection. HYENA's architecture comprises two layers that communicate asynchronously: One or more *frontends* manage user interaction, while a single backend manages the data and synchronizes the change requests from the frontends. The different platforms pose a challenge for authoring help content: It has to be bundled with a web application and an Eclipse plugin and it should be available online, browsable as HTML and downloadable as a PDF file and ZIP archive of HTML files.

18.2 Dependency injection

Frameworks that consist of interdependent components face several challenges: Scaling up is difficult. If there are many components, finding out what components are

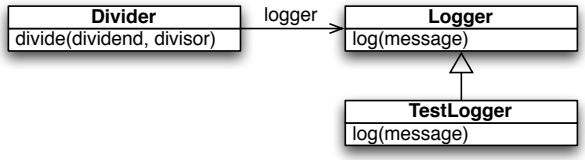


Figure 18.1: The component class `Divider` depends on the component class `Logger`. During unit testing, it should use the test version `TestLogger`.

available tends to be difficult. When accessing a component instance at runtime, one has to know "where to look" (find the right factory, the right global variable with a singleton, etc.). The more components are deployed, the harder it becomes to set them up properly; in the proper order, considering all dependencies, etc. Finally, adapting to different scenarios is challenging. The same abstract component can have several implementations. Depending on the scenario, different implementations might have to be picked, where each implementation needs to be set up differently. For example, HYENA's core components are used by both HYENA/Web and HYENA/Eclipse. Unit testing is another example. During unit testing, a component does not exist inside the complete framework, but is instead only connected to test versions of its direct dependencies. These test versions can just do nothing or perform some kind of logging or checking.

Dependency injection proposes two means to solve these problems: First, components state their dependencies on component interfaces declaratively and the system performs the wiring for them. Second, the *configurator* is a module-like mechanism that encapsulates scenarios and tells the system what concrete components implement the components interfaces.

18.2.1 Classes as components

Most frameworks implemented in class-based programming languages use classes in two ways:

- as components, which provide a service and usually exist as singleton instances in a given environment.
- as data holders, which encapsulate data and are used as parameters for components.

Fig. 18.1 gives an example: Classes `Divider` and `Logger` are components, whatever classes are used for dividend, divisor and message are data holders. This example illustrates that `Divider` and `Logger` will probably exist as singletons in a framework. To unit-test `Divider`, one checks that *something* is being logged if there is a problem such as division by zero. Thus, one would use a class `TestLogger` instead of the normal `Logger`. This class records in RAM when something has been logged, which allows the unit test to find out about it.

18.2.2 Managing components

Temporarily using `TestLogger` instead of `Logger` cannot be done in, say, Java if `Divider` instantiates `Logger` itself. It would also make it impossible to share

the `Logger` between several components. Thus, dependency injection uses the *container* as a registry for singleton components, similar to the singleton design pattern [GHJV95], but without the problems of using global variables. Thus, when `Divider` needs an instance of `Logger`, it asks its container for an instance. Depending on the scenario, the container maps the class `Logger` to an instance of `TestLogger` or the normal implementation. Components can be viewed as living inside the container (which explains its name).

18.2.3 Instantiating components

The container has a global perspective on the system, the component only locally knows what it depends on. To ensure that components are instantiated in the proper order and configured correctly, control over instantiation is taken away from it and given to the container. That is, there is an *inversion of control*, the `Divider` does not get to instantiate the `Logger`, the container does it for it.

To be able to do so, the container keeps a mapping from component interfaces to component implementation classes. Initially, the container is empty and then instantiates all implementation classes by asking each component what interfaces it depends on. If an interface already has an instantiated implementation inside the container, that implementation is handed to the component. Otherwise, the corresponding implementation must be instantiated, recursively. Obviously, this process can only start if the dependency relation forms a forest whose roots are components with no dependencies.

The simplest way to inject dependencies is *constructor dependency injection*: The constructor arguments state a component's dependencies and the container becomes a factory for instances. Thus, all arguments of a component C constructor are component interfaces. The container invokes the constructor after having either freshly instantiated or looked up all the components that C depends on.

18.2.4 Configurators

The life cycle of using a container is as follows: When starting up an application, one first creates an empty container. Then a set of configurators creates the container's mapping from component interfaces to component implementations. Configurators are similar to modules in programming languages. They are usually provided by the platform and ensure that the right component implementations are chosen. Finally, the container instantiates all of its components (the range of the mapping).

18.2.5 Other kinds of dependency injection

Two other kinds of dependency injection shall be briefly mentioned (for details, consult [Fow04]): *Setter dependency injection* use the container as a registry in exactly the same way as with constructor injection, but the constructor is nullary and all the setters of the new instance are invoked the same way that the constructor has been beforehand. Thus, handing in instances shifts from the constructor to the setters. *Interface dependency injection* is different from constructor and setter injection. Here, a class that needs to be injected implements an *injection interface* and during container configuration one maps injection interfaces to *injectors* that invoke the methods defined in the interface to set up a newly created instance.

While the way of dependency injection that we have seen so far was passive (a real inversion of control), you can also turn this idea inside out and use the so-called *service-*

locator pattern. The service locator is a registry that is used for actively retrieving dependencies. HYENA uses the container for both dependency injection and service location. The latter enables one to use the container for creating instances even after all components have been set up. To do so, the container itself can be requested as a dependency by a constructor. This mixed approach has worked well for HYENA. It is necessary, because components might have internal objects that have component dependencies, too. And the container can only inject dependencies into objects that it manages. A possible declarative solution would be to mark component fields whose values should be dependency-injected (similarly to setter injection), but often one wants to control exactly under what circumstances the instantiation happens and how often.

18.2.6 State of the art

Currently, there are a few dependency injection frameworks available. Some popular examples are: Java EE has introduced dependency injection with version 5.0. Spring has had dependency injection from the start and has recently lessened its reliance on external XML files by providing annotation-driven configuration.

Google Guice [gui08] is a lightweight dependency injection framework from Google that comes closest to what HYENA dependency injection is like. But, HYENA was created before Guice and thus, we are still using our code. Long-term, HYENA will probably migrate to Guice.

18.3 The HYENA container API

The container API is HYENA's implementation of dependency injection. Additionally, HYENA defines conventions for discovering components and for extending them. HYENA's life cycle management helps with initializing and cleaning up components.

18.3.1 Components and their roles

A component in HYENA can play zero or more roles at the same time. To designate itself as player of a role, a component implements an interface or extends an abstract class. The following roles are available:

- Service: provides functionality that can be accessed by other components. This role functions purely as a marker so that service providers can be easily looked up.

- Service contribution: Lightweight components can be placed in a container to contribute to a service. This role has two functions. On one hand, all service contribution classes (or interfaces) inherit from a marker interface and can thus be looked up. On the other hand, the service contribution role itself dictates the format of the contribution. A contribution is either directly data, performs a function (such as translating something) or a factory. An example of a factory is an inspector factory which is invoked with a resource and returns applicable inspectors.

- Life cycle participant: implementing one of these interfaces allow a container to be notified of container life cycle stages (startup, shutdown, etc.; see below).

Note how all of the above mechanisms can be controlled by a configurators and components. A configurator provides services by registering components that implement the service role. A configurator specifies a service it needs, by letting one of its components depend on an interface that none of its component implement. This interface dictates the required service. A configurator contributes to services by registering components that implement the service contribution role. Components signal the need to be informed of life cycle events by implementing the life cycle participant role.

18.3.2 Container life cycle

The container goes through several stages of which components can be notified:

- Startup: Components can be notified if a container has just been created, to perform one time only initialization tasks.
- Reloading: This stage has been introduced to allow components to reparse service contributions and configuration data in RDF (e.g. after it has been edited).
- Shutdown: The container is about to be destroyed. This is the time for components to perform clean up tasks.

If a component implements one of the following interfaces, it is automatically informed of life cycle stages by the container.

- ReloadModifier: notifies a component when the container is reloaded. As a reaction, a component may parse RDF or look for implementors of a ComponentContribution interface in the container. Via a sub-interface of ReloadModifier, a container can specify that it wants to be reloaded after another container (e.g. if it needs data that that component obtains during reloading). An alternate way of ensuring an order for reloading is another sub-interface where the component returns a reload priority. Components with lower priorities are guaranteed to be reloaded before components with higher priorities. The derivation update (Sect. 9.4) is also performed during reloading.
- ShutdownModifier, StartupModifier: are similar to ReloadModifier, but used for tasks that only need to be performed once, either at startup or when a container is shut down.

18.3.3 Examples of service contributions

The standard HYENA services accept the following contributions. Each one is an interface or an abstract superclass that is extended to make the contribution.

- Type constructors: are used to add support for new data types to Wikked (Sect. 10.3.4).
- Data importers: extend the Eclipse user interface with the ability to import new data.
- Embedders: make it possible to display a certain kind of data (as represented by a model piece, see Chap. 19) inside a wiki page.
- Model piece factories: derive representations in RAM by either parsing RDF or by transforming other representations in RAM.

- Publishers: implement ways of converting RDF data to external representations, for example wiki pages to printable HTML or feed resources to an RSS feed.
- Save interactions: implement additional actions that are necessary when saving some resources. For example, wiki page content needs to be added to the file-based version control system.
- HYENA libraries: are classes whose methods are added as commands to the Wikked markup language.
- Inspector factories: given a model piece, provide inspectors for editing it.
- RDF provider: the component implements a method that returns a Java resource path to an RDF file. The data in that file then becomes part of every HYENA RDF repository. This mechanism allows components to bring their own declarative RDF data.
- StatusProvider: allows a component to return a short status report in HTML. This status report is accessible via the user interface.

Note that one can also make a service contribution via RDF data (Chap. 17).

18.4 Core layer and GUI layer

Both HYENA/Web and HYENA/Eclipse have two layers:

- Core layer, backend: manages the data.
- Graphical user interface (GUI) layer, frontend: manages GUI interactions.

These two layers are necessary, because there are usually several frontends accessing a single backend: HYENA/Web has several web browsers that host the GUI layer, while the core layer is hosted by the server. HYENA/Eclipse has a single-threaded user interface as frontend and the HYENA engine as the backend. But there have already been experiments where distributed installations of HYENA/Eclipse were collaboratively editing the same data. This meant that the installations performed live synchronization. The synchronization layer could be viewed as a different kind of frontend, running in parallel with the graphical user interface.

18.4.1 Transferring data

Interactions between the core and the GUI layer usually happen asynchronously, results are returned via a continuation. A frequent pattern is that of model data being transferred between the layers: The core layer creates the model data and provides it with everything so that it can be used even when there is no direct access to the RDF repository. For example, a *labeled node* pairs an RDF node with its label, a text string, so that the node can both be displayed in a human readable way and be used in operations that change RDF data. The GUI layer displays the model data, changes it and sometimes sends pieces of it back to the core layer to change the RDF repository. From the above description, it follows that model data needs to be as modular as possible. Especially HYENA/Web often uses resources as references. For example, a projection does not refer directly to its lens, but only stores the resource. Thus, lenses can remain a server-only data structure.

18.5 Help content

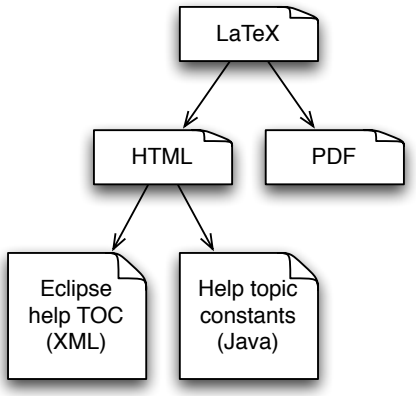


Figure 18.2: Help content. All of the help content is authored as LaTeX. From the LaTeX sources, HTML files and a PDF file a produced via standard tools. A custom process translates the HTML file to an Eclipse Table of Contents file and a Java source file with constants for context-sensitive help.

One of the challenges of help content is that it appears in several places and in different shapes:

- At the HYENA web site: The help content should be directly browsable as HTML and downloadable as a ZIP archive of HTML files and as a PDF file.
- HYENA/Web: should come bundled with the help content so that it is always available. Context-sensitive links in some of the views should jump to a description in the help content.
- HYENA/Eclipse: has the same requirements as HYENA/Web, except that there is a standard format for help content that consists of XML files with the table of contents and a set of HTML files with the actual content.

The solution is as follows. All of the content is authored as LaTeX source files, from which all other formats are generated. Standard tools are used to translate LaTeX to HTML files and a PDF file. The HTML is served by HYENA/Web as a sub-website `http://server/webapp/help/` A custom program parses the HTML to produce the Eclipse TOC files and a Java source file with enum constants for help topics. The former enables the help content to be bundled with HYENA/Eclipse. The latter is used by dialogs and views to specify where they are explained. Each constant contains the data that is necessary for HYENA/Web and HYENA/Eclipse to jump to the correct location in the bundled help content. A final deployment step generates a ZIP archive of the HTML files and copies all generated files to the appropriate places (Eclipse projects etc.).

18.6 Discussion

Dependency injection solves the challenges facing component-based frameworks that were outlined at the beginning of this chapter: Scaling to many components is not a

problem. Accessing one of many components at runtime is as simple as mentioning its interface in the constructor; there is no additional knowledge required to receive an instance. Setting up many components is performed by the container, which ensures that they are instantiated in the right order, receive all their dependencies, etc. The container also detects dependency cycles and displays descriptive warnings. Dependency injection also makes Java more dynamic and thus components are better decoupled: If one decides that a component needs one more service, one just adds the corresponding component to its constructor arguments. Usually, changing a constructor signature forces one to change all of its invocations. With dependency injection, this is not necessary. Adapting to different scenarios is handled by writing scenario-specific configurators that are used to initialize the container.

Additionally, HYENA provides conventions that help with several practical problems: By using a development environment to look up implementors, standard interfaces are used to find components that provide services and to find ways of contributing to them. Components implement life cycle interfaces to be informed of events such as startup and shutdown. Lastly, HYENA's way of mixing dependency injection and the service locator pattern brings the advantages of dependency injection to objects that are used internally by components.

Chapter 19

Multiple interpretations of resources

Contents

19.1 Overview

The same resource can be interpreted in several ways, be it at a low level, looking at the raw RDF, or at a high level, by giving more meaning to its content. Some interpretations can also be derived from others, e.g. if a resource contains a query that can be interpreted as either the query (when editing it) or as the query results. The derivation has to be updated when the interpretation it is derived from changes. Interpretations are a way of parsing RDF, to actually use them, they need to be *presented* to the user, by either embedding them inside a wiki page or by editing them with a graphical widget.

A *model piece* object holds a single interpretation, *multi-models* contain all model pieces of a resource. *Embedders* are responsible for translating model pieces to Wikked. *Inspectors* are graphical widgets for editing model pieces. Event handling ensures that model pieces, their derivations and their presentations are all kept in sync.

19.2 Requirements

RDF's basic building blocks are very fine-grained. On one hand that is the reason of its versatility. On the other hand, one usually needs to interpret the data before it can be used. Additionally, there are generally multiple possible interpretations of the same resource. One might use a wiki page editor to change its content, use the meta-data

lens to add the name of the author, or the "all properties" lens to show the complete resource.

This shows that the interpretations are means to an end, one wants to *present* such an interpretation, to either edit it via an inspector (a graphical widget) or to embed it inside a wiki page. The same interpretation can be presented in different ways. One (hypothetical) inspector can edit an interpretation of a resource as a wiki page in WYSIWYG mode, where text styles are directly visible. Another inspector can edit the same interpretation as plain text, where text styles show up as markup commands. Both interpretations and their presentation can come from multiple sources. At any time, there might be a new plugin that provides a new way of interpreting a resource or a new way of presenting an existing interpretation.

Some interpretations might be derived from others. For example, if a resource contains a SPARQL query, there will be one interpretation for editing the query and another interpretation for displaying the result of the query. If an interpretation is derived from another one, the former interpretation is called the derivation and the latter interpretation the base. Whenever a base changes, its derivations need to be updated, as well as their presentations. HYENA/Eclipse uses this mechanism to show a preview of a wiki page while editing it. Finally, as some interpretations are costly to create, one should only do so on demand, if they are needed by a presentation.

Speaking in terms of the model-view-controller pattern, what has been called an interpretation is analogous to a model, what has been called a presentation is analogous to a view.

19.3 Multi-models

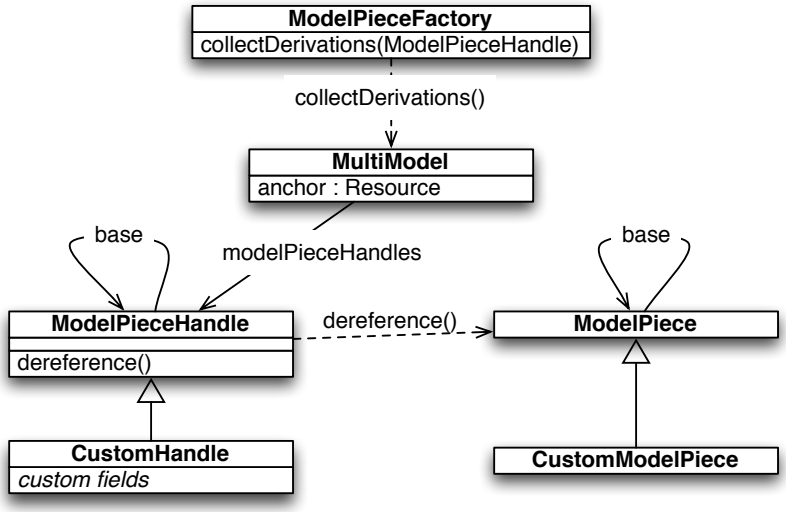


Figure 19.1: Model piece handles are produced by a factory and added to a multi-model. Each model piece handle points to at most one model piece handle it is derived from (its *base*). Model pieces also point to their base.

A single interpretation of a resource is called a *model piece* in HYENA. All model pieces of a resource are collected in a *multi-model*. A *model piece handle* (Fig. 19.1) is

a reference to a model piece. The model piece is created on demand, when the dereference() method is invoked for the first time, and cached afterwards. A handle must have enough descriptive data so that one can determine, for example, what inspectors can be used and what titles they should have. An example is the model piece handle for projections where the label of the lens is needed for the inspector title and the resource of the lens is needed to create the model piece.

Interpretations are provided via *model piece factories*. Each factory attaches derivations (as handles) to a model piece handle. The derivation process starts with a default model piece for a resource. Each newly created model piece handle is again offered for derivation to all factories.

19.4 Embedders

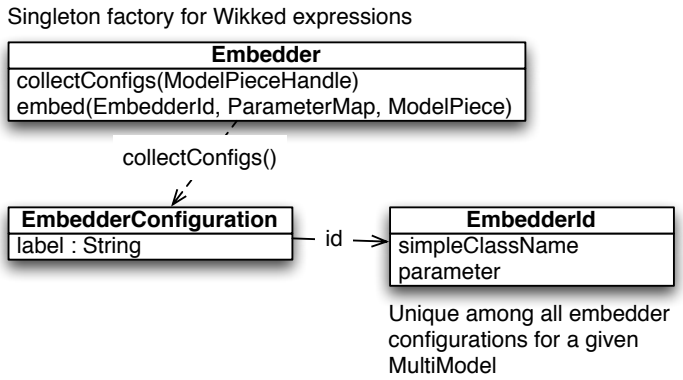


Figure 19.2: An embedder produces Wikked expressions and declares what it can embed by returning configurations.

An embedder (Fig. 19.2) presents a model piece inside a wiki page. It is a factory that translates a model piece to a Wikked expression. The same model piece can possibly be embedded in several different ways. An embedder declares its capabilities in this regard by returning zero or more *embedder configurations*, given a model piece handle. The most important part of such a configuration is the ID which uniquely identifies a way of embedding among all embedder configurations for all model piece handles of a single multi-model. Thus, an embedding command can precisely specify how to embed by providing a resource and an embedder ID (there is a simple serialization of an embedder ID as a text string). If only the resource is provided, the best embedder ID is picked according to an internal ranking system. An example of the same embedder producing configurations with different IDs are the projections of a resource. They all have the same embedder, but the `parameter` field of the ID contains the URI of the lens.

19.5 Inspectors

Inspectors (Fig. 19.3) are graphical widgets for editing model pieces. They are created with an indirection similar to model pieces: A factory takes a model piece handle and returns inspector configurations. These are used for constructing an inspector menu:

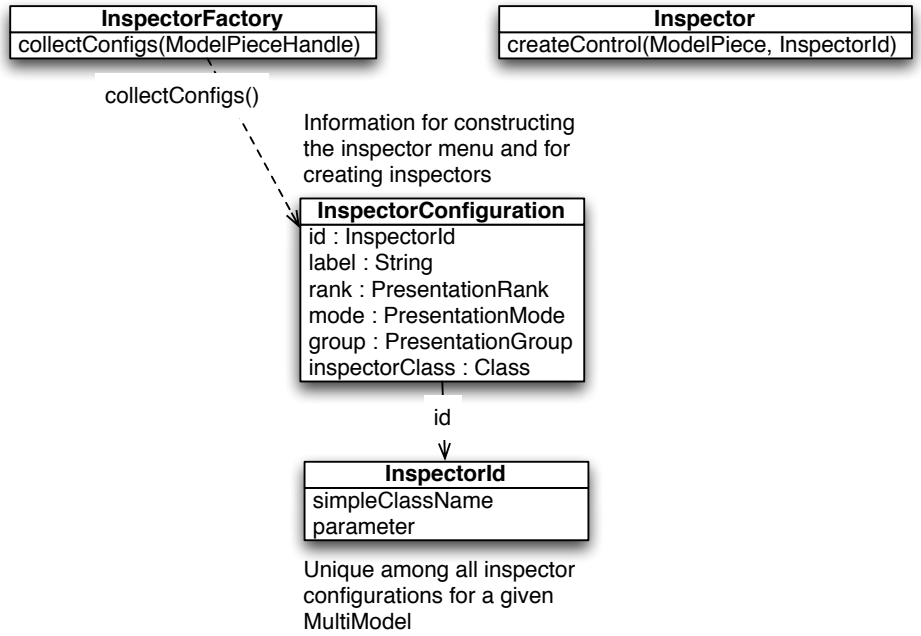


Figure 19.3: An inspector factory declares what can be graphically represented by returning configurations. An inspector is created from the class included in a configuration and initialized with a model piece and an ID.

A rank determines what inspectors to prefer by default. A mode indicates whether an inspector edits, displays, or configures. A group provides a title for a group of related inspectors (for example, all wiki page inspectors: edit page, display page, page history, etc.). The inspector is created on demand, using the inspector ID to distinguish different ways of inspecting, in a manner similar to embedder IDs.

19.6 Model piece methods

The methods of model pieces are concerned with saving and keeping derivations and presentations in sync. In many ways, RDF resources are treated like document files; they can be read into RAM, saved, etc.

Model pieces are always sent back and forth between the core layer and the GUI layer (Sect. 18.4), which, if they are hosted by different systems, are at least separate threads. They read from RDF and write to RDF at the core layer and are displayed and edited at the GUI layer. There are the following groups of methods in model pieces:

- Change propagation: a `recompute()` method is invoked any time a model piece has been changed. This informs the derivations that they have to adapt to they new content of the base.

- Read, write, reset the state: Reset is used for initialization, too, even if the model piece is never saved in RAM. This group of commands includes a method to remove the RDF at a given resource from the repository. By default, saving is a combination of removing the old content and writing the new content. Whenever

the model piece is changed by one of the methods in this group, the changes have to be propagated to the derivations.

- GUI update: After a change has been propagated through a derivation chain at the core layer, an event for GUI updating is sent through the same chain, but at the GUI layer. Inspectors react to this kind of event and update their display when they receive it.

Dirtiness is a boolean property of an entity. It indicates that the entity has changed and that the changes have not yet been "saved" to a more permanent storage; entity and storage are out of sync. Dirtiness serves two purposes. First, it enables some kind of "save" operation, which it only makes sense to execute if there are changes to be saved. Second, it indicates that action has to be taken when closing a dirty entity: One can warn about changes being lost or save automatically. There are two kinds of dirtiness related to model pieces:

- Model piece dirtiness: indicates that changes in the model piece have not yet been written to RDF. If a model piece is dirty, the operation to save it is enabled.
- Inspector dirtiness: indicates that changes in the inspector have not yet been written to the model piece. Most inspectors always immediately commit changes to the model piece. One example of an exception is wiki page editing where the content of the text field is only written to the model piece upon saving. A "preview" operation is used to perform an intermediate step: The edited text is written to the model piece, but not to RDF (yet). In HYENA/Eclipse, this allows one to use a second inspector for previewing the edited page. This inspector is shown side by side with the editing inspector and displays the rendered wiki page. It is updated whenever the model piece changes.

Chapter 20

Importing and exporting RDF

Contents

20.1 Overview

Especially when starting to work with HYENA, a problem is how to get data into it. Thus, HYENA can import several file formats: CSV (data entries encoded in plain text), web browser bookmarks, and BibTeX (bibliographical) entries. The main challenge when importing is to assign stable URIs to imported entities, so that the same data can be re-imported after it has changed.

The export capabilities of HYENA ensure that its data can be processed by dedicated applications: Wiki pages can be exported as LaTeX to start a document in that typesetting system or as HTML for printing. News and comment feeds can be exported as RSS, which allows one to read the entries in an RSS news reader, in a manner similar to emails in an email program. Data entries can be exported as JSON. Client-only data browsers run in a web browser and make this JSON accessible interactively, without the need for a special server. The main challenge of exporting was to support an additional way of authentication, to keep exported data safe: Web applications have cookie-based authentication, while external programs, such as RSS readers, that access files on the Web use HTTP authentication.

20.2 Importing

The main requirement when data to RDF is *change preservation*: Importing external data, changing the data externally and importing it again should add the changes made externally, but keep the changes made in RDF. Thanks to named graphs, there is a simple way of fulfilling this requirement, if the URIs of imported entities are stable: One reserves one named graph for each external data source. When importing, the named graph is cleared and all of the data is imported. Additions to the imported

Family name	#prop foaf:givenname	#prop tagging:tag	#type my:Contact
		#labelToResource	
Doe	Jane	friend	
Smithee	Alan	work	

Table 20.1: Example CSV table that is ready to be imported into RDF. The first column has the human readable title "Family name" that is converted to a property URI. The second and third column directly specify a property URI. The last column assigns the same type to each imported resource. In the third column, #labelToResource is used to look up tag URIs by label.

entities by the user are stored in a separate graph and are thus preserved. As the URIs are stable, the connection between the additions and the imported data does not break. The following sections describe how stable URIs are created for different data formats.

20.2.1 CSV

Data entries, such as lists of things, are some of the most useful information to be managed by CoIM. This information is often kept in spreadsheets and relational databases, both of which can export files in the *Comma-Separated Values* (CSV) file format. The translation consists of converting each row to a resource and the cells in that row to properties of that resource. The first row contains *column specifications* to indicate how to translate a cell to a property: What predicate to use, how to convert the cell value etc. Several challenges arise:

- Keeping property URIs stable: Each column must have either a human readable name or a URI as its title. In the former case, HYENA deterministically converts the name to a URI via URL-encoding. In the latter case, the URI is directly used for properties.
- Keeping entity URIs stable: The default is to combine the URI of the named graph and the row number to an entity URI. Alternatively, one use an extra column to manually assign a URI or a unique key (which will be URI-fied).
- Resource-valued properties: One can either directly enter URIs into cells or let HYENA look up the label of a resource to find its URI.
- Several values per cell: If an attribute of an entity can have several values, one often uses a single column and writes several values in a single cell. HYENA allows one to specify a regular expression to split this kind of text into multiple values before continuing processing.
- A fixed value for each row (such as the resource type): can be assigned via a column specification.

Tab. 20.1 shows an example CSV table.

20.2.2 Web browser bookmarks

HYENA imports web browser bookmarks in the standard Firefox format which is JSON-based. Firefox assigns each bookmark a stable integer ID, so generating stable URIs is

not a problem. Bookmark fields are translated using a Firefox-specific vocabulary that has been created as part of HYENA. The main challenge was handling tags properly, as Firefox encodes them with much redundancy. To allow users to only import part of the bookmarks (where privacy can be an issue), they can choose to only import bookmarks in a given folder or with a given tag.

20.2.3 BibTeX

The author has implemented an external tool called *Bibolizer* that converts BibTeX data to the Bibliographic Ontology[1]. BibTeX entries already have unique key which helps with generating stable URIs. It is also possible to override the automatic conversion of keys to URIs via a "uri" BibTeX field.

So the main challenge is to consolidate names, as the Bibliographic Ontology uses resources for the entities behind these names. Entities to consolidate are books, journals, persons (authors, editors), proceedings. One can provide Bibolizer with manual mappings in an external CSV file. Each row in that file is a mapping and contains the kind of name (book etc.), the name, and either a URI or a reference to another name. This kind of reference helps when the same name appears in different versions in a data set. When Bibolizer encounters a name, it uses the following strategy to convert it to a URI:

- If there is a manual mapping, that mapping is used.
- If the name is a journal, book, or proceedings, and a corresponding BibTeX entry is found, the URI of that entry is used.
- If a resource exists whose type is appropriate for the name and whose label is the name, then that resource is used.
- As a last resort, a new resource is created for the name and given the appropriate label so that it can be found again later.

20.3 Exporting

As the world around HYENA still thinks in files, this is the usual unit of export. HYENA/Eclipse writes exported data to files, HYENA/Web serves exported data as files. Because exported files will often not be accessed by someone who is logged into HYENA, HYENA/Web performs HTTP-based authentication.

20.3.1 LaTeX

Wiki pages can be exported as LaTeX which is useful when keeping notes as bullet lists that later should be integrated into, say, a paper written in LaTeX.

20.3.2 Printable HTML

Normally wiki pages are rendered as HTML with user interface elements (links to go to an embedded page etc.) and displayed inside the HYENA user interface. In order to have something that is easily printable, HYENA can export a wiki page as a stand-alone

[1] http://bibliontology.com/

printable HTML page. Links inside such a printable page do connect to other printable pages. This enables search engine support for future versions of HYENA, as search engines need access to linked HTML pages. Additionally, such support would entail an automatically generated site map that points to the stand-alone pages.

20.3.3 RSS

HYENA has feeds (sequences) of wiki pages to manage news and comments. These feeds can be embedded into a wiki page, so that one sees, for example, the four most recent news messages or a count of the comments that have been made. By exporting a feed as an RSS feed, one can use an external feed reader program with a comfortable email-like user interface. Most of these programs support HTTP authentication so that feeds can also be protected.

20.3.4 JSON

Exhibit[2] and Facetator[3] are two examples of frameworks that only need an HTML page and a file with JavaScript Object Notation (JSON) data to publish that data on the web. Compared with full-featured database web applications, one does not have editing capabilities. On the flip side, no special server is needed for deployment. The resulting web application is also very responsive, because no client-server communication is necessary (apart from the initial loading of the database into RAM, which is fast for small to medium-sized data sets). Exporting JSON from RDF is simple. HYENA uses JSON hashes where the keys are the RDF predicates and the values are the RDF objects. If a property has multiple values, they are exported as a JSON array.

[2] http://www.simile-widgets.org/exhibit/
[3] http://hypergraphs.de/facetator/

Chapter 21

Synchronizing files and RDF data

Contents

21.1 Overview

Synchronization is an important service of the HYENA platform. If there are two installations of HYENA that have the same project, the user can interactively reconcile the differences between them. Both the file system and the RDF repositories are synchronized. For synchronization, one compares a project on the local system (the *local project*) with a project on a remote server (the *remote project*). Then the user can choose how and if to propagate changes. Afterwards, both projects have the same content (except for changes the user explicitly chose to ignore).

Unless the user chooses not to propagate some of the changes, both projects have the same content afterwards. Synchronization starts by synchronizing the projects at file level and continues by synchronizing repositories at resource level.

21.2 Synchronizing files

File synchronization compares the files in the local project and the remote project. It has the following phases: *Scanning* shows all paths where the local file and the remote file are different or where a file only exists at one location. *Selecting* lets the user determine how and if the detected changes should be propagated. *Committing* propagates the changes In more detail, the algorithm is as follows.

- **Scanning:** A *summary* of a file contains the path, a digest, and a file size. First, the summaries for files of the remote project are created and sent to the local

system. Then local summaries are created. Local and remote summaries that have the same path are paired. If a pair has the same content (as determined by the digest), we discard it, leaving only the pairs with changes.

- **Selecting:** Two kinds of changes exist: Either there are files on both sides that are different, or only one side has a file. The latter case means that either a new file has been created or an existing file has been deleted. The user can then choose to keep the local state, to keep the remote state, or to do nothing and skip the pair.
- **Committing:** If a change is made then either the side to keep has a file and it is copied to the other side or it does not have a file, in which case the file on the other side is deleted.

21.2.1 Special files

The structure of a project is as follows.

`.hyena_project/`	hidden project data
`project.properties`	project properties
`local/`	data that is not synchronized
`sync_journal/`	keeps track of what has been synchronized
`thumbs/`	automatically generated thumb pictures
`page_history/`	histories of wiki pages
`_bookmark_cache/`	locally cached web sites
`_repositories/`	RDF files turned into websites by HYENA/Web
`_uploads/`	uploaded files

Two directories are exempt from file synchronization:

- `.hyena_project/local/`: the sync-journal is installation-specific and thumbs are always created on demand, by scaling an image file (that *is* synchronized).
- `_repositories/`: A repository file is only copied if it only exists on one side. Otherwise, HYENA always performs a synchronization with resource granularity (see below).

Finally, the page history (Sect. 10.5) is synchronized, but hidden from the user. This history grows monotonically[1] and is guaranteed not to have clashes, because the file names are globally unique and the files are immutable. Thus, synchronization can be performed automatically.

21.2.2 Keeping a journal

Putting a project online means that it is managed by a server that applies changes to it. Be it via synchronization, be it directly via the web application. If one assumes that file changes rarely clash (the same file being changed in two locations at once), one can virtually automate synchronization: After two parties have synchronized, they store the state of their version of the project at that time in a *journal*. Before the next synchronization that journal can be used to determine what has changed since the last synchronization. This information can be used for two purposes: First, the

[1] In principle, the history grows monotonically, in practice some pruning is performed to ensure that memory consumption does not explode.

change propagation can get a default setting so that we keep the change of the side that actually changed. Second, labeling each side of differing files can be improved: Without a journal, differing files could only be labeled as differing. With a journal, one of the sides is labeled as "changed", the other one (in most cases) as "unchanged". Without a journal, a file that only existed on one side was always labeled "new". With a journal, one side is labeled "unchanged", while the other side is labeled as "removed" (then the side with the existing file is unchanged) or "added" (then the side with the missing file is unchanged). A journal does not reduce the number of displayed changes, but it helps with handling them.

While each installation of HYENA keeps its own journal, it must keep a separate one for each synchronization partner. Thus, we need a way to globally identify HYENA installations. There are ways of doing this automatically (e. g. via a computer's MAC address), but none of them work for all scenarios (the installation might be copied to another system etc.). HYENA lets users manually assign *installation IDs*. Optionally, the user can let HYENA generate a unique ID and assign it. Not having assigned an installation ID switches the journal off, but one still can synchronize.

Critical for synchronization bookkeeping is when to do it and with what information, because the data to be synchronized constantly changes. To that end, HYENA uses the digests collected during scanning as a snapshot of the system. If the digests don't match during committing, the change is not performed. The data written to the journal comes from that snapshot, too[2]. Note that while pairs of files with the same content are not displayed to the user, they still have to be put into the journal. Otherwise, changing one of them leads to both of them being displayed as "new" during the next synchronization, as a result of the files being different and not existing in either journal.

21.3 Synchronizing RDF

The greatest challenge with synchronizing RDF is the existence of blank nodes. The literature provides two possible solutions: Use inverse functional properties to assign unique IDs to blank nodes [VG06] or use graph similarity [MTEBG07]. We use a very simple solution and completely ignore blank nodes, but provide the user with an operation that renames all blank nodes to generated URIs. In normal operation, HYENA avoids blank nodes as much as possible. For example, new resources are created as URIs that contain a generated globally unique identifier. This is in line with linked data principles that demand that every resource have a URI.

Furthermore, one has to decide on what should be the atomic units to be synchronized. Resources are an obvious choice, but collections should also be treated specially, because they are conceptually atomic, but stored as a chain of resources in RDF. Synchronizing them as such would be very cumbersome for users. SPARQL helps us with getting a list of relevant URIs:

- Non-collection resources: All resources that don't have a collection predicate (`rdf:first`, `rdf:rest`).
- Collection resources (first in a chain of collection elements): All resources that have a collection predicate but are not pointed to by `rdf:rest`.

[2] That means that the client sends the digests back to the server so that it can save them to its journal. It does not let the server compute its own digests, which might have changed since the last scan.

21.3.1 Synchronization algorithm

RDF synchronization uses two data structures for holding resources: A *holder* contains all the RDF data of a resource or collection, a *digest* contains only summary information. A digest is created by translating a resource or collection to a text string and by computing a digest for that text; using the same algorithm as for files. To ensure determinism of the translation to text, the statements of a resource have to be ordered. This can be done reliably, because there are no blank nodes involved.

1. Server: sends all digests and the installation ID.
2. Client: computes all holders, but only keeps those that are different from the remote data (as determined via the digests). If a resource only exists on the server, an empty holder is created. The resulting holders are annotated via the journal (what is unchanged, removed, new, different).
3. Client: The URIs of the resulting holders (which all differ from the server) are sent to the server.
4. Server: uses the URIs to compute holders (including empty ones) and annotates them via the journal. All RDF data inside the holders is sent so that the changes to be performed can be computed and offered to the user as a preview.
5. Client: constructs pairs of holders and presents them to the user.

Each of the pairs differs in some way and the user can choose to keep the local state, to keep the remote state, or to ignore. These operations are the same as with files. Additionally, RDF synchronization offers to merge, where both states are kept. Merging an assembly (a collection or a container) is not really possible, instead, HYENA concatenates assemblies. The local assembly comes first, then the remote assembly. The changes are computed on the client as four sets of RDF data:

- Add locally: If the remote data is kept or merged, it is added locally.
- Remove locally: If remote missing data is kept, it is removed locally.
- Add remotely: Upload local data if it is kept or merged.
- Remove remotely: Remove remote data if missing local data is kept.

21.4 Future research

The main topic of future research will be distributed version control (DVC) for RDF. With DVC, no central server is need and changes can be freely synchronized between peers. Furthermore, all copies of a main version control repository have all of the data of the main repository, they are complete clones. This means that the complete history is available, even when a copy cannot connect to the main repository. The choice of the main repository is arbitrary, if it fails, any copy can replace it.

Examples of file-based DVC are Git[3] and Pastwatch [YCM06]. Project files should probably be synchronized using such a system, instead of implementing a custom solution. Another challenge is how to handle long literals, such as the ones used for wiki

[3] http://git-scm.com/

page content. File-based version control systems save space for text files by only saving change increments. RDF DVCs should do the same for long literals. Note that if version control keeps prior versions, those take over the responsibilities of the journal. The current user interface can still be used for reconciling changes.

21.5 Discussion

HYENA's algorithm for synchronizing files uses the common practice of creating digests to detect differences. The user interface for change reconciliation and the idea of the journal was inspired by the open source Unison file synchronizer [PV04].

The actual contribution of this chapter is to apply these ideas to RDF synchronization. To make this application possible, several problems had to be solved. Blank nodes pose a problem, because they are often used in such a manner that nodes with different IDs express the same entity. HYENA's solution is to ignore blank nodes during synchronization, to offer to the user to rename all blank nodes to URIs, and to avoid blank nodes in normal operation. To avoid handling collections as a chain of resources (making them cumbersome to synchronize), they are treated as an atomic unit, just like plain resources. Digests for resources and collections are created by deterministically translating them to text and computing the digest for the text.

The result is a simple, but reliably algorithm for synchronizing RDF. By ignoring blank nodes, some complex issues were avoided. This may seem like an easy way out, but there is actually a growing consensus in the community that blank nodes need more permanent IDs.

Part VI

Related work

This part examines the state of the art in several areas that are related to HYENA: *Hypermedia and Hypertext* systems predate and inspired the world wide web. Yet, at the same time, they still exceed it in power. As traditional wikis can be considered a poor man's hypertext, it serves any wiki implementation well to consider literature in this area. *Annotating text* describes solutions for adding annotations to text, semantic or otherwise. HYENA's support for annotation is currently limited, so this section has been mainly added for the sake of completeness. *RDF editing* has been implemented by various programs, a section describes the most important ones. RDF editing is at the core of HYENA, so these programs are closely related to it. A section on *information managers* covers systems that are often based on semantic web technologies and are similar in principle to RDF editing, but take a higher-level view. HYENA is currently more generic and thus closer to being an RDF editor, but information managers provide important ideas about how to improve usability. *Semantic wikis* extend wikis with support for RDF data. HYENA is an RDF editor that has been extended with support for wiki markup. *Faceted navigation* is a way of efficiently navigating sets of objects with attributes. Some systems are described, including several ones that specifically apply faceted navigation to the semantic web. *Synchronization and versioning* examines how the problem of synchronizing RDF repositories has been solved in the literature. Most effort has been focused on how to handle blank nodes properly, because to synchronize blank nodes (whose IDs are not stable), one has to find isomorphic subgraphs; not an efficient operation.

What sets CoIM apart from its competition is its vision of a ubiquitous platform for information management with a unified model for the content. The most closely related work are the information organizer Haystack, the social semantic desktop Nepomuk,

and the social semantic wiki Kiwi. Haystack does not include a wiki and is a Desktop-only application without multi-user access. There is no unified model for its data with a level of abstraction higher than RDF. Nepomuk is a collection of Desktop-only applications whose common foundation does not abstract beyond RDF. Furthermore, generic RDF editing is not possible.

The closest relative of CoIM is Kiwi. This project, which has been started after CoIM, shares many of its goals: It aims to provide a reusable ubiquitous platform where information boundaries are broken down: System boundaries between applications and between different kinds of information are abolished. The objective is to not just link data, but to provide true integration. However, because CoIM and Kiwi have a different focus, their contributions are complementary and not in conflict: CoIM focuses on data, Kiwi focuses on knowledge. CoIM focuses on personal information management, Kiwi focuses on social information management. CoIM focuses on generic browsing and navigation, Kiwi focuses on a query language and reasoning. While both CoIM and Kiwi provide a content model with a higher level of abstraction than RDF, their approaches are different. Kiwi's *content items* are a combination of unstructured text (encoded as XML) and RDF and can be nested. CoIM also integrates unstructured text and RDF, but provides an encoding in pure RDF. An advantage of the Kiwi approach is that more structure is infused into text and can be used for more sophisticated querying. An advantage of the CoIM approach is that it stays closer to raw RDF, facilitating future integration with the linked data ecosystem.

Chapter 22

Hypermedia and Hypertext

Contents

22.1 Overview

This chapter looks at work related to HYENA in the fields of hypermedia and hypertext. *Conceptual Open Hypermedia* analyzes hypertext pages to dynamically add hyperlinks to related content. Relatedness is determined via knowledge bases. The next paper analyzes experiences with the *NoteCard* hypermedia system and posits seven issues for hypermedia systems. Even though the paper is quite old, these issues are still relevant today. *Aquanet* is a system that stores frame-based data and offers visual support for knowledge structuring tasks.

22.2 Conceptual Open Hypermedia (COHSE)

The *Conceptual Open Hypermedia Service* (COHSE, [JSB+08]) aims to solve disadvantages of linking in traditional hypertext:

- Static links: cannot adapt to users' needs, always reside at the link source, never at the target (backlinks).
- Ownership: of a document is needed to add links. External updates are not possible, neither are annotations.
- Legacy: link targets can become invalid.
- Binary links only: links always have a single target and no label.

COHSE is implemented as a browser that dynamically adds links to web pages, based on a knowledge base (KB). The browser does this by mapping terms found in the

document to lexicons in the KB, which contains the links. Links can be adapted to different audiences by using corresponding KBs. Links can have multiple targets and additional links are created by examining broader and narrower terms specified in the KB.

The requirements on a knowledge base are as follows:

- It should provide rich lexical support for matching terms in documents.
- It should represent relationships between concepts, especially generalizations and specializations.
- It should be flexible enough to accommodate data from many kinds of knowledge bases, so that external data can be easily imported.

The authors found out that ontologies are too precise for their applications and that the semantically looser *knowledge organization systems* (KOS) used in library and information science (thesauri, etc., see Chap. 4) are a better fit. The main difficulty is a different notion of generalization and specialization: In ontologies, it implies subsumption, whereas for humans, looser association is useful: Specializations of `accident` in the Medical Subject Headings knowledge base include kinds of accidents such as `traffic accidents` (which are subsumed by `accident`), but also `accident prevention` (which is not subsumed by `accident`, but still a narrower concept).

COSHSE initially tried to import external knowledge bases into OWL, but many had the above mentioned imprecision which was difficult to formalize in OWL. Instead, the RDF-based standard Simple Knowledge Organization System (SKOS) for defining knowledge organization systems was chosen. In SKOS, one defines concepts and for each concept a preferred label, alternate labels and a definition. Concepts are arranged into hierarchies by `broader` and `narrower` relations and linked via associative relations. SKOS fulfills the requirements for knowledge bases: its labels provide lexical support; `broader` and `narrower` specify generalization and specialization; its semantics is loose enough to accommodate a variety of external sources.

22.2.1 Comparing with HYENA

HYENA does offer dynamic linking via its faceted navigation, but does not currently offer related content for a wiki page.

22.3 NoteCard and issues for hypermedia systems

The paper "Reflections on NoteCards: Seven issues for the next generation of hypermedia systems" [Hal01] describes the classic hypermedia system *NoteCards* and issues for future systems that were discovered while implementing and using it. NoteCards was created in the mid 1980ies and contained a number of very modern features that make the conclusions drawn from their use in practice still relevant today.

The NoteCards information model is based on two main constructs: notecards and links. Notecards are containers of something editable: text, graphics, etc. A link is a directed connection between two notecards and has a label. The link shows up as an icon in the source card. There are two special kinds of notecards for aggregating information: a *browser* is a notecard with an editable diagram of a set of notecards and the links between them. A *filebox* is similar to a directory in a file system: It holds and categorizes other notecards and fileboxes, leading to a hierarchy of categories.

NoteCards requires that notecards be contained in at least one filebox. Navigation in NoteCards happens by following links (in content notecards, browsers and fileboxes) and by searching the notecards for content.

The author characterizes hypermedia systems along three dimensions: scope (between the extremes of personal private use to global distributed use), browsing versus authoring (read-only systems such as instructional delivery environments will have few or no editing tools when in use), target task domain (systems are never completely generic, they reflect the needs of their target audience). Using these dimensions to categorize NoteCards, it is designed for individuals and small work groups; it is mainly an authoring system; and it was originally designed as a tool for idea processing and authoring in a research environment.

The seven issues for hypermedia systems are as follows:

1. Search and query. Navigational access worked well for NoteCards in the following situations: if the network was small (50 to 250 cards) and few people (2 to 3 persons) were involved; if the task was about visual editing of a diagram (where mainly an overview browser was involved and the actual network played less of a role); if the network was an online interactive presentation (as these are mainly about navigation, about guiding a user through the content). On the flip side, if large unfamiliar, heterogeneously structured networks are involved, search is essential. The author distinguishes content search (for example, looking for text in text cards) and structure search (which additionally considers the structure of the network, such as "two connected cards that both contain the word 'hybrid'"). The former is well understood, while the latter still holds research challenges: query languages need to be designed, they need user-friendly ways of being expressed (visually, by example, etc.), and efficient evaluation algorithms need to be found (possibly for subsets of the language). In recent years, RDF query languages represent progress in this area, but many research questions remain: simple user interfaces; support for custom datatypes such as date intervals ("all entries that have been modified between March 1999 and September 2005") or location-based search ("all entries whose coordinates are close to my current location"); etc. The author envisions search to be used as a filtering mechanism, to only display part of a network.

2. Composites. NoteCards lacks a way of representing composite nodes; links always designate reference and never inclusion. Overview browsers can display a subnetwork, but it doesn't inherit its incoming and outgoing links. Document cards can be compiled from a tree of text fragments and images, but changes are not propagated automatically. One only can inspect the document at a single level; there is no way to zoom into parts of it. Questions to be answered for composites include how versioning of composites should work; whether a node can be included more than once; how composites inherit incoming links from their parts and whether incoming links point to the node itself or to the node as part of the composite (comparable to URL fragment IDs).

3. Virtual structures for dealing with changing information The static nature of linking hampers the ability of hypermedia systems to react to changes in the content. The "problem of premature organization" is encountered when one has to name and file and segment content when creating it, often before one has a firm grasp of the information space. When this space evolves, the initial decisions become obsolete. Possible solutions to this problem are *virtual structures* and *virtual links*. Virtual structures are

defined by search queries, by turning such a structure in a composite node, one could annotate and complement it. Virtual links specify their source extensionally (explicitly) and their destination intentionally (e.g. via a query).

4. Computation in hypermedia networks Normally, hypermedia systems passively store information. In contrast, expert systems contain inference engines that actively derive new information and add it to a system. One could add such an engine to a hypermedia system, making it even more similar to a frame-based system. Note that this issue is orthogonal to issue 7, extensibility and tailorability.

5. Versioning With a versioning system, users can look at a history of changes of a hypermedia network or explore several alternate versions (so-called *branches*) at the same time. This is an essential enabler for some applications, for example, in software engineering where versions play an important role in release management. Version control makes linking much more complicated, as one has to decide what is being linked: specific versions of nodes, newest versions, or the newest versions in a particular branch. Additionally, one has to decide for composites if and when new versions of their parts should lead to new versions for them. As a change often spans several nodes, *version sets* are the idea of collecting all individual node versions that are the result of that change. *Layers*, an alternative to version sets, encode changes so that one can explore their effects by applying layers or sets of layers to a base version. This is especially useful in a collaborative system.

6. Support for collaborative work The paper mentions several collaborative activities that are all well supported by hypermedia systems: creating annotations, maintaining multiple organizations of a single set of materials, transferring messages between asynchronous users. Collaboration builds on two kinds of foundations that need to be improved: the mechanics of multiuser access and the social interactions involved in editing. The former consists in extending the standard technologies for shared databases (transactions, concurrency control, and change notification, etc.) to hypermedia systems. For example: transactions in hypermedia systems tend to be long-running; locking must be very fine-grained; new ideas for conflict management need to be explored (e.g. create a version branch if editing changes conflict); interested parties need to be notified of changes, possibly even before a change happens. Support for social interactions involved in collaborative hypermedia editing centers on the idea of *mutual intelligibility*. The paper lists three kinds of collaborative activities: substantive activities (creating content), annotative activities (annotating the content with comments, critiques, questions, etc.), and procedural activities (discussions about how to edit the content, agreement on conventions, etc.). The last activity is not well supported in hypermedia systems and would involve functionality such as change histories, discussion forums, tracking individual contributions, recording usage conventions. To make content easier to understand for people it is shared with, a *rhetoric of hypermedia* should be developed. An already developed example are Landow's discourse techniques for link traversal: the *rhetoric of arrival* has to describe why the link should be followed, the *rhetoric of arrival* needs to describe how the node one has just arrived at relates to where the user departed from.

7. Extensibility and Tailorability The genericity of hypermedia systems is a blessing and a curse. It is a blessing, because such a system can be used for almost any

task. It is a curse, because it is never well suited for a specific task. The most frequent request from the NoteCards user community was a manual showing examples of the system being used for specific tasks. To remedy the problems of genericity, hypermedia systems are extensible and tailorable. At the time of the paper, this was supported via a programming interface, which precludes non-programmers for adapting such a system to their needs, even if the changes would be minor. The challenge is to provide minor modifications for non-experts, while keeping the powerful programming interface. GNU Emacs is cited as an example of a system that is built around an interpreter for a full-featured programming language. Its extensibility scales from simple calls into the system via a built-in command line to full-blown changes of its internals.

22.3.1 Comparing with HYENA

HYENA solves some of the issues raised by the paper; offering support for structured (meta-)data, in addition to hypertext, is essential for doing so. Other issues still remain to be solved and the relevancy of the paper even today is surprising and shows how little has changed in recent years.

1. Search and query. Having the structure of a HYENA repository explicit in RDF enables the kind of structural query that the paper mentions. Currently still missing is more efficient content search (which would have to be combined with SPARQL for structural search) and more sophisticated datatype-specific search. HYENA's resource sets represent the vision of using search for filtering a network.

2. Composites: HYENA has composites and distinguishes between linking and embedding. It currently does not let composites inherit incoming links of their parts.

3. Virtual structures: HYENA has resource sets that can be manifested and annotated, but only limited support for virtual linking (related content etc.). Premature segmentation is also a problem and can be fixed by providing operations for refactoring between in-resource structures and embedded content. Additionally, in-resource structure should be better supported (referencing, annotation).

4. Computation: HYENA does not currently use an inference engine. Two routes in this direction are possible: On one hand, simple inference for things such as transitive properties and for declaring to resources as equal. On the other hand, special-purpose inferencing for time, geolocation, aggregating values, etc. Currently, there is no obvious need for an advanced logic inference engine (e.g., one that supports full description logics).

5. Versioning: HYENA has a history for single wiki pages. Versioning for all RDF data would be nice, as would be branches. Ideally, one would have all the capabilities of a modern distributed version control system, which would lead to new interesting usability challenges.

6. Collaboration: HYENA only has very rudimentary conflict management. One possible feature for the future is live collaboration: Changes of several users are displayed directly, at the same time at for each user that makes a change. The idea of a rhetoric for hypertext is intriguing, but much more complex. If successful, information would be easier to understand for more people and over a possibly very long time.

7. Extensibility and tailorability: The programming framework of HYENA has always been very important (simple extensibility of the wiki markup language, invocation of GUI operations, etc.). In the future, this aspect will probably be expanded, even an evolution into a full-featured programming environment is conceivable, as software engineering and information management face many similar problems (manage complexity, be self-explanatory, easy exploration, etc.).

22.4 Aquanet: a hypertext tool to hold your knowledge in place

AquaNet [MHRJ91] has been developed based on the experiences with NoteCards. Its goal is to support *knowledge structuring tasks* that is the encoding and manipulation of knowledge for which traditional hypertext offers rudimentary support, but is not explicit enough. AquaNet stores its information in a frame-based database and focuses on flexible diagrammatic editing of this information. How data is represented as a diagram can be specified declaratively.

22.4.1 Comparing with HYENA

In contrast to AquaNet, HYENA has currently very little support for diagrams, but offers better integration of structured and unstructured data, and better search and navigation.

Chapter 23

Annotating text

Contents

23.1 Overview

This chapter surveys papers that investigate the semantic and non-semantic annotation of text. The first paper describes how annotation and navigation is handled in current semantic wikis. The *Unstructured Information Management Architecture* is a platform for unstructured information management solutions (text analysis, image recognition, etc.). *Open Calais* is a web service provided by Reuters that converts natural-language text to entities, facets, and events encoded in RDF.

23.2 Annotation and navigation in semantic wikis

The paper [ODM+06] defines a conceptual model where an annotation is a quadruple (subject, predicate, object, context) where the annotation itself is considered the object and is connected to the subject to be annotated via a predicate. The context of the annotation is also recorded. The paper distinguishes three levels of annotations in semantic wikis: layout (bold, italics, ...), structure (headings, hyperlinks, bullet lists, ...), and semantics (annotations that relate page elements to resources). Traditional wikis only provide the first two kinds of annotations while semantic wikis have all of them.

The authors describe a challenge for wikis: to distinguish a concept from the wiki page about it. For example, if a wiki page is about a book and located at a URI for that book, it is not clear what the value of property `dcterms:created` means: the creation date of the wiki page or the creation date of the book? SemperWiki, an implementation of the ideas in the paper, solves this problem by using resource names (such as `urn:w3.org`) for concepts and resource locators (such as

`http://wikibase/W3C`) for documents and uses the property `semper:about` to link a document to the concept it describes.

23.2.1 Annotations in wikis

dimension \ wiki	**SMW**	**SemperWiki**	**KiWi**	**HYENA**
attribution	current	current, any URI	current	current
granularity	page	page, doc. fragment	content item	page, loc. in page
repr. distinction	no	yes	no	no
terminology reuse	no	yes	yes	yes
object type	literal, page	literal, page, URI	lit., res.	lit., res.
context	no	no	yes	yes

Table 23.1: Annotation in the semantic wikis Semantic MediaWiki (SMW), SemperWiki, KiWi and HYENA.

The authors distinguish three levels of annotations in semantic wikis (Tab. 23.1):

Subject attribution What is the subject of the annotation? Semantic MediaWiki (SMW) and SemperWiki mostly author RDF via a special syntax inside wiki pages. SemperWiki can also create properties for resources other than the current page. KiWi and HYENA don't have that option, because RDF can be edited directly which avoids the problems of duplicating RDF data inside wiki pages.

Granularity How fine-grained are the annotated entities? SMW annotates pages, SemperWiki can additionally annotate a fragment of an external XML document by creating a statement whose subject points to the fragment via XPointer. But that is more a theoretical than a practical feature. KiWi and HYENA can in principle add RDF data anywhere. Furthermore, KiWi has a fine grained unit of information called *content item* and HYENA can embed small pages. HYENA can also place annotation links anywhere in a wiki page.

Representation distinction Can one distinguish a wiki page and the concept it describes? SemperWiki has a scheme where one can point from the page to the concept and annotate either one of them. KiWi and HYENA have support for editing RDF outside of a page and thus do not need direct support for this feature.

Terminology reuse Can an existing vocabulary be used for annotations?

Object type What kinds of objects can an annotation have?

Context KiWi and HYENA can record who made an annotation and when, but only for externally assigned data, not for annotations made from inside the wiki content.

23.2.2 Comparing with HYENA

The kind of RDF annotations that can be made in wiki markup by SemperWiki are treated as a separate concern in HYENA and handled by lenses. This approach is less brittle than duplicating the data (which HYENA avoids as much as possible). HYENA does not currently make the distinction between documents and concepts. Should the

need arise, a predicate similar to `semper:about` could be supported. Apart from that, one can always use lenses to edit data-only resources and refer to those resources from wiki pages.

Broad facet values are added externally to wiki pages and can hold context data (such as who created the annotation). Links, embeddings, and normal annotations are narrow and do not store context.

23.3 Unstructured Information Management Architecture (UIMA)

Quoting from [Apa08]:

> UIMA is an open, industrial-strength and extensible platform for creating, integrating and deploying unstructured information management solutions from powerful text or multi-model analysis and search components.

UIMA originated at IBM in 2005 and has become an Apache Software Foundation incubator project in October 2008.

Unstructured data is a large and quickly growing source of information, it includes text, voice and video. While unstructured data contains content that is both high-value and most current, that content is also buried in noise, semantics are implicit and search is inefficient. Unstructured information management applications can be characterized as analyzing large volumes of unstructured data to discover, organize, and deliver relevant knowledge. This usually means that concepts of interest (for example, named entities such as persons, organizations, locations, etc.) are detected in unstructured data. More advanced analyses can detect opinions, complaints, threats, or facts. Also detectable are relations such as located-in, finances, supports, purchases, etc. The results of these analyses are exported as a special data structure from which one can feed conventional data processing and search technologies such as search engines, databases, or data mining applications.

Unstructured information management employs a variety of analysis technologies, including: natural language processing, information retrieval, machine learning, automated reasoning, ontologies and knowledge sources (such as CYC, WordNet, etc.). They are usually developed independently, using different techniques, interfaces, and platforms. The *Unstructured Information Management Architecture* (UIMA) is a framework that integrates these analysis technologies and builds a bridge between unstructured information and structured information. UIMA is language-independent, currently supported languages are Java and C++. The following terms form the conceptual foundation of UIMA.

Analysis engines, annotators and results. Analysis engines perform analyses by invoking one or more annotators "inside" them and return annotations as so-called *analysis results*.

Representing analysis results in the Common Analysis Structure (CAS). The CAS is an object-based data structure where objects have types (organized in a single-inheritance hierarchy) and properties with values. For text documents, there is a predefined *annotation type* with a *begin* and an *end*. Other types such as *Organization*, *Phone Number*, *Noun Phrase* can inherit from this type. If at least two annotations

refer to the same entity, an *entity type* instance is introduced, it contains *occurrences*, that is, the annotations that refer to it.

Component descriptors. The building blocks of the UIMA framework are called *components*, annotators and analysis engines being two examples. Components consist of two parts: the component descriptor with component meta-data and the implementation, for example a Java program. The descriptor contains standard meta-data such as the component's name, author, version. But it also contains the required input CAS and the types that exist in the output CAS.

Aggregate analysis engines. While simple analysis engines (AE) only contain a single annotator, many contain a chain of annotators. An example would be an entity detector that contains the following annotators:

- language identifier
- tokenizer
- part of speech annotator
- shallow parser
- named entity annotator

Each annotator adds more data to the CAS. Users of the aggregate AE don't need to know about the internal structure, aggregate AE descriptors only note the input requirements and output types. The chain itself can be specified declaratively or programmatically.

Multimodal processing. For multiple modalities and other applications, UIMA supports the simultaneous analysis of multiple views of a document, in the form of *CAS views*. A CAS view holds both the data to be analyzed and the analysis results. For example, for speech analysis, one might first segment the audio data, then transcribe it to text, and finally detect named entities in the text. The first step annotated audio data, the second step produced text, the third step annotated text. Thus, two CAS views were involved, one for audio, another one for text. Applications of CAS views for text are transforming an HTML document to plain text or translating a text from one language to another.

Example application: semantic search. If a search engine not only indexes keywords, but also entities, one can perform searches such as "the *island* Java" and "the *mythical person* Paris" (avoiding obvious synonyms).

23.3.1 Comparing with HYENA

UIMA is completely complementary to RDF and HYENA. In fact, when HYENA's focus turns to features such as analyzing wiki content and making tag recommendations, it might very well use UIMA to implement them.

The CAS is a language-independent runtime data structure. Introducing RDF-specific ideas is not necessary, because it already supports arbitrary ontologies and export formats. In the future, one might define RDF-based exchange formats for the UIMA.

23.4 Open Calais

Open Calais is a web service maintained by Thomson Reuters. It converts a natural-language[1] text into RDF data that describes the meaning of the text: It identifies entities and the facts and events they are involved in. Open Calais explicitly sees itself as part of Linked Data on the Web (Chap. 3), integrates with a number of tools such as Drupal and Wordpress, and is used by online news services such as CBS Interactive / CNET and Huffington Post. Furthermore, they provide the a tool called Marmoset that inserts RDFa (Sect. 3.3.3) into an article. For example, given the input

> Angela Merkel has visited Barack Obama at the White House.

Then Open Calais infers the following data:

- Entities
  - Category "Organization": White House
  - Category "Person": Angela Merkel, Barack Obama
- Events & Facts
  - Generic Relations: visit, Angela Merkel, Barack Obama
  - Person Communication: Barack Obama, announced

23.4.1 Comparing with HYENA

Similar to UIMA, future versions of HYENA might rely on Open Calais for semantic analysis.

[1] Currently, English, French and Spanish are supported, with plans to support more languages.

Chapter 24

RDF editing

Contents

24.1 Overview

This chapter presents work related to HYENA that concerns RDF editing. The *Protégé OWL plugin* is an OWL-based editor that is based on the frame-based ontology editor Protégé. *Tabulator* is an RDF editor for distributed editing based on the linked data principles. *OntoWiki* is an RDF editor with wiki-like editing interface. Apart from database features, it also provides social features such as rating of resources and popularity tracking. The *TopBraid suite* is a family of products for ontology-supported RDF editing, visually specified data mashups, and semantic rich internet applications. *Annotation profiles* is a framework that turns declarative specifications into editors for metadata expressed in RDF.

24.2 The Protégé OWL plugin

The paper [HKM04] argues that RDF syntax is complicated and error-prone. As there are no comprehensive methodologies for authoring ontologies, tools play an even more important role. Tools should intelligently support authoring and debugging ontologies and turn-around should be quick to enable interactive ontology development. Two dimensions of scalability are important: First, ontologies can grow quite large. Second, authoring might be done collaboratively. Finally, tools should be extensible, because customization will be frequently necessary. They can provide a useful platform for experimenting with new technologies (such as new OWL reasoners).

Protégé. Protégé is an ontology editor and a knowledge acquisition system. OWL support has been added via a plugin. Protégé has a simple and flexible metamodel that is similar to UML's Meta-Object Framework: It provides modeling constructs and is meta-circular (modeled in itself). The metamodel represents ontologies as classes, properties (slots), property characteristics (facets and constraints), and instances. Protégé provides a Java API to query and manipulate its models. It can automatically create user interfaces for editing instances. Plugins can extend Protégé with new widgets for editing instances and larger sub-tools called *tabs*. Examples of tabs are performing queries, accessing data repositories, visualizing ontologies graphically and managing ontology versions. Furthermore, Protégé has a multi-user mode where several users can edit the same ontology at the same time, distributedly. Protégé's database for ontology storage is highly scalable.

The OWL plugin. Protégé is frame-based and OWL is description-logic-based. Thus, not everything can be mapped directly, but Protégé's flexible meta-classes help. For example, to represent disjoint class relationships, a new property :OWL-DISJOINT-CLASSES was added to Protégé's owl:Class metaclass. Other constructs required more work to bring them to Protégé. For example: OWL stores cardinality restrictions in anonymous superclasses, while Protégé stores them as *facets* with the property declarations. Thus, the OWL plugin automatically synchronizes facet values with restriction classes. As an alternative to the verbose OWL RDF syntax and the more compact OWL abstract syntax, Protégé opted for an even more concise syntax that is based on description logics. Classes are displayed in a hierarchy, like in frame-based systems. Class definition components are partitioned into "necessary&sufficient" ($\equiv$ equivalence), "necessary" ($\sqsubseteq$ subclass-of), and "inherited" (from subclasses).

Ontology Maintenance and Evolution. To ease ontology development, the OWL plugin borrows ideas from integrated programming environments, where, during compilation, one receives a list of errors and can use test cases to check programs. Test cases find a loose analogon in making class definitions very specific and then using description logic (DL) reasoners to reveal inconsistencies, hidden dependencies, redundancies, and misclassifications. So-called *ontology testing* also mimics test cases and compile buttons. Support for DL reasoners allows for *consistency checking* (whether a class could have instances) and *classification* (inferring a subsumption tree from the asserted definitions). To help users, the plugin displays both the manual assertions and what the reasoner has inferred. Asserted information is annotated with inferred information, where relevant.

24.2.1 Comparing with HYENA

Protégé is the platform of the OWL plugin in much the same way that Eclipse (and, to a lesser degree, GWT) is the platform for HYENA. The Protégé OWL plugin does not offer HYENA's integration of structured and unstructured data, but is similar in many regards when it comes to editing structured data. For example, it argues for extensibility via a general-purpose programming language and automatically generates from-based graphical user interfaces for editing. Furthermore, Protégé's data model is frame-based and similar to the CoIM RDF editing metamodel (REMM). An example of this similarity is Protégé supporting abstract classes (that cannot have instances). In contrast, OWL does not and the authors of the paper have defined an annotation prop-

erty for declaring OWL classes abstract. Furthermore, the plugin provides a simplified mode for users that shows properties together with their restrictions (as opposed to a list of generic conditions). REMM also prefers this way of displaying schema information.

24.3 Tabulator redux: writing into the semantic web

The paper [BLHL+08] argues that the semantic web lacks an element that made the web (of documents) so popular: instant gratification by seeing the results of one's work. Instead, semantic web technology is mainly used in the back end. Tabulator has originally been written as a browser of linked data (Chap. 3), without any domain-specific programming, except for a few common concepts such as time and geographical location. To turn the web into a read-write space, tabulator needed to provide editing in addition to browsing. When implementing editing of linked data, one faces a number of challenges: With the web of data, linked data introduces a new level of abstraction above the web of documents. This complicates the user interface, as breaking levels is sometimes necessary, for social reasons and for helpful error reports. Editing will involve a potentially unbounded vocabulary. Lastly, with multiple data sources, one faces the view update problem when trying to understand the effects of editing.

The above mentioned breaking of levels happens in two cases. First, with data coming from many sources (including inference), the user needs to be able to find out where it comes from to properly use it. Second, if an error happens, the error might be caused at document/repository level or below. For example, data might be missing in a document, a document might have syntax errors or network errors might prevent one from reading a document at all. Thus, Tabulator displays resources with a visual indicator of the status (unfetched, fetching, ok, error) of the document they are from. More information about the document and possible errors are given on hover. There are three ways of editing distributed RDF data: First, one document at a time is read and written. Second, several documents are accessible read-only, additions are written to a single document. Third, several documents are read simultaneously and a subset (possibly all) can be written to. Tabulator opted for the third alternative. It tightly couples vocabulary URIs and editable documents. For example, if the subject of a triple has the URI `doc#name` and `doc` is the URI of an editable document, the triple is written to that document.

There are two editing modes in Tabulator: Outline mode incrementally expands a tree starting at a given resource. Table mode constructs a table from a set of properties: All resources that have those properties are table rows, the properties themselves are table columns. Tabulator caches all displayed data locally and additionally automatically retrieves property and class definitions. *Panes* display additional information about a resource underneath the pane for the outline. For a class, a pane lists the instances. For a document, a pane shows network transactions and the content (which might be HTML or RDF).

During editing, Tabulator uses the cached RDF data to hide URIs as much as possible. To specify a URI, one types labels (assisted by auto-completion) or drags existing URIs. Predicates are selected from a list of predicates that have already been encountered during browsing, be it directly in RDF or in the schema.

Protocol-wise, Tabulator uses HTTP for reading, WebDAV for writing files, and SPARQL/Update for RDF changes. Plans for the future include an offline mode, real-time collaboration (currently one only sees changes when retrieving new information), and spreadsheet operations for the table mode.

24.3.1 Comparing with HYENA

Tabulator offers distributed editing of data, that is, multiple data sources to which one has read and write access. But it does not provide access control, nor offline capabilities. Tabulator has universal versions of HYENA's lens-defined form and table editing. For form-based editing, HYENA uses fixed definitions for what properties are shown and how deeply the tree of RDF data is expanded. Tabulator always shows all properties and expands the tree on demand, to arbitrary depth. For table-based editing, HYENA again pre-defines a set of properties via a lens (which also filters by type what resources to display). Tabulator lets one pick a set of properties and displays all resources in the table that have those properties.

Tabulator hides URIs whenever possible, even for entering data. HYENA uses URIs for entering data, but displays the labels otherwise, whenever possible. HYENA supports any kind of literal, Tabulator currently only supports plain literals.

24.4 OntoWiki

OntoWiki [ADR06] is not a true semantic wiki, but rather an RDF editor with wiki-like editing features. The paper observes the following problems with most current semantic wikis:

- Usability: wiki markup becomes complicated, because it needs to be able to express RDF statements.
- Redundancy: RDF data is both stored in a repository and encoded in wiki markup.
- Scalability: when changing statements, one has to modify both wiki markup and RDF data.

In OntoWiki, browsing happens in a three-part interface. A bar on the left offers starting points for browsing: available knowledge bases, a class hierarchy, and a text box for searching. The content area in the center displays the initial results as lists whose entries link to more detailed individual views. In addition to lists, sets of resources can also be displayed on a map or a calendar. A bar on the right holds tools and context information for the content area.

OntoWiki has widgets for statements, nodes, resources, literals, literals with a specific datatype (dates, HTML fragments, ...), and file uploading. These widgets can be styled and configured depending on the context where they appear. The context is the property, the datatype of a literal, the property and the class of its value, the knowledge base, the user, or the group of the user. Widgets are aggregated in three kinds of views: *Metadata views* edit data such as labels, and annotations that can be attached to any resource. *Instance views* use the OWL class of an instance to find widgets for editing it (optionally hiding properties that appear in the schema, but never in the repository). *Compound views* edit several resources at once, for example in a table. Social collaboration is supported by

- change tracking: Every change is tracked and one can subscribe via RSS/Atom or email to change notifications. Such notifications can be restricted to specific instances, to all instances of a given class, or to changes made by a specific user.
- commenting on statements: which is implemented via RDF reification.

- rating of resources: along several dimensions that can be defined per class. For example, publications could be rated according to originality, quality, and presentation.
- popularity tracking: Accesses to resources are tracked, allowing to determine how often they are visited.
- activity/provenance: The system records who contributed what.

Search is supported in the form of faceted browsing and full-text search. The latter can be refined by property (e.g., only search in values of `rdfs:label`) or type of the resource (e.g., only search in publications).

24.4.1 Comparing with HYENA

HYENA aggrees with the OntoWiki observation that duplicating RDF data in wiki pages is not a good idea. But it still considers wiki markup essential for integrated data management. OntoWiki does not have a construct that corresponds to HYENA's lenses. Even though its widgets are context-dependent, there is no way to offer alternatives for the same context. Lastly, HYENA's search is more powerful than OntoWiki's. For example, faceted navigation and keyword search can be used in parallel.

24.5 TopBraid suite

The TopBraid suite (TBS) is a family of products:

- TopBraid Composer: an Eclipse-based semantic modeling tool. External data sources such as spreadsheets, XML and UML can be integrated. It can also be used to develop semantic client/server applications based on TopBraid Ensemble and TopBraid Live.
- TopBraid Ensemble: allows one to deploy rich internet business applications that are dynamic and model-based. It supports data mashups (that is, data processing chains and visualizations of the results) via the SPARQLMotion visual scripting language.
- TopBraid Live: a semantic web application platform with a focus on service-orientation and dynamic, model-based, multi-user applications.

24.5.1 Comparing with HYENA

TBS focuses on enterprise applications and semantic applications. HYENA focuses on personal and collective information management and data modeling. TBS is a very powerful software package, much larger than HYENA. Similar to HYENA, it comes in both an Eclipse-based desktop version and a web version. HYENA's different focus affords it the following advantages:

- Standard-based form editing: HYENA extends the Fresnel display vocabulary standard where TBS is built with proprietary technology.
- Ajax-based: HYENA does not require any kind of browser plugin, TBS relies on Flex/Flash.

- Manual synchronization: TBS does automatic replication and partial caching, HYENA/Web manually synchronizes complete repositories, for total offline operation.
- Wiki: TBS is purely form-based, HYENA can integrate structured and unstructured data via its wiki functionality.

24.6 Annotation profiles

Annotation profiles [PENN07] are a framework that turns declarative specifications into editors for metadata expressed in RDF. The authors focus on end user friendliness and call their approach a *configurable annotation tool*. They contrast it with *generic annotation tools* which are not end user friendly and *fixed annotation tools* which are not as easily adaptable to new metadata vocabularies. Annotation profiles currently provide form-based editing and need two models to define an editor:

- The *graph pattern model* contains SPARQL-style triple patterns to parse and produce RDF. The subject of a pattern is always a variable. The predicate is always a URI. *Path triple patterns* have variables as objects while *constraint triple patterns* have (ground) nodes as objects. The patterns form a tree connected by variables. When parsing RDF, path patterns read objects from RDF, while constraint patterns prevent a resource from being parsed (e.g. when it does not have the right type). When editing RDF, constraint patterns are hidden from the user, but add information (such as types) to newly created data.
- The *form template model* is a tree that references the variables defined in the graph pattern model. It provides order, grouping, language-specific labels, and descriptions. Controls determine how values are edited. For example, a literal can be edited in a text field or chosen from a list of defaults. The form template model can also specify cardinalities for properties.

The *form model* is an instantiation of the form template model with data parsed from RDF via the graph pattern model. It is used to construct a graphical user interface.

24.6.1 Comparing with HYENA

Separating graph pattern and form template makes annotation profiles more complex, but also more universal. For example, annotation profiles could be used for SQL by using an SQL-compatible version of the graph pattern model. Annotation profiles do not have an exchange format, while HYENA's lenses can be delivered together with the RDF data.

Chapter 25

Information managers

Contents

25.1 Overview

This chapter covers programs that perform information management in a way that is related to HYENA. The paper on *information scraps* is a study on small units of information that are hard to capture digitally, either because it is not convenient or because dedicated tools are not available. The paper makes interesting observations about the nature of information scraps that are directly relevant for HYENA. *Lifestreams* is a novel way of doing document management. Its main modeling element is called a *stream of documents* and integrates several services while being conceptually simple: quick filing, dynamic classification, archiving, reminding, etc. *Haystack* is a platform for personal information management that has many similarities with HYENA. The *Social Semantic Desktop* extends the classic desktop environment with semantic technologies and data replication to integrate data between applications and between users or systems. *DBin* is a system for peer-to-peer sharing of RDF data in so-called *topic channels*. Optionally, custom user interfaces can be retrieved as part of such a channel.

25.2 Information scraps

Even though *information scraps* have a narrow definition [BVKKS08], they do appear frequently in daily life: They are about managing pieces of information when one does not want to encode them with explicit detail and the right tools are either non-existent, unavailable or too complicated to use. Information scraps fail us if we cannot retrieve

them later or if we cannot make sense of their content. Excluded from this definition is email used for communication, word processor documents with papers or full essays, contact information in the computer address book. In contrast, using a tool for information that it has not been created for often does lead to information scraps: an email with a todo, a word processor document with a list of contacts, etc. Examples: of 533 scraps studied for the paper, 92 were todos, 44 meeting notes, 38 name and contact information, 25 how-to guides. Distribution of scrap types is long-tailed: scraps of rare types cumulatively appear as often as scraps of common types. Information scraps play various roles. They are used as temporary storage, to supplement short-term memory; for archiving, to hold information for long periods of time; for cognitive support, to brainstorm, design, help with thinking; for reminding, to inform one's actions in the future. Information scraps exist, because the existing tools could not capture the information properly.

The nature of information scraps informed the tool created for managing them. Such as tool has to support

- Lightweight capture: Low time and effort barriers for creation.
- Mobility and availability: Migrate information to and from mobile devices. Different capture methods may be necessary in different situations.
- Visibility and reminding: Information has to appear at the right place at the right time.
- Flexible content and representation: Support multiple capture modalities. Record any kind of data, at any level of completeness. Be flexible about the schema.
- Flexible use and organization: Integrate existing tools. Make categorization easy and flexible. Allow to add meta-data.

The authors describe a few of the psychological foundations: *Channel factors* are "small but critical facilitators or barriers" to action. Thus: Even a small hindrance can prevent users from using a computer to capture an information scrap. Classification or filing is a cognitively difficult activity: If the cost of filing is perceived to be to high, one often creates an information scrap. The *flow state* is a meditation-like state of mind where the concentration is highest. Then unrelated thoughts and ideas may be unwanted which is why they should be written down quickly to keep the interrupt low (and the ability to resume the original activity high). Scraps can serve as *exosomatic memory* (a memory prosthesis) by later reminding us of our original thought. This is helped by a variety of cues that index into our memory: such as location, when and where a scrap was created, contextual information such as textual content, visual elements, implicit narratives around creation, etc.

Use of information scraps is driven by several factors. In a study, most workers' desks were piled rather than filed, but computers required to file rather than pile. The reason for the prevalence of paper in most office workplaces is its ease of annotation, its flexible navigation, its spatial reorientability and its support for collaboration. For files, people use few categories and often rely on location to find files in these categories. This suggests again the usefulness of location for human recall. Information scraps tend to have a short shelf life, are loosely filed or not at all, and are difficult to manage in large quantities. The paper cites a study where the semantics of file system folders changed continuously, to reflect the evolving understanding of the information. Only few items of high perceived value were filed, while the remainder

was left unorganized (3% of files, 41.6% of email, and 38.8% of bookmarks). Finally, with information scraps, one has to contend with information fragmentation, which happens across devices, applications and media. Negative consequences are file compartmentalization across tools, lack of ability to coordinate work activity between tools, inconsistent design vocabularies, and the inability to gather all data about a single topic or to effectively link such data.

The paper quotes studies of specific data types where information scrap phenomena showed up:

- Email: is used for many information scrap purposes. Emails are used as reminders, todos, drafts, etc. A study revealed that nearly a third of all archived e-mail was sent by the owners to themselves.

- Todos: A study found out that todos are created by expending as little effort as necessary and "only elaborated enough to provide a salient clue". A large number of separate tools (an average 11.25 per person) are being used for managing todos. Todos are usually not kept in a special-purpose application, but in many, often non-digital, locations: backs of hands, scraps of paper, unstructured text files, post-it notes.

- Calendaring tools: Example uses include keeping track of the week of the semester, diary entries with references to supporting material, reminders, logs of how time was spent, notes of prospective, but not finalized events.

- Photos: Cameraphones allow pictures to be used much more spontaneously. The pictures they take can be considered information scraps, as they are often hard to categorize with unclear semantics.

In the study undertaken by the authors, several kinds of tools were used. Electronic tools comprised e-mail (26.4%), text editors (16.8%) and word processors (6.4%). Physical tools comprised paper notebooks (37.2%) and post-it notes (23.7%). Tools needed to adapt to novel uses, to be general-purpose. Annotation and revision were common. The paper makes several observations regarding the information scrap life cycle:

- Capture: creation needs to be quick. "When time and effort were at such a premium, the fastest tool would often win out." Even a few clicks made a difference. "Even when data was implicitly structureable, such as with potential calendar events, participants chose the faster, structureless route of recording a scrap". Three major sources of information during capture are directly authored material (representing an intentional effort to record information), automatically archived material (external source, e.g. email), copy-pasted material (when only parts of external data are interesting).

- Transfer: Moving the information scrap from one medium to another after it has been captured occurs for three major reasons: First, transcription to fill in incomplete details and to make more appropriate for archiving or for consumption by others. Second, information was only in a preliminary state and necessitated a specialized tool for further processing. Third, mobility: Carrying post-its to another room, sending information via email so that it is available on the road or at home.

- Organization: Users had different criteria for grouping. Prevalent were grouping by time of creation and grouping by type or purpose. Users reported difficulty filing information scraps accurately and created a "miscellaneous" group for scraps that were difficult to categorize.
- Reference, retrieval, and recall: only few scraps were referenced regularly. One group of scraps (to-do list, post-it notes) was referenced frequently until it lost its usefulness and was either archived or thrown away. A second group of scraps was archived immediately, without a period of active reference.

As an anecdote, the paper describes that while information scraps were ubiquitous, many participants were slightly ashamed of having them and thought they ran counter the ideal of being organized.

25.2.1 Comparing with HYENA

HYENA is already well equipped to handle information scraps: New resources do not need to be given a name or a category. Some meta-data such as date of creation is added automatically. Batch operations help with managing large quantities of scraps. HYENA's wiki syntax, automatic operations for URLs (retrieval of the page title, local caching), and efficient tagging, help with quick capture.

25.3 Lifestreams

Lifestreams [FG96] were created to correct some of the shortcomings of document management when used for personal information management. Even today, document management is still the dominant way of information management, both conceptually and as a user interface metaphor. Some of the shortcomings are:

1. Naming a file and choosing a storage location is unneeded overhead.
2. Directories are inadequate as a classification mechanism. Classification should be dynamic and multi-dimensional (more than one "directory" a file can be in).
3. Archiving should be automatic. Often users completely remove files to avoid clutter. Putting them away in an organized (and retrievable) way is difficult.
4. Summarizing, compressing, visualizing groups of documents is important and should scale.
5. Computers should make "reminding" convenient. The goal should be to make calendars active (send an email etc.) and integrate them into the system.
6. Personal data should be accessible anywhere and compatibility should be automatic.

Lifestreams correct these shortcomings by having streams of documents as a storage model. Time is the basic ordering principle and split into past, present, and future. The past is used for archiving. The present holds what the user is currently working on. The future is about reminding: documents that have to be worked on in the future, emails that have to be sent, upcoming events, etc. This time-centric approach leads to temporal correlation of documents, something that is often prohibited by directories. Operations on documents are mostly generic and include "new" for creating a

new unnamed document, without having to specify a location; “clone” for duplicating documents; “freeze” for making documents read-only; “transfer” for distributing documents, sending emails etc.; and “print” for printing documents.

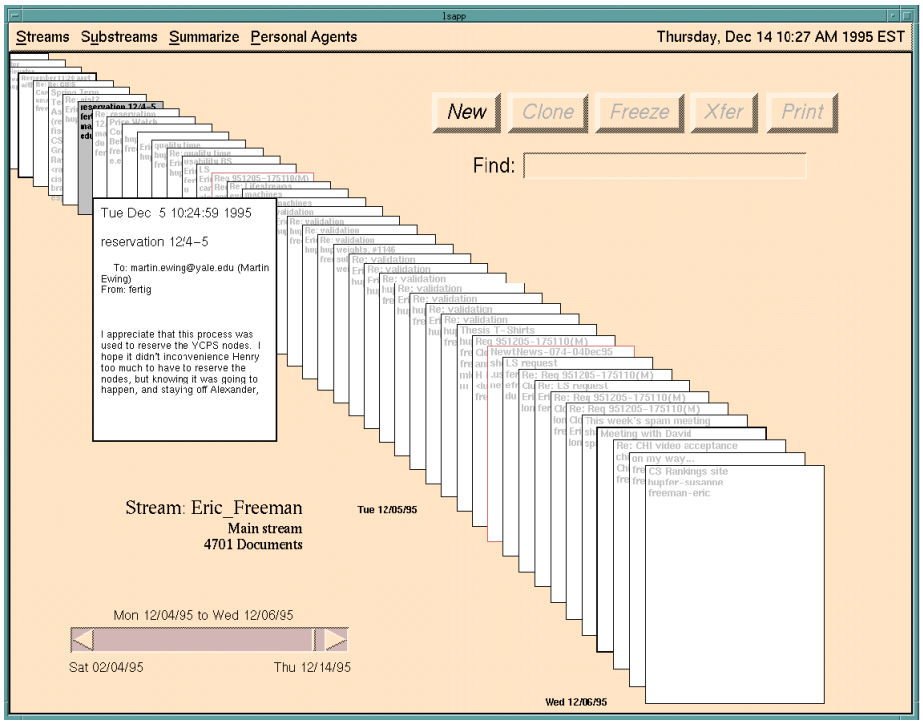


Figure 25.1: The graphical user interface of the Lifestreams application.

Streams of documents are displayed as a diagonal stack of thumbnails (Fig. 25.1) with visual clues for properties such as “not yet read” or “writable”. A full-text “find” operation allows one to create a dynamic *substream* of the current stream. A substream only comprises those documents of its superstream that contain the search text. It is again a stream and can be recursively filtered into substreams. A *summary* of a stream is a dynamic document that summarizes one aspect of that stream. For example, “by document size” shows a table with document sizes. An *agent* is like a plugin for lifestreams and provides new application-specific functionality. Support for several applications shows how universal the lifestreams model is:

- Email: is sent via the “transfer” operation. Automatic sending in the future is done by place the email creation date in the future. The future is thus a convenient metaphor for automation.
- Phone call records: the agent-supplied “make a phone call” creates a phone call record. A calling list can be created by finding and summarizing.
- Stock portfolio management: Each stock is a document. Summaries provide lists and graphs.
- Bookmark management: A daemon watches web browsers and automatically adds new bookmarks to a stream (as a URL document). These documents can be easily transfered to other users, via the “transfer” operation.

25.3.1 Comparing with HYENA

The main contribution of lifestreams is the concept of streams and how they are used. In contrast, the fine-grained documents and meta-data find a superior replacement in RDF resources. HYENA's resource sets are directly inspired by streams and offer a few enhancements. Using a future creation date for reminding is an intriguing idea, but problematic, because one might still want to know when one has created the reminder. HYENA's solution is to be multi-dimensional about time and separate the time of creation and the time when something is due. Time-related searches can easily be refined to any of these dimensions. Similarly, using the temporal concept "now" to denote data one is currently working on is problematic, association by tag can be used instead, again in a multi-dimensional manner.

Making time the dominant criterion for ordering and retrieval is very clever. In the future, location can play a similarly important role, because computers are increasingly capable of geolocation, so that this information can be automatically added to resources. Archiving is not yet completely solved in HYENA. The editing history for wiki pages are a first step, in the future full support for distributed versioning of RDF will have to be added. Lastly, Lifestreams always needs a server, it does not support offline operation the way HYENA does.

25.4 Haystack

Haystack [QHK03] is a platform that enables users to manage their information in a customized fashion. In this vein, Haystack solves a number of problems that traditional approaches for desktop information management have. In the traditional setting, applications are isolated information islands. The paper gives several examples where a user is in one application and wants to find related information in another application, but finds it impossible to do so. Similarly, explicitly linking information is impossible. To bridge the application gap, Haystack proposes the obvious solution: Store everything in the same repository. The authors argue that RDF offers the necessary flexibility and power to do so. Humans often seek information by *orienteering*, by first loosely finding the general "area" where the information might be and then zooming in on it by a series of associative steps. RDF's relational nature supports this kind of associative navigation. On the desktop, there usually is much external data. Haystack uses pluggable *extractors* to import file directory hierarchies, documents in various formats, music and ID3 tags, email (through an IMAP or POP3 interface), Bibtex files, LDAP data, photographs, RSS feeds, and instant messages.

To display the data in the repository, Haystack offers a variety of mechanisms. The system manages a set of application-specific *views* that have a nested rendering strategy: The view renders much internal information itself. Whenever an embedded resource needs to be displayed, the view asks the system for a view that fits the current data and display constraints. Views are defined in RDF via *view prescriptions* that state how to divide the display region up, what to display where and with what graphical widgets. Parameters for view prescriptions are passed down dynamically and include available space, colors, and fonts sizes. Selection criteria for view prescriptions are the type of a resource and the available space (thumbnail versus full picture). *Lenses* are used for aspects of resources that cross-cut types. They mainly specify a set of properties and can be applied to any resource; if a resource does not have any of those properties, nothing is displayed. Sets of lenses can either be displayed in a view to dis-

play information about the currently selected resource or they can be used to customize a table view for a collection of resources. Furthermore, lenses are context-sensitive to the currently active tasks. For example, a "recommended categories" lens is only active if the user is performing the task "organizing information". A "help" lens would only be shown if a help mode is active.

Being able to switch between different views for the same data is especially useful for collections where Haystack provides several options: row layout (the usual table widget, configurable via lenses), calendar (places collection members chronologically), graphical (as a graph diagram, with boxes and arrows), "last resort" (just shows all properties of the collection resources). Collections can also be turned into menus. These menus are easy to configure, because the user just edits the collection. They appear either as a popup menu when clicking on the name of the collection somewhere or as task menu for task-specific operations and items.

Program functionality is made available to users in a tangible way via *operations*. They are defined in RDF and appear in context menus when they are applicable in the current context. Furthermore, the user can create new operations from existing ones by pre-filling arguments[1]. Thus, given the operation sendEmail(data, receiver), if pre-filling the receiver as "Joe", results in a new operation that emails data to Joe. The target of the context menu is filled in as the first argument. If there are more arguments, a mask for entering them is added to the Haystack window. Views can also choose an operation to perform if something is dragged on them.

Haystack supports orienteering by giving users a fitting starting point ("in the neighborhood" of the relevant information) and then offering associative links to continue. This kind of navigation is complemented by text search. Haystack does not strive to make its search interface general-purpose. Instead, developers are expected to pre-package queries as operations for end users. Haystack applies information retrieval techniques to RDF by turning RDF resources into pseudo text documents. The techniques used with these "documents" are similarity search (finding similar resources) and query refinement (suggesting ways for continuing search).

25.4.1 Comparing with HYENA

HYENA and Haystack have many things in common: Multiple views for the same resource, flexible collections of resources, tagging, etc. Haystack goes beyond HYENA with its powerful (but also complex) view prescriptions. Tasks, making the user's activity explicit for context-sensitive adaptation, are an intriguing concept that should probably be added to HYENA in the future. Extractors integrate Haystack into the desktop environment and are something that HYENA currently does not have. On the other hand, HYENA goes beyond Haystack by offering more RDF-specific functionality. That functionality is less suited for end users where Haystack's focus lies. HYENA provides more flexible authoring abilities, thanks to its semantic wiki component. Haystack also does not have multi-user support and access control; it is a purely personal application.

25.5 The Social Semantic Desktop (NEPOMUK project)

The Social Semantic Desktop (SSD, [BDE+08]) is the vision of enhancing the desktop environment with semantic web technologies to help bridge two kinds of data islands:

[1] This is called "currying" in functional programming.

On one hand, applications are isolated islands of data; it is difficult to exchange and link data between different applications. On the other hand, desktops of different users are isolated islands, exchanging data is difficult, especially if live collaboration is involved.

Services. SSD is envisioned as a service-oriented architecture (SOAP-based) that spans desktops and provides various services to users and application implementors: *Resource management* allow users to manipulate semantic relations about anything on the desktop. *Application integration* integrates legacy applications with the SSD (for example by importing pictures and their meta-data from a photo application). *Notification management* informs of important events via emails, RSS, text messaging, etc. *Offline access* ensures that important data is available when a desktop is not connected. *Resource sharing* enables different SSD users to work on the same resources. *Search* works either locally or across a peer-to-peer network of semantic desktops. It exploits semantic relations and can find related items. *Access rights management* and *user group management* is used to secure shared data. A *publish and subscribe* mechanism offers feeds of relevant information. More advanced features include profiling to intelligently support the user and ensure trustworthiness of communication partners; and data analysis to support finding and presenting information.

Ontologies. Ontologies are a crucial element for making the exchange of data easy on the SSD. The authors distinguish four levels of ontologies:

- Representational ontologies: define the vocabulary for defining other ontologies. Examples include RDFS and OWL. Concepts appearing at this level: classes, properties, constraints.
- Upper-level ontologies: are high-level and domain-independent and provide basic concepts on which more specific ontologies can be based. These are often called "common sense concepts" that are basic for human understanding of the world. Concepts appearing at this level: person, organization, process, event, time, location, collection.
- Mid-level ontologies: Provide a bridge between the very abstract upper-level ontologies (hard to understand) and the very specific domain ontologies. They are also called "utility ontologies". Concepts appearing at this level: company, employer, employee, meeting.
- Domain ontologies: describes concepts and relations specific to a particular domain of interest. The same concept may have different representations in different domains reflecting domain contexts and assumptions. Domain ontologies are often partially based on mid-level and upper-level ontologies. Concepts appearing at this level: group leader, software engineer, executive committee meeting, business trip, conference.

25.5.1 Comparing with HYENA

Much of the Social Semantic Desktop is still vision, some of it has been integrated into the KDE Linux desktop system. As such, it is difficult to compare it with HYENA. In general, HYENA focuses on specific usability problems and the web environment, NEPOMUK has a very broad scope and focuses on the Desktop.

25.6 The DBin platform: A complete environment for Semantic Web Communities

The paper [TM08] presents a system for publishing RDF data among communities, in a peer-to-peer (P2P) fashion. This includes mechanisms for presenting and for distributing the data. The user interface has been implemented as an Eclipse plugin. A *Brainlet* is a plugin for DBin that supports a particular RDF ontology. It can contain

- Ontology data, to support editing and displaying the RDF data.
- A declarative definition of a graphical user interface for displaying data. This definition specifies both widget layout and interactions between widgets. For example, selecting a class from a tree might lead to its instances being displayed in a table.
- Templates for form-based editing.
- Templates for form-based querying. Blanks in the query are filled in by the form.
- A trust model and information filtering rules.
- Scripts for helping with entering URIs, e.g. to make sure that existing URIs are reused or that a given scheme is adhered to.
- Scripts for adding menu commands or buttons to the user interface.
- Supporting files for icons, help content, etc.
- Java code for user interfaces that cannot be implemented with the built-in mechanisms.

Usage of DBin is focused on browsing data: One initially creates an identity as a pair of a public and a private key (Sect. 3.4.2). Then one subscribes to *P2P topic channels*, depending on one's interests. A subscription to such a channel is bidirectional; adding new data makes it available to all subscribers. At any time, a topic channel can contain links to new data and communities. Improved support for new communities is provided by ad hoc installation of Brainlets.

The most common approach to P2P distribution of RDF data is to distribute query evaluation across peers. In contrast, DBin synchronizes the data denoted by a topic channel with peers. Thus, all data is available locally, offline and disconnecting peers are not critical. Each topic channel uses a GUED (Group URIs Exposing Definition) to specify what data is relevant for it. The first step to collecting this data is to compute a list of resources, usually via queries such as "Select all resources of type 'Paper' which have the topic 'Semantic Web'". The actual data being synchronized between peers is each statement whose subject or object is among the computed resources. The *RDFGrowth* server is used (and necessary) for hosting topic channels. Apart from the GUED, it also tracks the URIs published per channel, allowing clients to look up related channels. Files are supported by DBin by designating web servers to which files can be uploaded and then have a stable URL. This URL is used as the subject or object of RDF statements. RDF data can be published as an *RDF dump* on the web, in one of the RDF exchange formats that can be processed by non-DBin clients. RDFGrowth servers can be queried for a list of communities where each has zero or more dumps associated with it. Pieces of data, usually statements, are digitally signed to track authorship. As

data normally grows monotonically, explicit revocation commands need to be issued to remove it. These revocation commands show up as RDF data and are also digitally signed. Local trust policies allow one to specify what data (including revocations) to show and what data to ignore.

25.6.1 Comparing with HYENA

DBin focus is on supporting communities via topic channels one can subscribe to. HYENA's unit of synchronization are projects or repositories. DBin's way of replicating all shared information locally is similar to how HYENA handles this task. An RDFGrowth server is needed to host channels; this diminishes the P2P capabilities of DBin. The cross-cutting nature of topic channels makes it easy to share data between communities. One difficulty arises when browsing a resource that has been extracted for a channel: If it contains a link to an external resource, it is not clear how to follow it. Linked data principles won't work, because data is managed in a decentralized fashion. Digital signing and revocation are powerful features, but they also add much overhead. File handling is a weak point of DBin: They are hosted by a single server and not synchronized.

Chapter 26

Semantic wikis

Contents

26.1 Overview

Semantic wikis are wikis to which RDF support has been added. HYENA can be seen as a semantic wiki turned inside out: it is a generic RDF editor to which wiki support has been added. This chapter reviews a few representative semantic wikis and compares them to HYENA. *Semantic MediaWiki* builds on Wikipedia's MediaWiki engine and adds semantic annotations. The *KiWi platform* is a comprehensive vision for a social semantic wiki platform. *AceWiki* is a semantic wiki with support for controlled English.

26.2 Semantic MediaWiki

Semantic MediaWiki (SMW, [KVV+07]) has been created with the goal of making it possible to annotate typical wiki data semantically and not to offer full-blown ontology editing. SMW identifies three problems with current wikis: First, consistency of content: Content is often duplicated and thus prone to be inconsistent. Second, accessing knowledge: Finding and comparing information in large wikis is difficult. Third, reusing knowledge: The text format of wikis does not make it easy to import and export information.

MediaWiki (on which SMW is based) provides several means for structuring content. The basic structure in wiki text consists of text style and hyperlinks. Additionally there are *categories* where each page can be assigned zero or more categories. Each category is reified as a wiki page. Categories can be organized hierarchically. Synonyms lead to redirection: Pages with different names for the same topic redirect to one standard page for that topic. Homonyms lead to disambiguation pages: Ambiguous page names are disambiguated by appending more information in parentheses (example:

"1984 (book)"). All variants are then listed on a disambiguation page. Lastly, one can include external *template* pages in a page and optionally fill in content via parameters.

Semantic MediaWiki builds on these basics and supports *semantic annotations*. SMW sees pages as concepts and introduces mechanisms to better describe them. Its *properties* and *types* correspond almost directly to RDF, but the core construct is not the resource, but the wiki page. This makes it easier to use for people who are already familiar with wikis. *Properties* are either links to other pages or values that are attached to a page. Both are characterized by a property key. Example: `London is the capital city of [[capital of::England]]`. The only addition to a traditional wiki link is the property key. Properties are introduced by using them on a page, they can be further configured by editing a wiki page that has the name of the property. On that page, one can assign a *type* to a property (which in RDF would be its range). Types also have dedicated wiki pages and a few of them are predefined. For example "String" (character sequences), "Date" (calendar dates), "Geographic coordinate" (locations on earth) or "Page" (wiki pages).

These structures can be mapped to OWL: pages correspond to abstract individuals, properties correspond to OWL properties, categories correspond to OWL classes, property values can be abstract individuals or typed literals. The schema modeling power of SMW has intentionally been kept simple, but it has also been used in conjunction with more expressive background ontologies.

To exploit the semantics, a "factbox" below the page content lists all properties of a page and all properties whose value the page is. Queries can be formulated in a description-logic-like syntax and either be invoked interactively or their result can be embedded in a wiki page. The semantic data of a wiki can also be exported as OWL/RDF. In that RDF, page URLs are converted to special URIs that use content negotiation to either serve the corresponding wiki page (human access, via a web browser) or the exported RDF (automated access). This corresponds to the linked data principles (Chap. 3).

26.2.1 Comparing with HYENA

HYENA solves the issues of traditional wikis, that have been raised by SMV, quite will. it goes further than SMW when it comes to content-specific editing and finer-grained pages. Pages can even be queried for, because they are stored in resources. On the other hand, SMV's lightweight semantic enhancements of wiki data is clever and completely complementary to integrated ontology editing or form-based editing.

26.3 The KiWi platform

KiWi [SEG+09] is a platform for managing databases with structured and unstructured data. Small web applications are used to display the data. A semantic wiki would be one such application. The paper argues that while there are already platforms that support diverse kinds of content, they are often not able to change data so that it meets unforeseen uses or it is difficult to support new kinds of data. Contrarily, KiWi builds on what it calls "content versatility": *content items*, its atomic pieces of information hold both human readable content (in XHTML) and meta-data (in RDF). The KiWi platform can be extended via KiWi applications, KiWi services (for managing the content), KiWi widgets, KiWi actions (invocable by the end user), and exporters/importers. The paper describes the following KiWi applications:

- KiWi wiki: a typical semantic wiki.
- Dashboard: a FaceBook-like homepage of a user that displays a user's recent activities (edits, ratings, watched content items, information derived from a user's network of trust), recommendations of things that might be relevant to the user, a history of recently edited content items, tags applied by the user.
- TagIT: conceived as a "youth atlas of Salzburg", lets users collaboratively annotate a map to point out interesting locations.
- KiWi search: a generic, centralized search that directs back to the previous application when a content item is inspected. The search provides keyword search and faceted navigation (by tag, type, and person).
- KiWi inspector: lets advanced users access more detailed information about content items.

The most interesting aspect of the KiWi architecture is its service-orientation with services such as a revision service, a transaction service, or a content item service (to manipulate content items). The core concept of the data model is the *content item* which is any unit of information (wiki pages, multimedia, users, roles, rule definitions, layout definitions, widgets, and so on, extensibly). Each content item is identified by a URI. It consists of core (meta-)data (author, creation date), content, and associated RDF data. Core data and content are stored as XML in a relational database, while RDF data is stored in a special RDF repository. The content supports linking, is versioned, can be queried and transformed via XSLT. It is the human-readable representation of a content item. The RDF repository attaches custom attributes to each triple to support things such as versioning, transactions, authorship, and reason maintenance.

KiWi distinguishes explicit tagging (manually assigned; records the tag, the tagger and the tagged content item) and implicit tagging (automatically assigned; directly relates two content items).

26.3.1 Comparing with HYENA

KiWi is an impressive vision for social semantic wikis and similar in many regards to HYENA (which predates KiWi). KiWi's focus is on social applications and inference, HYENA's focus is on personal information management and direct manipulation (e.g. via lenses). HYENA's generic interface is currently more powerful than KiWi's, which has better support for special purpose applications. KiWi's versioning and provenance tracking of RDF go beyond HYENA's abilities.

HYENA's core data management is simpler, it only has resources, not a combination of content and RDF. This does not mean that it is less capable, except in one way: KiWi's markup is stored as XML and its structure can be exploited in queries. HYENA's conceptual model comprises the search calculus which goes beyond KiWi's formalizations. KiWi does not provide synchronization between systems.

26.4 AceWiki

AceWiki [Kuh08] uses Attempto Controlled English (ACE) to let end users collaboratively edit and query ontologies. Thus it is more an ontology editor with wiki-style editing. ACE has a bi-directional mapping with OWL and can even store facts whose

expressiveness is beyond OWL. It can also be used to query ontologies. Reasoning is performed with existing OWL reasoners. Prior evaluation confirmed that ACE, supported by a predictive editor (which is loosely similar to auto-expanding editors for programming languages), allows non-experts to author quite complex ontologies.

26.4.1 Comparing with HYENA

While a fascinating feature, controlled English is not currently relevant for HYENA, as it does not focus as much on knowledge management.

Chapter 27

Faceted navigation

Contents

27.1 Overview

This chapter explores the state of the art in faceted navigation. We first look at a paper that extends faceted navigation so that it works for RDF data. This paper identifies faceted navigation as working well for RDF data, introduces basic operations and defines metrics for automatically ranking facets. The second system is called *Ontogator* and answers the question: What kind of search engine service and application programming interface are needed for supporting a variety of semantic view-based search interfaces? The third system, */facet*, applies faceted browsing to heterogeneous semantic web repositories where scalability is an issue: multiple types need to be supported (as opposed to a single type being supported by many faceted browsers) and relations between types need to be made navigable. The last system, *gFacet* implements *hierarchical facets*: facet values can, recursively, be specified via faceted browsing.

27.2 Extending faceted navigation for RDF data

The paper [ODD06] describes RDF data as large, interconnected and heterogeneous (no fixed schema). Thus, navigation mechanisms should be scalable, allow graph-based access (following links), not depend on a fixed schema and not require users to know the structure of the data beforehand (it has to be *exploratory*). Four traditional navigation mechanisms have been identified as suitable for RDF:

- keyword search: difficult to explore data, not graph-oriented.

- explicit queries: difficult to write, require a-priori knowledge of the data.
- graph visualization: does not scale.
- faceted browsing: manually constructed, domain-dependent, only partial support for graph-based navigation.

In faceted browsing, one classifies an entity by attaching a set of key-value pairs to it. A *facet* means viewing a key as the set of all values that appear at its side. As such, it is a summary of all currently shown entities. For navigation, the facet values are used as *restrictions*: When the user selects one of them, only entities are shown that have the corresponding key-value pair. Advantages of faceted browsing are: it is exploratory, as the system suggests next steps via the restriction values; it is visual, there is no need to type anything; and it prevents meaningless queries by only showing restrictions that do not lead to empty results.

To formalize faceted navigation, the paper introduces the following operations:

- Basic selection: selects resources that have a given key-value pair.
- Existential selection: the pseudo facet values NONE and ANY indicate that a property should have the cardinality zero or a cardinality greater than zero.
- Join selection: creates paths of operators to express queries such as "all resources who know somebody, who in turn knows somebody named Stefan".
- Intersection: Combines operators that all have to hold at the same time. A conjunction of the conditions.
- Inverse selection: Resources are often just as determined by properties pointing *to* them as they are by their properties. Inverse selection adapts basic selection, existential selection and join selection to incoming properties.

Next, the paper defines metrics for automatically ranking facets.

Descriptors: What data best describes entities for humans? According to Ranganathan, a good facet candidate is data that is either temporal (year of publication, date of birth), spatial (conference location, place of birth), personal (author, friend), material (topic, color) or energetic (activity, action). Automatically finding good descriptors is very difficult and beyond the scope of the paper.

Navigators: What facets allow for efficient navigation? If one views faceted navigation as traversing a decision tree, then one is looking for facets that minimize path lengths in such a tree. The paper defines three metrics for good navigators. These metrics are combined by weighted multiplication and have to be recomputed for each navigation step, because the information space changes. The first metric is *predicate balance*, as balanced trees optimize decision power. Intuitively, it measures how evenly a property attaches objects to subjects. The second metric is *object cardinality* measures that a property has neither too little values to choose from, nor too many. The third metric is *predicate frequency* which ensures that entities use a property as often as possible. Then, restrictions affect as many entities as possible.

27.2.1 Comparing with HYENA

HYENA does not just use facets for navigation, but also for editing. It advocates using assisted querying (Sect. 7.6) for path queries instead of making faceted navigation more complicated. HYENA does have existential qualification, though: When using a only facet key as a restriction (and not a key and a value), the facet value is assumed to be ANY. This allows one, for example, to select all resources that have types. In the future, negation would be an interesting feature, both as NOT (key=value) and as NOT (key=ANY).

Finding intuitive facets is hard which is why HYENA relies of manually defined facets. Lastly, HYENA supports broad facets where the assignment of a facet value can be annotated, for example: Who made the assignment? When was the assignment made?

27.3 Ontogator—a semantic view-based search engine service for web applications

While the semantic web makes large amounts of relational data available, querying that data is difficult. Thus, much research has focused on making specifying complex queries as simple as possible. *View-based search* is rooted in the tradition of faceted classification. It uses multiple simultaneous views on an information space, each showing a different classification. Search consists of restricting the result by selecting values in a view. A key differentiating feature between view-based search on one hand and traditional keyword and boolean search on the other hand is that with the former, views are used for both querying and for summarizing the result. Each restriction is annotated with the number of items that a restricted result would have. This kind of look-ahead prevents the user from making selections that return no items or too many items.

Ontogator [MHS06] solves the following problem: What kind of search engine service and Application Programming Interface (API) are needed for supporting a variety of semantic view-based search interfaces? Where traditional keyword search can make do with fairly simple APIs, the requirements for a general view-based search API are much more complicated: facet visualization (including hit counting), facet selection, and result visualization are needed in addition to the search logic.

27.3.1 Requirements for a view-based search API

A view-based search API needs to provide the following services:

1. Return facets and facet values, including hit counts.
2. Basic view-based search: Facet-based boolean querying.
3. Specifying the shape and content of the result.
4. Reclassifying a result set along different facets. For example, when the user opens a new view on an existing result set.
5. Combining traditional keyword-based search with view-based search.
6. Knowledge-based semantic search: Inferred categories allows one to find data that is only implicitly related to the search categories.

More abstractly, Ontogator strives for adaptability and domain independence, standards compliance, extensibility (with regard to querying mechanisms), and scalability.

27.3.2 View projection: preprocessing the knowledge

In the *view projection* phase, Ontogator translates RDF data into a set of trees whose roots are facet keys, with (possibly hierarchical) facet values underneath them. These trees form the categories of the items to be categorized. This phase consists of two steps:

1. Projecting the view (facet value) tree from the RDF graph. A mix of RDF-based data and Prolog-like rules are used to specify how to project a category tree (per facet) from the RDF domain data. This might involve dropping categories in the middle of the tree, pruning category subtrees, eliminating cycles in the data, and expanding categories with multiple parents into separate subtrees. *Category identification* assigns unique identifiers to categories whose URI might appear several times across facets or several times within a facet (if it has multiple parents within that facet).

2. Linking items to the projected categories. Categories can be directly attached to a domain resource or via a path of properties.

View projection encapsulates the domain semantics and thus makes the components feeding on the data it produces domain-independent. View projection generates indices that enable search to efficiently scale to millions of search items. A disadvantage of this phase is that one cannot incrementally add data.

27.3.3 Queries and extensibility

One uses queries to request data from Ontogator. Basic queries contain two clauses. An items clause selects items, a categories clause selects category subtrees to be used for grouping items. This allows users, for example, to search for items by keyword and group them according to geolocation-based proximity. Each clause is specified by combining selectors with union and intersection operators. Item selectors include a category selector (returns all items in matching categories), an item URI, a keyword (to be found in the item data). Category selectors include a category URI, a category identifier (disambiguated URI), a keyword (to be found among facet keys and values). Extensibility is achieved because new selectors just need to return a list of matching items. That means that it is relatively easy to integrate external services.

27.3.4 Indexing

View-based search requires efficient access to the following data: Given a category, one needs direct subcategories, directly linked items, all transitively linked items, and the path to the tree root. Given an item, one needs the categories it belongs to. Ontogator generates indices for direct subcategories and directly linked items, but generates the transitive closure on the fly, which can be done efficiently, because the category IDs adhere to a prefix labeling scheme. The scheme means that the IDs also contain the path to the root.

27.3.5 Comparing with HYENA

HYENA's faceted navigation works dynamically and does not scale as well as the view projection approach of Ontogator. On the other hand, incremental changes are directly considered, without having to recompute the view projection. The reloading phase of HYENA, where it parses data in RDF such as schema and facet definitions, is a loose equivalent of view projection. HYENA facet modeling is less powerful than Ontogator's, but most common use cases are easier to specify. In addition to navigation, HYENA also supports facet editing. Lastly, HYENA handles both narrow and broad facets, where Ontogator only handles narrow facets.

27.4 /facet: a browser for heterogeneous semantic web repositories

The paper [HvOH06] explores how heterogeneity of data affects faceted navigation. Two key points are that such a browser needs to support multiple types (where many faceted browsers specialize on a single type) and that with multiple types, the necessity increases to make relations between types navigable.

27.4.1 Requirements for multi-type facet browsing

The authors describe the following requirements for multi-type facet browsing:

- Dynamic selection of facets: faceted navigation scales fairly well to large facet sets, because it only displays the facets that apply to the current result set. Two phenomena require special consideration:
  - `rdfs:subClassOf` hierarchy: For a single class the approach to only show the facets/properties of the result set works well. With subclasses, one has to decide between showing the union of facets and the intersection of facets. The difference is noticeable for classes that are higher up in the hierarchy: The union always allows one to start browsing, but might show too many facets. The intersection focuses on the facets that the subclasses have in common, which usually are the most important ones. But if the subclasses have too few facets in common, one cannot continue browsing. The paper excludes customization as an alternative to union and intersection as being out of scope. Note that this discussion applies to any kind of value hierarchy, not just to subclasses (which are the primary facet of the implementation presented in the paper).
  - `rdfs:subPropertyOf`: structures the facet keys hierarchically. While that structure can help with organizing the facets, it also leads to more facets (and more complexity).
- Search in addition to navigation: Navigating deeply nested facet trees is complex. And so is handling many facets or unknown facets. Both problems can be alleviated by allowing one to select facets by search.
- Multi-type queries: If a facet has many values and if these values are also faceted, then it makes sense to use faceted navigation to filter the values. The paper views this problem differently, as relating two sets of entities. It assumes that each set

holds instances of a single type and offers a relation that connects both types as a filter criterion. For example: “created by” is a relation that connects the types “Work of art” and “Person”.

- Run-time facet specification: The paper advocates configurable facet definitions (as opposed to hard-coded ones). Automatic generation of such definitions is desirable, as is the ability to later adjust generated definitions to the needs of, say, less technical users that might not want to see all details.

- Facet-dependent user interfaces: Some facet values are better displayed using a custom technique. For example, locations on a map or dates in a calendar.

27.4.2 Functional design for multi-type facet browsing

The authors have taken the following design decisions:

1. Browsing multiple resource types: the type of a resource is one of the facets, it is primary among the facets inasmuch as it is always shown and browsing usually starts with it.

2. Semantic keyword search: Keyword search with auto-expansion works for

   - the facet values in a single facet box. Helps with navigating the hierarchy of a single facet.
   - the facet values in a tree of all facets. Helps with finding the right facet.
   - property values of instances, via the type facet box. It displays matching instances in a tree, as children of their type. Helps with finding the right type to start navigation.

3. Specifying queries over multiple resource types: happens in two steps. First one queries for a set of instances of a type A. Then one switches to another type B. /facet now gives the option of relating the old set and the new set of B instances by showing all properties whose domain is A and whose range is B.

4. Run-time facet specifications: are stored in a separate RDF file that is automatically generated and then usually customized by hand.

5. Facet-specific interface extensions: can be plugged in. /facet comes standard with a timeline visualization that displays several time-ranged facets in parallel, across all currently active sets of instances.

27.4.3 Comparing with HYENA

Making type the dominant facets does not always work. For example with tagging, some tags are almost like types, such as all wiki pages about books being tagged with #book. Relating two result sets is something that will be investigated for future versions of HYENA. In this case, /facet limits itself by relying on types. This does not work well when a facet relates resources and literals. It is also problematic whenever the type is not the defining characteristic of a result set.

27.5 gFacet: a browser for the web of data

gFacet [HZL08] implements what the paper calls *hierarchical facets*: facet-based browsing where facet values can, recursively, be specified via faceted browsing. Its main contribution is that of using a graph-based visualization of the facet paths. The paper gives the impression that gFacet combines graph-based browsing and faceted browsing, but it actually only uses graph-based visualization techniques to better present the chained facet restrictions. gFacet also uses data-specific visualizations for facet values, currently the only special case (apart from value lists) is a map visualization for geographical data.

27.5.1 Comparing with HYENA

HYENA advocates assisted querying (Sect. 7.6) for path queries, the usefulness of chained facet restrictions is under investigation.

Chapter 28

Synchronization and versioning

Contents

28.1 Overview

This chapter reviews literature about RDF synchronization and versioning. Synchronization means computing the changes necessary so that two different repositories end up containing the same data. Versioning means keeping a history of changes with the ability to look up old versions or to revert to them. One crucial problem in this area is how to handle blank nodes. HYENA avoids the issue by ignoring blank nodes, with the option to rename them to URIs. This is in line with linked data principles which advise to avoid blank nodes.

The first paper is about *RDFSync*. Its unit of synchronization, the *minimum self-contained graph* (MSG), ensures that blank nodes are synchronized correctly, even if they have different (internal, temporary) IDs. The second paper presents a framework for RDF versioning and evolution. It defines small-grained evolution operations that can be scaled up to support powerful ontology changes. If blank nodes are encountered, their statements are grouped similarly to MSGs to ensure that they are properly handled. The last paper is about *SemVersion* a system for versioning RDF ontologies that draws its inspiration from CVS. Blank notes are made identifiable by adding inverse functional properties.

28.2 RDFSync: efficient remote synchronization of RDF models

This paper [MTEBG07] solves the blank node problem by computing so called *minimum self-contained graphs* (MSG). The definition of an MSG is recursive:

- The MSG of a ground statement (no blank nodes) is the statement itself.

- The MSG of a statement with (at most two) blank nodes is the MSG of all statements that contain one of those blank nodes.

This explains the adjective "self-contained", the statements within an MSG do not refer to any blank nodes outside the MSG. Decomposing a graph into MSGs is unique. By applying a deterministic labeling algorithm to MSGs, one can give them unique hashes. These hashes are then used for synchronization.

Compared to HYENA, the main advantage is the handling of blank nodes. Disadvantages are that no equivalent to HYENA's journal exists which allows one to list changes on either side that happened since the last synchronization. Handling of blank nodes suffers somewhat, because, small changes (such as changing a property) result in new MSGs, instead of indicating what blank nodes have changed. Lastly, which side to keep versus merging is specified for the complete sync, and not per resource, as in HYENA.

28.3 A versioning and evolution framework for RDF knowledge bases

The paper [AH06] presents an approach to versioning and evolving RDF data and ontologies. Blank nodes are considered when it comes to specifying atomic changes. There are positive atomic changes (adding statements) and negative atomic changes (removing statements). The statements of an atomic change are defined via the concept of an *atomic graph*:

> A graph is atomic if it cannot be split into two non-empty graphs whose blank nodes are disjoint.

That means that a graph with a single statement that does not contain blank nodes is atomic. So are graphs that, for a given set of blank nodes, comprise exactly those statements that contain any of those blank nodes. Thus, an atomic graph is the same as the minimum self-contained graph from the previous section. To handle blank nodes properly, the blank nodes of a positive atomic change must be disjoint with the blank nodes of the graph to change. A negative atomic change C_G must contain all statements of the graph to change that contain blank nodes from C_G. These requirements ensure proper handling of blank nodes in distributed settings. A drawback of this approach is that one often has to remove and add many statements if just a single statement with blank nodes is to be changed.

Atomic changes can be group into nested sequences, so-called *change hierarchies*. This allows one to review changes at various levels of detail (statement level, ontology level, domain level, etc.) and to improve transaction management. For distribution, change hierarchies can be encoded in RDF and annotated with data such as IDs identifying predefined action classes, the name of the editing user, the time of the change, or a string with a human-readable description of the change.

Evolution patterns support higher-level changes and are an extension of the concepts that have been introduced so far. An evolution pattern comprises graph patterns with variables that encode changes and a migration algorithm. The former can be considered templates for change hierarchies with parameters to be filled in. The latter is necessary if the evolution pattern applies to ontology classes. Then the instances of a changed class must be migrated so that the ontology is consistent again. An example

includes removing a property from a class resulting in the property being removed from its instances.

28.4 SemVersion: an RDF-based ontology versioning system

SemVersion [VG06] draws its inspiration from the CVS version control system. It versions ontologies stored as *models*, sets of triples. Versions are arranged in a directed acyclic graph that denotes how versions were derived and includes version branches. A version can contain its own meta-data such as a comment. To create a new version that is stored in the DAG, one commits a complete model, one commits a diff between the previous version and the current version, or one merges two versions (usually the most current versions of two branches). Structural diffs contain statements added and statements removed and are useful if a new version contains only few changes. To merge a branch c into a branch b, one looks for the most recent common version a and then adds the diff between c and a to b. In addition to structural diffs, SemVersion also provides semantic diffs whose content depends on the ontology language used to define the semantics of the content. It is defined as the structural diff between the asserted and inferred triples of two models.

Blank nodes are handled by adding inverse functional properties with unique IDs to them. This is similar to HYENA's renaming of blank nodes to URIs.

Part VII

Evaluation, summary, and future research

This part evaluates HYENA. The chapter on integrating structured and unstructured data shows how HYENA supports this kind of integration. It also elaborates the requirements of quick note taking and how these requirements influenced HYENA's features. A user study examines how a group of test users applied HYENA: How they searched and navigated, what data they created and how, and what data they revisited after creation. Many web 2.0 applications such as calendars and discussion forums are partially replacing (and improving on) wikis. Thus, the author has surveyed a random group of wiki users to find out what wiki features will survive in the long term. The last chapter offers a summary of this dissertation and describes future research.

Chapter 29

Integrating structured and unstructured data

Contents

29.1 Overview

This chapter makes the case as to why wikis are great for note taking and how HYENA improves on them by integrating structured and unstructured data. First, it is shown how one can start with unstructured data in HYENA and incrementally introduce structure. Then, it is explained how small notes, a frequent source of unstructured data, are stored, annotated with meta-data, and retrieved. The chapter finishes by showing how data can be collated and how files (another kind of data that is unstructured to HYENA) are handled. Many of the mechanisms that have been introduced in previous chapters are shown in use, together.

29.2 Wikis and structured data

Wikis are very popular as a general purpose tool for many kinds of information management: On one hand, they can handle many kinds of data, because they can simulate more specialized tools. For example, an outline can be simulated by having a bullet list; forms can be simulated by creating a skeleton with data to fill in; tables are entered as plain text. In all cases, the flexibility of unstructured text and the ability to edit and save are what makes the simulation possible. In that regard, wikis share many of the

desirable properties of paper: location can be used, one is relatively free when entering data, annotations can be put anywhere.

On the other hand, data entry is quick. Whereas with specialized tools, one has to frequently use the mouse or is relatively fixed in one's way of managing data. Three examples: First, creating a table in a word processing program takes a while; in a wiki, a table is created by typing only. Second, using a spreadsheet program limits one to tables only; in a wiki, tables can be changed to something else should the need arise. Third, a database management systems such as Microsoft Access require one to prepare for new kinds of data and data entry is slowed by having to use forms; in a wiki, data entry is free-form.

But while wikis often come close enough to more specialized tools, they have one obvious deficiency: handling structured data. While there is structured data in wikis (tables, lists), it is difficult to retrieve, process, and export: Data inside a wiki page is static, it cannot be sorted or filtered. It is difficult to share data. If it is needed in two places, one has to copy and paste. Exporting implicit structured data is also relatively complicated.

Similarly, structured data *about* wiki pages (meta-data) is useful for retrieving and collating them. For example, one could use a query to display all pages with a given tag or that have been created before a given date. Meta-data can enable multi-dimensional classification, where "linking as classification" in traditional wikis is much more limited.

HYENA has been created to meet the above mentioned challenges, with the goal to go beyond either-or when it comes to structured and unstructured data. The wish was to freely mix both. While doing so, one should initially profit from a wiki's ability to quickly enter data. Structure should be something one optionally adds afterwards, in an incremental fashion. The result turned out to benefit both the wiki side of HYENA (which became more expressive, because more things were modeled explicitly) and the database side of HYENA (which became more flexible in presenting information). Additionally, HYENA provides proxies for external data such as files that stay up-to-date when that data changes. Its project-centric approach makes backup and synchronization of content easy.

29.3 Incrementally introducing structure

Users like wikis for information management, because they allow them to introduce a small amount of structure, on demand. That is, initially, one uses the freedom of unstructured text to quickly put down information. Afterwards, one can introduce more structure—in the form of pages and links—to better organize the data. This is similar to creative writing [Mag] where one initially brainstorms and then introduces structure and linearization.

Traditional wikis have pages and links, HYENA has three more means of structuring:

1. Medium-grained subpages: blocks of content that are usually manually assembled into a compound page.

2. Structured data: RDF data can be embedded inside a wiki page.

3. Fine-grained notes: contain often just one sentence. They are annotated with meta-data that helps collating them via queries.

Notes are the topic of Sect. 29.4, the following example illustrates the first two items. Let us assume that we start with one large page:

My page
```
Topic: Gardening
= Indoor =
* Plants that need little light
= Outdoor =
* Evergreens
= Guerrilla gardening =
* History: Green Guerrilla group, 1973
```

Then one might decide to turn "Outdoor" into a sub-page that is embedded. This has the advantage of simplifying independent editing and sharing. To do so, one uses an empty `\embed{}` command which when displaying the page turns into a link offering to create a new RDF resource (which can be a page or data). Furthermore, "Guerrilla gardening" is turned into link in a similar way:

My page
```
Topic: Gardening
= Indoor =
* Plants that need little light
= \embed{} =
= Miscellaneous =
* \link{}
```

We add content to the newly created pages and they look as follows.

Outdoor
```
* Evergreens
```

Guerrilla gardening
```
* History: Green Guerrilla group, 1973
```

Lastly, we turn the line

```
Topic: Gardening
```

into structured data. This is done by creating a new wiki page for the topic whose title is "Gardening". Then one adds a property with the key `tagging:tag` to the original page that refers to the topic page. Inside original page, we can now refer to the value of property `tagging:tag` instead of mentioning the topic directly. This has two advantages: the topic becomes a link one can click on to get more information; and one can use the topic in queries and navigation (see Sect. 29.4).

```
Topic: \prop{tagging:tag}
```

Fig. 29.1 shows what the final version of the page looks like in a web browser. Creating the embedded and the linked page has filled in page IDs and titles.

29.4 Small notes and meta-data

In everyday life, there are all kinds of small notes that come up: What to remember, who to meet, a bookmark, etc. Furthermore, brainstorming can be viewed as collecting notes and collating them in various ways. Often, people find it too complicated to

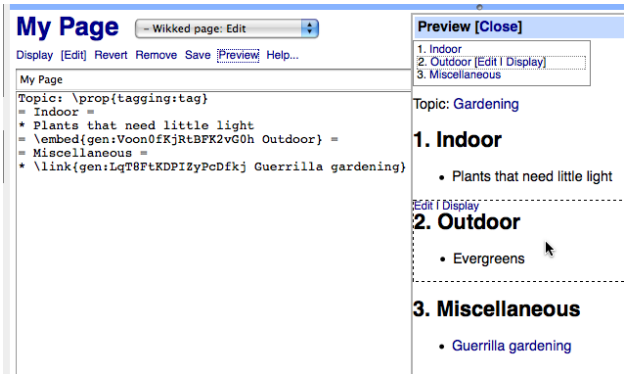


Figure 29.1: The version of the compound wiki page where the links have been filled into the external text. To the right is a preview area. It shows the embedding structure on mouse-over (only).

manage this kind of note with a computer and rely on post-it notes instead. This way, it is simple to create a note, but recall and archiving are a problem [BVKKS08]. On the computer, wikis are already a great improvement for note-taking over standard software such as word processors or spreadsheets. The free-form way of data entry reduces the typical barriers for quickly getting the data into the computer. Alas, wikis are not ideal for the fine granularity of notes. One usually collects several notes on a single page where the page also serves to categorize the notes on it. But classification of notes often needs to be multi-dimensional. For example, what domain ("private", "business", "my sister's wedding") a note applies to and when it is "due". Thus, the note would need to be put on two pages, one for the domain, one for the due date. Even if there was no other category, the latter way of time-based organization is not ideal. HYENA uses a combination of ideas to ease note taking:

- Quick entry: A key combination brings up a dialog to enter a single line of text, return commits the note. The latter is in line with web applications such as Google Maps that let the user enter data in a single text field instead of forcing them to switch between several ones.
- Adding meta-data: HYENA has a simple syntax for adding meta-data (such as tags, dates, persons or other context) to a note. In the user interface, tags are auto-expanded to a list of all tags that already exist. Similarly, dates can either be entered via a simple syntax or via a calendar widget.
- Bookmarking: If the note is a URL, HYENA downloads the web page title as the content of the note and attaches the URL to the created resource. Meta-data can be added in the same manner as for text notes.

Below there are three notes to illustrate the syntax (each note is a single line). Details on the syntax are given in Chap. 8.

- `Write paper #todo #due=dec-19`
  File the note to write a paper that is due next December 19 under "todo".
- `http://www.amazon.com/dp/1847064140/ #read #when=2010-01-01`
  A note whose title is retrieved from amazon.com as "Amazon.com: Discourse of

Blogs and Wikis (Continuum Discourse): Greg Myers: Your Store". The note is tagged with "read" (i.e., things to read), and is due in January 2010.

- `Dinner #time=tom @Jill`
  Dinner tomorrow, with Jill.

Further speed-up is achieved by the fact that you don't have to pick a place where to store your note before creating it. That is, the note is piled, not filed [BVKKS08]. It is initially "orphaned" and retrieved by its meta-data: At the very least, one can find it by date (of creation or last modification) and in the list of all wiki pages. The next section describes various ways for easily accessing resources. As an aside, while notes start out as empty pages with the note text in the title, one often later elaborates them and adds content. Tags and facets can be edited at any time, using the same notation as during creation.

Other meta-data such as the date of last modification is handled in a way that is generic for all RDF resources. RDF does not make much distinction between meta-data and data. Both are used extensively to search and browse resources, as we shall see in the next section.

29.5 Browsing resource sets

Notes are initially not linked to anything. Thus it is important that there be other means for finding them. HYENA provides a variety of search and browsing mechanisms for resources that work well for wiki pages, too. All of the views listed below visualize *resource sets*.

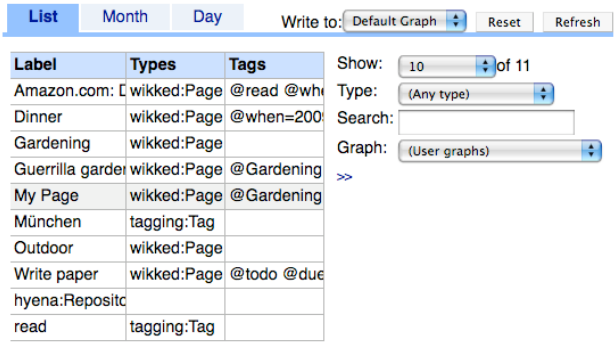


Figure 29.2: Master view, "List" tab. Displays resources in a table, each row holds a resource's label, its types and its tags. The table can be filtered via the controls to the right: One can limit the amount of resources displayed, one can filter by type, by text, or by graph (which are URI-named partitions of an RDF repository).

List tab

The list tab shows a table where each resource is a row and the columns display the label, the types, and the tags (Fig. 29.2). This list can be filtered by type (e.g. to focus just on wiki pages), by text or by graph (sub-sections of an RDF repository).

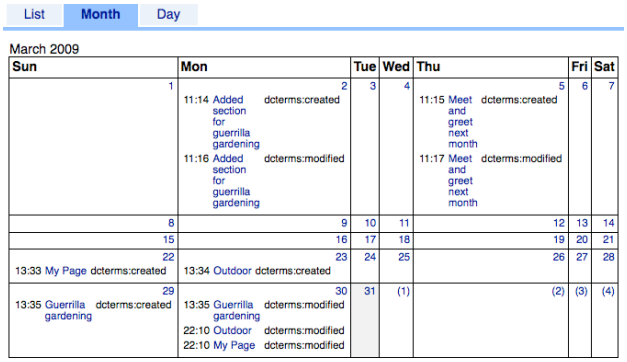


Figure 29.3: Master view, "Month" tab. Resources are placed on a month grid depending on what time information is contained in them. The same resource can appear multiple times, such as "Guerrilla gardening" which has been created on March 29 and modified on March 30.

Month tab

The month tab shows a month grid where resources are placed according to time information that is attached to them (Fig. 29.3). That means that due dates, date of creation, date of last modification are ways of locating a resource.

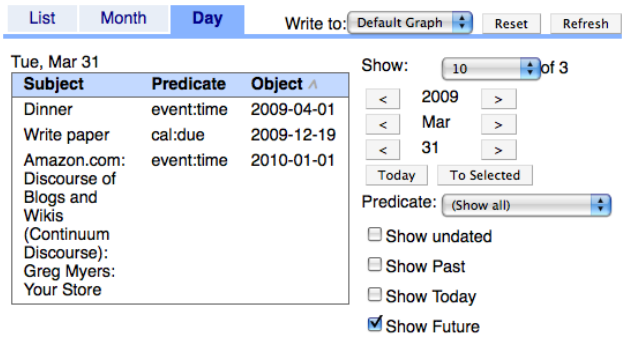


Figure 29.4: Master view, "Day tab". Lists resources by time. By focusing on the future, one gets a list of reminders and things happening in the future.

Day tab

HYENA's "day" tab displays a set of resources depending on their time (Fig. 29.4). By limiting the time to the future, one gets a list of reminders, similar to Gelernter's lifestreams [FG96].

Faceted navigation

If a resource is a set of (key,value) pairs then a key can be seen as a *facet* across a set of resources. Put simply, a facet is a group of tags; facet members are usually shown annotated together with their facet to ease navigation. Music programs are good examples of faceted navigation; genre, artist, album are all facets. When displaying

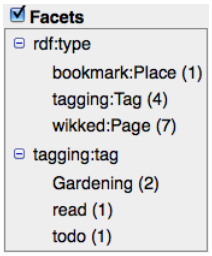


Figure 29.5: For a list of resources, this tree aggregates the values for the two facets `rdf:type` (the types of a resource) and `tagging:tag` (the tags of a resource). Accordingly, one resource in the list has the type `bookmark:Place`, four have the type `tagging:Tag`, etc. Clicking on one of the values filters the resource list to contain only those resources that contain the given (key,value) pair.

a list of resources, HYENA aggregates the values of some properties into a facet tree (Fig. 29.5). When clicking on one of the values, the list only shows those resources where the corresponding (key,value) pair appears. Then the facet tree is recomputed for the new list, again giving contextual information.

29.6 Collating data

Embedding resource sets

Sect. 29.5 described resource sets as the underlying data structure for browsing RDF. The configuration of such a set can be saved to RDF and retrieved later. Resource sets are not static, they are computed each time they are displayed or embedded and thus always reflect the current state of the RDF repository. The options for embedding a resource set are: comma-separated links, bullet list with links, or a sequence of embeddings.

SPARQL queries

Meta data is stored in the same RDF resource as the wiki page content and can thus be accessed by the RDF query language SPARQL. In HYENA, one can save SPARQL queries to a resource and display or embed their result as a sequence of pages, a table, a bullet list of links, or a comma-separated list of links. This allows us to answer two common requests for wiki page meta-data: "What pages have been changed last?" (Fig. 29.6) and "what pages are orphaned?" (Fig. 29.8). Fig. 29.7 shows the result of the former query displayed as a table. Note that HYENA does not expect end users to write queries, resource sets can be seen as a limited end user friendly version of queries. But whenever they are not powerful enough, one can resort to SPARQL queries. Storing them in a resource makes it possible for experts to author them and end users to just use them, without knowing SPARQL.

News feeds and comment feeds

Feeds are sequences of small-grained wiki pages. A *news feed* displays entries with a title and the date of last modification (Fig. 29.9). New entries are added first. A

Query: last changes

```
1 SELECT ?resource ?modified WHERE {
2     ?resource dcterms:modified ?modified .
3 } ORDER BY DESC(?modified)
```

Figure 29.6: Lists all wiki pages with the ones that have been most recently modified appearing first. The first line specifies what to display in the table of results. The second line says that we are looking for any resource (see below for how to restrict this to just wiki pages) that has a date of last modification attached to it. The last line describes how to order the result.

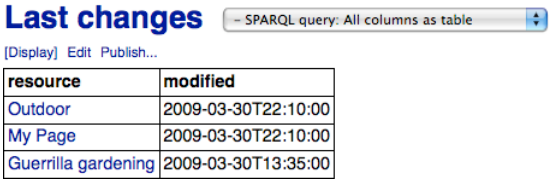


Figure 29.7: Displaying the result of Fig. 29.6 as a table, using the embedding style "all columns as a table". Other embedding options are "first column as bullet link list", "first column as comma-separated links", "first column as table via a lens".

Query: orphaned pages

```
1 SELECT ?page WHERE {
2     ?page rdf:type wikked:Page .
3     OPTIONAL { ?s2 ?p2 ?page }
4     FILTER ( !bound(?s2) )
5 }
```

Figure 29.8: Lists all pages that have no inlinks. Line 2 states that we are looking for wiki pages. We then try to find other wiki pages that point to the current one (line 3) and reject the current page if anything can be found (line 4).

Figure 29.9: Displaying a news feed. A news feed is basically a sequence of small wiki pages, which are highlighted when the mouse is over them, as is the case for the first entry.

comment feed additionally displays the author of an entry. New entries are added last. Feeds can be embedded anywhere on a wiki page and can be published as RSS feeds (Sect. 29.7.2).

Tags as compound pages

Tags can become compound pages (Fig. 29.10): One adds wiki content to the tag resource and a special `\linkin` command lists all resources that that refer to the tag (which includes resources where it is the value of `tagging:tag`, the RDF property for tagging). If one would rather embed the tagged resources, one uses `\embedin`.

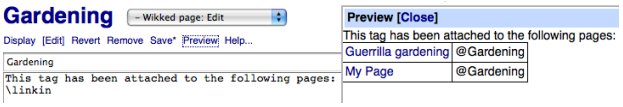


Figure 29.10: The tag "Gardening" has been turned into a wiki page that links to all pages that are tagged with it. Such a wiki page is the ideal place to describe what a tag is about.

29.7 Files and data export

29.7.1 Using files

Any file can be linked to. Clicking the link in HYENA/Web will download (or display, depending on the web browser) the file. In HYENA/Eclipse, it will be opened in the appropriate program. Image files in a project can be displayed inside a wiki page. And one can configure a resource to list the files in a directory (Fig. 29.11). Images in the directory are displayed as thumbnails (Fig. 29.12). All proxies for files (including the one in the file list resource) are kept up-to-date should a file be moved or renamed (Sect. 9.5.3).

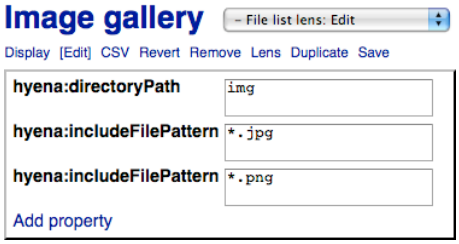


Figure 29.11: Using a lens to edit a file list resource. The path `img/` of the directory with the files is given as a literal. Future versions of HYENA will use proxy resources (Sect. 9.5.3) for this purpose.

29.7.2 Publishing resources as files

HYENA has a generic mechanism for *publishing* resources, for exporting them to files. In HYENA/Web that means that resources are served as files. In HYENA/Eclipse, we

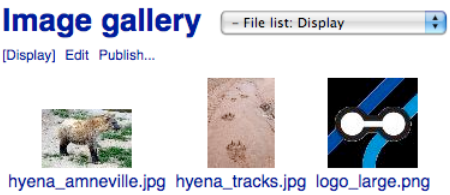


Figure 29.12: Displaying the file list resource from Fig. 29.11.

write the published data to a file and open it in an external program. Wiki pages can be published as printable HTML pages or as LaTeX source. News feeds or comment feeds can be published as RSS feeds. The published feeds are protected via normal HTTP authentication (in contrast to the cookie-based authentication of the HYENA web application), so that they can be accessed by feed reader programs.

29.8 Discussion

This chapter explained how HYENA integrates various kinds of data. Special care has been taken that small notes were easy to capture and manage. For more information on small notes consult Sect. 25.2 about related work on *information scraps*. An extended version of this chapter has been published as [Rau10].

Chapter 30

User study

Contents

30.1 Overview

A user study was performed to find out how people used the CoIM implementation HYENA. 5 people[1] participated in the study which lasted a week. They used online HYENA accounts that the author provided for them. This made it possible to log user interface activity. The evaluation comprises the following parts:

- Structure of the RDF repository: What data was stored in the repository, what structure does it have?
- Use of features: What features were used and how often?
- Questionnaire: A questionnaire allowed participants to go into detail about how they used HYENA, what they perceived as strengths and weaknesses and what features they wanted to have in future versions.

30.2 Structure of the RDF repository

After the study, the repositories filled by the participants were examined automatically. This revealed the following details about their contents.

- Tags: Each user created an average of 2.8 tags of which 1.6 were used once, 0.4 were used twice, 0.4 were used three times, and 0.4 were used four times. The tag average is relatively low, as is reuse. This can be attributed to the learning curve and to some users spending considerable less time with HYENA than others.

[1] See the discussion as to why the group of participants was so small.

- Types: Each user created an average of 15 resources. 64% were wiki pages, 21% were tags, 4% were bookmarks. The remainder consisted of lens-related constructs and persons (created when a person is mentioned in the title tags). This shows that HYENA was mainly used as a wiki, although wiki pages performed a variety of functions (see questionnaire results below).
- Wiki pages: Users created an average of 9.6 wiki pages. Per page, there were 0.33 references to other resources (made from within the wiki content, which excludes tags), 0.48 references from other resources to the page and 0.5 tags. This shows that HYENA enables two complementary ways of wiki usage. First, the traditional wiki way where all pages are interconnected and are usually rooted in a single start page. Second, access via tags. In the latter case, the wiki page is orphaned (not referenced by another resource) and can only be accessed via its tag. This is reflected in the fact that approximately half the pages had an incoming link and half the pages had a tag.

These numbers can be compared to the HYENA repository of the author which has been in heavy use for over a year.

- Tags: There were 73 tags, of which the top 4 tags were used 50 times, 36 times, 23 times, and 13 times. On the other hand, the bottom 28 tags were only used once, and 15 tags were used twice.
- Types: The repository contained 443 resources of which 56% were wiki pages, 28% were bookmarks, and 16% were tags.
- Wiki pages: There were 247 wiki pages. Per page, there were 0.05 outlinks, 0.17 inlinks, and 1.21 tags.

30.3 Use of features

The logging of user interface actions during the study showed what features were used and how. By summarizing the differences between time stamps, rough estimates of usage times can be given.

- Search: 98% of the time, facet-based navigation was used. Type-based filtering was used 2% of the time. Text-based search was never used. Facet-based navigation subsumes type filtering via a combo box that is shown close to the table of the resource set. The study suggests that this additional way of providing this functionality is not necessary. That text-based search is used so little is a surprise and warrants further investigation. Two reasons can be assumed: Due to tagging, users did not find text-based search as important (this corresponds to the author's experience). Or that the text box for full-text search was difficult to find.
- Creating a resource: There are four ways to create new resources in HYENA. The menu command "New wiki page" was used 34% of the time. 34% of the time, a reference (a link or an embedding) to a new page was inserted into an existing page. This is the traditional wiki way of creating a wiki page. The menu command "New any resource" was used 24% of the time. The menu command "New lens" was used 7% of the time. This shows again how the traditional way of creating wiki pages is complemented by the creation of orphaned wiki pages, which are usually accessed via their tags.

- Changing the inspector: Per user, the "display" mini-link was used 19.4 times, the "edit" mini-link was used 17.2 times, and the inspector combo box was used 14.4 times. Thus, while the combo box is an important user interface element, the mini-links are used more often to change the inspector.

- Revisiting resources: Each resource that was manually created (this excludes resources that were, for example, imported from a CSV file) was, on average, visited 2.07 times during the first hour after its creation and 1.4 times afterwards. There were also resources that were visited only after the first hour or only during it. This result is congruent with note-taking studies, where not all notes were revisited after their creation.

- Skill level: The simple "wiki" level was used an average 22min per person, "wiki + tags" was used 38min, "wiki + tags + database" was used 1h 27min, "wiki + tags + DB + semantic web" was used 19min. This shows that the skill levels were used and fulfill a need for task-specific interfaces. For example, in "pure wiki" mode, HYENA's user interface becomes much simpler and one can focus on using it as just a wiki.

- Tabs: The tabs "Vocabulary", "Query" and "Assisted Query" were each used under 5s per user. The "Day" tab was used an average 7min 31s per user, "Diagram" was used 14min 32s, "Month" was used 14min 39s, "List" was used 1h 48min 57s. As expected, the standard table tab dominates usage, but showing the data in a calendar and visualizing it as a diagram was also popular.

30.4 Questionnaire

Three of the participants filled out a questionnaire which provided more information on how they used HYENA, what they liked or disliked, and their feature wishes. Participants used HYENA to keep notes, journal entries, ideas, links, appointments, outlines, and wish lists. They liked title tags for quickly adding meta-data, web page titles being fetched automatically when adding a URL, and light markup (tables, bullets) for structuring their content. Data they imported included HTML files, text files, tables, and pictures. While using HYENA, the participants appreciated the multiple ways of visualizing and accessing data: Journal entries could be displayed in a calendar view, the data and its relationships could be displayed as a diagram, and faceted navigation could be used to browse the data.

Participants also mentioned what they might use HYENA for in the future: collaboration, journaling and blogging. For collaboration, live editing was considered a useful future feature. For journaling, HYENA is currently missing encryption and more control over text styling and layout. For blogging, more sophisticated text styling and layout was on the wish list, as was the ability to export blog posts to blogging web applications such as Blogger or Wordpress. Participants would like more extensive export of data, for example: tables, text files, PDFs of wiki pages, and diagrams (graphs, trees, mindmaps). Exporting the complete repository as a ZIP file for backup was also mentioned (which would complement synchronization in this regard). Further suggestions for improvement were to make entities such as images and PDFs true first-class citizens; currently, handling files is not as comfortable as handling other entities such as wiki pages. More help content and tighter integration of it was asked for, to cope with

the breadth of features provided by HYENA. Lastly, a WYSIWYG wiki page editor was considered desirable.

30.5 Discussion

This study is still ongoing. The work involved in participating in the study (learning HYENA, using it for a week, filling out the questionnaire) makes it difficult to find volunteers. Nevertheless, the five participants cover a broad spectrum of users: Four Germans participated——two colleagues of the author's, a journalist, a student—and a Nigerian small business owner[2].

The study confirmed the decisions that were made for HYENA. As expected, wiki pages were used in various capacities (notes, journal entries, outlines, wish lists, ...). The wiki features were used in two ways, with similar frequency. On one hand, in traditional wikis, there is a single start page and all other pages are reachable from it via hyperlinks. New pages are created by going to an existing page and adding a link to a page that does not exist yet. On the other hand, CoIM supports note taking via orphaned[3] pages that are accessed via their tags. New pages are created directly, without first having to find a page from which to link. The popularity of the latter way of wiki use shows that it does help users. Finding pages via their tags was also very popular, far more so than via full text search.

Bookmarks were the most frequently used kind of structured data. This shows the usefulness of structured data at the object level. To further encourage this kind of use, creating and managing lenses probably has to be simplified even further, so that it becomes practical even for casual users. In general, participants liked HYENA's versatility and embraced quick note taking, tagging, faceted navigation, the calendar views, and the structure diagram. They also suggested many new use cases, proving that there is a need for the versatility offered by HYENA. On the flip side, participants sometimes found the complexity of the generic user interface intimidating. Skill levels (and their acceptance) hint at a solution to this problem: By providing more special-purpose views on the CoIM data, particular tasks can be greatly simplified.

[2] He found a blog post where the author of this dissertation asked for participants in the study and was willing to participate.

[3] Not linked to from another page.

Chapter 31

A survey on wikis: What features have long-term merit?

Contents

31.1 Overview

Wikis have been a popular web application for some time now. But the recent rise of Ajax [Gar05] has changed the perception of web applications: Many wiki-like web sites have been appearing, often specialized to do a single task well (where wikis are more universal). Examples include Google Calendar and Flickr.

This begs the question: What aspects of wikis will survive in the Web 2.0 age? What aspects are worth preserving? To answer this question, one has to find out what people use wikis for, what features they like and what features they are missing. A small survey on this topic [Rau08b] helped the author do that. This chapter interprets the survey results as requirements for the next generation of wikis and then argues that HYENA, by mixing wiki pages and RDF data, is well equipped to meet those requirements.

31.2 The survey

The survey has intentionally been relatively simple. For example, it was probably not representative for all potential wiki users, because the participants (a total of 23) were chosen in an ad-hoc fashion by announcing the survey to the author's colleagues and to a mailing list about semantic wikis. But the results are still interesting and point out several possible trends for wikis.

The survey participants answered in percentages indicating how important a given fact was to them or how often they performed a given activity. The reported percentages are averages of these answers. Note that the answers do not necessarily apply to a single

wiki; many participants use several wikis. The following sections present groups of questions and observations that the author derived from the answers.

31.2.1 What is the purpose of your wiki? What do you use it for? (Table 31.1)

Even though traditional wikis are all about text, people use the flexibility of text to store data inside wiki pages. Intuitively, this is already obvious, if one looks at all the lists and tables that are contained in traditional wikis. The survey made this hunch more concrete: On average, 91% of the wiki use is about collecting data or knowledge. Naturally, this data and/or knowledge could be more flexibly processed if it were explicitly stored and had dedicated editors. For example, spreadsheets handle tabular data well, so it would be nice if one could embed little spreadsheets inside a wiki page. Note that the survey results do not indicate that wikis should become pure databases. Rather, being able to mix text and data is what seems to make wikis attractive.

Collect data or knowledge	91%
Coordination, planning, project management	56.75%
Web site, light-weight content managment system	54.5%
Document creation (and later publishing)	48.75%
Discussions, forum	42%
Brainstorming, (possibly shared) whiteboard	39.75%
Weblog, relatively small journal-style entries	16%

Table 31.1: Question: What is the purpose of your wiki? What do you use it for?

31.2.2 Who uses the wiki? (Table 31.2)

Observations: At heart, wikis can be considered groupware. Still, having the wiki information available anywhere and the flexibility in structuring information, makes wikis good personal information managers: Survey participants attributed an average importance of 50% to this task.

Several collaborators, all reading and writing	60.25%
Personal use, a single person	50%
Few editors, many readers	46.5%

Table 31.2: Question: Who uses the wiki?

31.2.3 Current and future wiki features, wiki alternatives

The following is a list of web applications that the survey participants use as alternatives to wikis for some tasks:

1. BackPack
2. Blogger
3. del.icio.us

4. Facebook
5. Flickr
6. Google Calendar
7. Google Docs
8. iusethis
9. Online Contacts
10. Trac
11. Wordpress
12. WikipediaReview.com

Interestingly, the majority (all except 1, 4, 11) of the web applications is very task-specific. Accordingly, Table 31.4 indicates that users would like to see more task-specific editing support in wikis. The difficulty is to do so without significantly raising the learning curve.

It is curious to note that wiki users do like that a wiki is available anywhere you can connect to the web (Table 31.3) and use it for personal information (Table 31.2), but rank offline use relatively low (which would increase the availability of wiki information for personal use).

Information roaming: the wiki information is available anywhere you can connect to the web.	78.5%
Collaborating: a wiki is useful for sharing and jointly editing information.	77.25%
Linking: a wiki allows me to relate and collate pieces of information.	72.75%
Publishing: a wiki is useful for disseminating information.	67%

Table 31.3: What core aspects of (traditional) wikis are you interested in?

31.3 Discussion

The users' need to collect knowledge or data is met by HYENA by its support for structured data and the ability to annotate it with wiki markup. The typical scenario of everyone writing and reading a wiki occurred most frequently, but so did personal use and having few editors and many readers. The former is supported by synchronization so that one's personal data is available offline, too. The latter is support by access management. Users expressed a need for comfortable task-specific editing, which HYENA provides via editors which are can be considered small task-specific web applications. Among the important wiki features, HYENA offers the highest ranked ones such as an editing history, file upload, wiki page metadata and easy printing. Others such as a WYSIWYG editor and live collaboration still need to be implemented.

Version control (editing history, who edited what, unlimited undo, etc.)	78.5%
File upload and management	69.25%
Wiki page meta-data (annotations and labels describing the content of the page)	58%
WYSIWYG text editor	56.75%
Generate a PDF file from a wiki page	52.25%
Diagrams (UML, mind maps, organizational charts, etc.)	48.75%
Finer-grained wiki pages	47.75%
Outliners (edit indented lists such as tables of contents)	46.5%
Live collaborative editing (all editors work on the same copy of the document, changes show up immediately)	46.5%
Spreadsheets (with calculation)	46.5%
Discussions (forums)	41%
Offline editing, synchronization	37.5%
Calendars	37.5%
Form-based data entry (similar to MS Access)	37.5%
Blogs	25%

Table 31.4: Question: How important are the following features to you (independently of whether your wiki has them or not)?

Chapter 32

Summary and future research

Contents

32.1 Overview

This chapter summarizes this dissertation and describes future research.

32.2 Summary

Despite the positive effects of the proliferation of information in modern society, it has also made information management more difficult. There is an ever increasing amount of information and it exists as disconnected heterogeneous islands. *Connected information management* (CoIM) investigates how to build the next generation of information management systems that prevents disconnectedness. It presents a ubiquitous platform with a unified model for the content.

Two technologies exist that remedy two kinds of disconnectedness: Dynamic, Ajax-based web applications lead to ubiquitous availability of information and applications, semantic web technologies integrate many kinds of data. CoIM and its implementation HYENA have been created to continue these threads and enhance both technologies with more integration abilities and formal foundations. Where the semantic web data format RDF can be used to integrate various kinds of *structured* data, CoIM extends it so that *all* kinds of data can be integrated. A CoIM repository is truly universal and can contain unstructured text, structured data, and files. That means that data that was previously spread across applications can be managed in one place, for example, all data relevant to a given project such as database entries with contact information, notes in unstructured text, diagrams in PDF files. This data is not just consolidated in a single place, but can be mixed via wiki markup and queries. For example, a wiki page can be created that explains a sub-project and lists all contacts pertinent to that sub-project. Furthermore, even though the database entries are stored in RDF, CoIM's *RDF editing meta-model* ensures that it can be edited with forms like

in relational database applications such as MS Access or Filemaker. Contrary to those applications, the schema is very dynamic and extensible; tags, links, and other kinds of data and meta-data can be added. CoIM data does not exist in isolation, thus two kinds of external data are supported: Data in files can be imported and exported. For example, Firefox bookmarks and BibTeX files can be imported and JSON data can be exported. Data in external databases can be integrated dynamically, by storing live references in RDF. Integration of files is based on this mechanism, the file system is considered an external database of the RDF repository.

Traditionally, one had to choose between offline access to one's data via desktop applications and online access via web applications. Web applications have recently made progress by providing offline modes, often by temporarily storing data in the SQL databases provided by web browsers. HYENA goes one step further. All data can be edited either online or offline and synchronized between the two modes. This synchronization is universal, no preparation or adjustment is needed to support new kinds of data. Currently, different kinds of data have different navigation metaphors. Files are browsed as trees of nested directories, web pages are searched with keywords, music is explored using faceted navigation. CoIM provides *multi-paradigm navigation*: several kinds of navigation are integrated and work for all kinds of data. Additionally, a new kind of navigation is investigated, one that is based on facets that cross-cut facets. This new kind of facet is called *meta-facet*. Examples include "time" (which comprises facets such as "time of creation" and "time of last modification") and "text" (where the search text can appear in any facet). One pervasive kind of data that is difficult to integrate are small notes such as reminders, todos, and ideas. They are usually managed either externally to the computer or by applications that were never intended for this task (such as word processors and spreadsheet programs). HYENA encourages note taking via several measures: Creating notes is easy, one can pile instead of file (no category or title needed), some meta-data for retrieval is added automatically, and more can be added effortlessly. Multi-paradigm navigation ensures that notes can be recalled, synchronization ensures their availability.

The core of HYENA is an extensible framework with services for both information management and user interface interactions. This framework serves several purposes: HYENA itself is based on it and it allows one to add support for new kinds of data. Additionally, it can be seen as a foundation on which to base one's data-intensive applications.

32.3 Future research

Several themes for future research have emerged as a result of the experience gained with HYENA.

Special-purpose applications. At the moment, HYENA is a generic tool where inspectors provide some degree of special-purpose editing. The problem with generic tools is that while they are usually very powerful, they are also difficult to understand, especially for non-technical users. Thus, the next version of HYENA will have special-purpose sub-applications that occupy the complete browser window. For example, discussion forums, a blog editor, or a calendar. The generic foundations of HYENA remain, meaning that all these sub-applications operate on a common database and can reuse modeling constructs such as tags. How to do this will be informed by the results of the survey that determined what features of wikis should stay generic and what

features should be replaced by special-purpose mechanisms.

Note taking. The author has extensively used HYENA for managing small notes and organizing them in multiple dimensions. Note taking can be extended in two ways. First, one can support more modalities. The ubiquity of cameras and microphones in desktop computers and mobile devices, makes photos and sound recordings an attractive way of taking notes. Sketches are another promising candidate. Second, one can support more meta-data, to improve recall. As even a small additional effort might prevent users from capturing a note electronically, automatically added meta-data is to be preferred. The most prominent example is location information, which can be retrieved using the W3C standard for geo-location[1].

Collaboration. HYENA's current focus is on personal information management. Collaboration is already supported via authentication, synchronization, and conflict management for page editing. Several features are envisioned to extend this support. Commenting on someone else's data is an important part of collaboration. Inspiration can be drawn from the annotation features of Adobe Acrobat where things like virtual post-it notes can be added to text. HYENA should offer something similar, if possible for both wiki pages and locally cached HTML pages. Distributed version control (DVC) should be enabled for all of HYENA's data. DVC allows peers to synchronize and preserves all old versions, even if they have been created on different peers. Peer-to-peer synchronization is currently possible for all data, but distributed versioning is only available for wiki pages. It still has to be designed and implemented for RDF, where one faces several challenges (Sect. 21.4). In recent years, live collaborative editing has become increasingly popular. The most recent entrant into this arena is Google Wave[2]. This kind of editing is especially useful for wikis, because many kinds of conflicts can be avoided.

More external data. For many personal information management and groupware tasks, more external data needs to be integrated. One example is email. In teams, individuals often exchange emails that are either important for the project history or that need to be read by several team members. If one can import these emails into HYENA, they can be accessed and annotated by the whole team. Even for personal use, being able to file emails with other data is useful. Other examples of external data whose integration and ubiquitous availability would be beneficial are calendars, contacts, and bookmarks.

A middleware for social linked data. More and more web applications are being written that need to support heterogeneous data and social features such as collaboration. Each one tends to reinvent features that are already part of HYENA. And the next version will gain even more of those features. If linked data principles are integrated into HYENA, it should thus become a middleware for social linked data.

REMM, data editing. There are several areas in which REMM could be improved: Editing data in a table is a proven techniques for quickly accessing several entities at the same time, while having a good overview. The usability role model in this case are

[1] http://dev.w3.org/geo/api/spec-source.html
[2] http://wave.google.com/

spreadsheets. Naturally, columns would be typed, but there are spreadsheet programs, such as Apple Numbers or the discontinued Lotus Improv, that also work this way.

Usability could also be improved by hiding more of RDF's complexities. An unsolved problem is how to handle data that is shared by several entities. Currently, if the data is displayed inline in a form, it will be removed when only one of the entities is removed. If the shared data is only shown as a link, it won't be removed, even if all of the entities are removed. The more general problem is called *extent computation*: How far does a given entity extend? Is the extent context-dependent or fixed? Another challenge with RDF is vocabulary management. End users should be able to get started quickly. The current solution is to start with generated predicate URIs that have human readable labels and can be renamed to more "beautiful" URIs later. This should be complemented by making existing vocabularies easy to (re-)use via cheat sheets and similar means. Experts will want more sophisticated tools for vocabulary management such as statistics on the use of a particular URI.

Furthermore, advanced features for data management and meta-data management can be envisioned. Data refactoring would change the schema while adapting the data it describes. Operations include extracting classes, merging classes, and moving data between classes. Visual editing of the schema would also help and could be based on the unified modeling language UML [BRJ99]. The author has had positive experiences with using UML class diagrams for sketching REMM schema ideas. Both are a natural fit, due to the object-oriented feel of RDF data management. Lastly, the REMM schema should probably get its own serialization to RDF, instead of being derived from OWL. The author believes that there is room for a schema language specializing on data modeling with an expressiveness between RDF schema and OWL.

Software engineering. Information management is about managing a multitude of small pieces of data. Software engineering faces similar challenges, which is why one can probably apply many of the ideas of CoIM to software engineering. Folksonomies are one especially promising candidate; it could add a lightweight semantic layer on top of programming language source code that greatly helps with navigating the code. Work in this area would build on research already performed by the author [ZR04, RR04b, RR04a, Rau04, RR05b, RR05a, RAN07]. Another possible research direction is to improve the reflective capabilities of programs [Rau05c], by expressing part of the internal structure in RDF. HYENA has already started to do so, as it can be configured via the RDF repositories it contains. Ideas for using HYENA as a universal model editor for software engineering have been collected in [Rau05a, Rau05b].

More future research. More future research is listed at the end of some of the chapters. App. B describes the benefits and challenges of implementing inferencing and sketches a solution.

Part VIII

Appendix

Appendix A

Wikked syntax

A.1 Wiki Creole

Tab. A.1 summarizes the WikiCreole[1] markup standard.

A.2 LaTeX

This section gives a brief overview of the LaTeX markup constructs in HYENA. The next section goes into more detail.

A.2.1 Resources

`\link{uri}`	→	link to resource
`\link{}`	→	"create page" link
`\embed{uri}`	→	embedded resource
`==\embed{uri}==`	→	embedded resource with a second-level title
`\embed[nohead]{uri}`	→	embedded resource without a title
`\annotate{uri}`	→	add resource as annotation

A.2.2 Files

`\imgfile{picture.jpg}`	→	image
`\linkfile{picture.jpg}`	→	link to file
`\namefile{picture.jpg}`	→	name of file

A.2.3 Other

`\href{http://foo.com}{Foo Inc.}`	→	link to external web site
`\url{http:/bar.org}`	→	link to external web site
`\raw{html}`	→	enter HTML directly

Inlinks, in-embeddings, links to other projects, links to published data: see Sect. A.3.1.

[1] http://www.wikicreole.org

`//italics//`	→	*italics*
`**bold**`	→	**bold**
`~**escaped bold**`	→	**escaped bold**

`* Bullet list`
`* Second item`
`** Sub item`
	→	o Bullet list
		o Second item
		o Sub item

`# Numbered list`
`# Second item`
`## Sub item`
	→	1. Numbered list
		2. Second item
		2.2. Sub item

`Link to [[wiki page]]`	→	Link to wiki page
`[[URL|link name]]`	→	link name
`= Large heading`	→	**1. Large heading**
`== Medium heading`	→	**1.1. Medium heading**
`=== Small heading`	→	**1.1.1. Small heading**
`===* Unnumbered small heading`	→	**Unnumbered small heading**
`No` `linebreak!`	→	No line break!
`Use empty line`		Use empty line
`Forced\\linebreak`	→	Forced line break
`Horizontal line:` `---`	→	Horizontal line: ———
`{{Image.jpg|title}}`	→	Image with title

`|=|=table|=header|`
`|a|table|row|`
`|b|table|row|`
	→	Table

`{{{`
`== [[Nowiki]]:`
`//**don't** format//`
`}}}`
	→	`== [[Nowiki]]:`
		`//**don't** format//`

Table A.1: WikiCreole markup

A.3 Commands

A.3.1 Linking and embedding

Resources:

- `\link{uri}{label}`: label is optional.
- `\embed{uri}`
  - Option: @noframe
  - No arguments: self-embedding
  - Looking up embedder IDs: in the info dialog for the currently displayed resource.
- `\linkin`: show incoming references as links.
- `\embedin`: show incoming references embedded (useful for tags).
- `\annotate{uri}{label}`: add resource as an annotation (similar to a post-it note).

Files (the following commands create external literals for synchronization via Brij):

- `\linkfile{file.txt}`: link.
- `\namefile{file.txt}`: name.
- `\imgfile{picture.jpg}`: image.

Inter-project links

- `\linkext[webapp=http://localhost/hyena]{proj:repository#resource}{label}` (project is optional, label is optional)
- `\linkfileext[webapp=http://localhost/hyena]{proj:dir/file.txt}{label}` (project is required, label is optional)

Other:

- `\url{url}`
- `\href{url}{label}`
- `\linkpub[@label=xxx,...]{publisherId}{uri}`: link to published data (feeds etc.). Works in web and in Eclipse mode. The URI argument is optional; if missing, the current resource is used. You can look up the exact command via the dialog that comes up after clicking on the "Publish" mini-link.

Notes:

- Labels next to RDF URIs are completely ignored and serve only to improve readability for humans.

A.3.2 Miscellaneous commands

- `\lastChanges[limit,style]`: with limit an integer (default is 10) and style either `table` or `bullets` (default is table).

A.3.3 Google gadgets

Include Google gadgets via the `\gadget` command:

- When you have found a gadget you like, use the configuration page "Add this gadget to your webpage" until your gadget is set up according to your wishes.
- Then click the button "Get the Code". The resulting code is

  ```
  <script src="..."><script>
  ```

- Copy the value of the `src` parameter. It is the (only) argument to `\gadget`.

A.4 Mixing in structured data

- Self-embeddings: using `\embed` without an argument embeds the data of the current resource. You do have to specify an embedder, though, because the default is the Wikked page embedder.
- `wikked:useLens` (via the Wikked page lens): Attaches a lens projection to the bottom of a wiki page. That projection is present both during displaying the page and editing it.

Appendix B

Inferencing

Contents

B.1 Overview

Inferencing is the process of deriving new RDF statements from existing ones. It is the foundation of the semantics of RDF schema and of some parts of OWL. This chapter first presents inferences from RDFS and OWL that are interesting for future versions of HYENA. It also describes high-level HYENA-specific inferences such as inferring facets or computing property values. Then the challenges involved in implementing those inferences are explained and a solution is sketched. The chapter concludes by enumerating some related work.

B.2 Kinds of inferences

The following inferences should be considered for future versions of HYENA. RDFS inferences:

- `rdfs:subClassOf` is transitive.
- `rdfs:subPropertyOf` is transitive. Additionally, if b is a subproperty of a then all statements whose predicate is b are also asserted for a.
- `rdfs:domain` infers the types of subjects.
- `rdfs:range` infers the types of objects.

Inferencing in OWL:

- `owl:FunctionalProperty` infers objects with the same subject as equivalent.
- `owl:InverseFunctionalProperty` infers subjects that have the same property value as equivalent.
- `owl:SymmetricProperty` infers symmetry.
- `owl:TransitiveProperty` infers transitivity.
- `owl:equivalentClass` infers two classes as equivalent. The simplest way of implementing equivalent classes is to treat them as equivalent resources (see below).
- `owl:equivalentProperty` infers equivalence between properties: If one property in a set of equivalent properties is the predicate of a statement, all other properties get a corresponding statement.
- `owl:sameAs` infers equivalence between resources: If one resource in a set of equivalent resources is the subject of a statement, all other resources get a corresponding statement.

Ideas for special-purpose inferencing in HYENA:

- Inferred facets: For example, every time a tag is present, a more general tag should be added, too.
- Computed property values. All kinds of computations could become part of an inferencing infrastructure.

Note that transitive properties p are often better handled by computing transitivity for a separate property p' and making p a subproperty of p'. As a result, all p statements are also asserted for p', but the transitive closure is only performed for p'. Thus, one can use p to determine direct relatedness and p' to determine transitive relatedness. One example is the transitive ancestor-descendant relationship being derived from the parent-child relationship.

B.3 Challenges of inferencing

The main challenge of inferencing is when to update. The simplest way to do that is to recompute all inferences, at the same time. A decision must then be made when to invoke that process. It cannot be performed too often, because it is time-consuming. Thus, one usually lets the user invoke it manually, whenever she feels enough changes have been made to warrant the time it consumes. For most data, this is not a problem and can be handled similar to HYENA's reloading life cycle event (Sect. 18.3.2). Labeling is one area where this approach can be problematic. Labels need to be as current as possible. Thus, if subproperties are used to infer `rdfs:label` then the inference should be performed any time the subproperty changes.

Sometimes, having the result of an inference is not enough, one also needs to know what caused the inference. An example is equivalence. If two resources A and B are declared equivalent then properties of A will turn up in B. But those properties will be read-only. As the intended effect of equivalence is to create the illusion that these two resources have been merged, this is not satisfactory for editing. Instead, if the user tries

to edit the read-only property of B, one should change the original statement in A and then redo the inference. This is a specific case of a general problem of inference: if an output has been cause by an input, one cannot change the output directly, one must change the input. Thus, one probably has to record for most (if not all) inferences what caused them. In logic, this is called *truth maintenance* [McA90].

B.4 Outline of a solution

HYENA currently prefers a more dynamic solution to inference, because it avoids the time lag of invoking inference. For example, subproperties of `rdfs:label` are collected during reloading (Sect. 18.3.2) and used to dynamically look up the label of resources. Inference will become more important as support for linked data is implemented in HYENA, due to the distributed nature of that data and the integration that inference enables.

One way of implementing inference is to store all inferred statements in a separate named graph. Except to the inferencer, that graph is read-only. REMM would immediately handle such a solution correctly, because it already supports read-only graphs. Recomputation is triggered similarly to reloading and first clears the graph and then computes the inferences. Integrating various kinds of inferencers is simple, they only need to store their inferences in the graph. Truth maintenance could be performed by logging the causes of an inference in a separate relational database. Maybe this data can be used to implement incremental computation, because whenever a statement is removed, one might be able to undo the inferences it caused. Undoing is particularly challenging when inferring transitivity.

B.5 Related work

The *SPARQL Inferencing Notation*[1] (SPIN) is "a collection of RDF vocabularies enabling the use of SPARQL to define constraints and inference rules on Semantic Web models. SPIN also provides meta-modeling capabilities that allow users to define their own SPARQL functions and query templates. Finally, SPIN includes a ready to use library of common functions." SPIN can be used to calculate the value of a property based on other properties, to perform constraint-checking with closed-world semantics, and to determine rules to be executed under certain conditions. Uses of the last feature are incremental reasoning, interactive applications and automatic initialization of properties after resource creation.

The *NEPOMUK Representational Language* (NRL, [SESH07]) assigns roles to named graphs. That allows for data aggregation (via graph inheritance) and views that apply different ways of inferencing or other kinds of transformations (such as hiding statements) on data.

"*Networked RDF Graphs*[2] extend named graphs with a SPARQL based view mechanism. Briefly, a networked graph, among other statements, contains statements describing the graph itself in terms of SPARQL queries against other graphs. The networked graph then contains some explicitly listed content plus statements computed from the views on other graphs."

[1] http://spinrdf.org/

[2] http://www.uni-koblenz-landau.de/koblenz/fb4/institute/IFI/AGStaab/Research/systeme/NetworkedGraphs/

Bibliography

[AB06] Ben Adida and Mark Birbeck. RDFa primer 1.0: Embedding rdf in XHTML. http://www.w3.org/TR/xhtml-rdfa-primer/, accessed 2009-04-02, May 2006. W3C Working Draft.

[ACHZ] K. Alexander, R. Cyganiak, M. Hausenblas, and J. Zhao. void guide – using the vocabulary of interlinked datasets. http://rdfs.org/ns/void-guide, accessed 2009-09-04.

[ADR06] Sören Auer, Sebastian Dietzold, and Thomas Riechert. Ontowiki - a tool for social, semantic collaboration. In *International Semantic Web Conference*, pages 736–749, 2006.

[AH06] Sören Auer and Heinrich Herre. A versioning and evolution framework for RDF knowledge bases. In *Ershov Memorial Conference*, pages 55–69, 2006.

[AH08] Dean Allemang and James Hendler. *Semantic Web for the Working Ontologist: Effective Modeling in RDFS and OWL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[AKD07] Hend S. Al-Khalifa and Hugh C. Davis. Towards better understanding of folksonomic patterns. In *HT '07: Proceedings of the eighteenth conference on Hypertext and hypermedia*, pages 163–166, New York, NY, USA, 2007. ACM.

[Apa08] Apache Software Foundation. *UIMA Overview and SDK Setup*, 2008. Version 2.2.2-incubating.

[B+a] Sean Bechhofer et al. Owl web ontology language: Reference. http://www.w3.org/TR/owl-ref/, accessed 2009-04-02. W3C Recommendation.

[B+b] Diego Berrueta et al. SIOC core ontology specification. http://rdfs.org/sioc/spec/, accessed 2009-06-29.

[BBL08] Dave Beckett and Tim Berners-Lee. Turtle—terse RDF triple language. http://www.w3.org/TeamSubmission/2008/SUBM-turtle-20080114/, 2008.

[BCH07] Chris Bizer, Richard Cyganiak, and Tom Heath. How to publish linked data on the web. http://sites.wiwiss.fu-berlin.de/suhl/bizer/pub/LinkedDataTutorial/20070727/, 2007.

[BCM+03] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[BDE+08] Ansgar Bernardi, Stefan Decker, Ludger van Elst, Gunnar Aastrand Grimnes, Tudor Groza, Siegfried Handschuh, Mehdi Jazayeri, Cédric Mesnage, Knud Müller, Gerald Reif, Michael Sintek, and Leo Sauermann. The social semantic desktop: A new paradigm towards deploying the semantic web on the desktop. In Jorge Cardoso and Miltiadis D. Lytras, editors, *Semantic Web Engineering in the Knowledge Society*, chapter XII, pages 290–312. IGI Global, 2008.

[BG04] Dan Brickley and R.V. Guha. RDF vocabulary description language 1.0: RDF schema. http://www.w3.org/TR/rdf-schema/, accessed 2009-04-02, February 2004. W3C Recommendation.

[BHBL09] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data - the story so far. *International Journal on Semantic Web and Information Systems*, 2009.

[BLa] Tim Berners-Lee. Linked data. http://www.w3.org/DesignIssues/LinkedData, accessed 2009-06-16.

[BLb] Tim Berners-Lee. Notation3 (N3) a readable RDF syntax. http://www.w3.org/DesignIssues/Notation3.html, accessed 2009-04-02.

[BLHL+08] Tim Berners-Lee, J. Hollenbach, Kanghao Lu, J. Presbrey, E. Prud'hommeaux, and Mc Schraefel. Tabulator redux: Browsing and writing linked data. In *Proc. Wsh. Linked Data on the Web (LDOW)*, 2008.

[BLP] Chris Bizer, Ryan Lee, and Emmanuel Pietriga. Fresnel—display vocabulary for RDF—user manual. http://www.w3.org/2005/04/fresnel-info/manual/, accessed 2009-04-02.

[BM] Dan Brickley and Libby Miller. FOAF vocabulary specification. http://xmlns.com/foaf/spec/, accessed 2009-06-29.

[BPKL06] Christian Bizer, Emmanuel Pietriga, David Karger, and Ryan Lee. Fresnel: A browser-independent presentation vocabulary for RDF. In *Proc. 5th Int. Semantic Web Conf. (ISWC)*, 2006.

[BRJ99] Grady Booch, Jim Rumbaugh, and Ivar Jacobsen. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[BVKKS08] Michael Bernstein, Max Van Kleek, David Karger, and M. C. Schraefel. Information scraps: How and why information eludes our personal information management tools. *ACM Trans. Inf. Syst.*, 26(4):1–46, 2008.

[C+] Jeremy Carroll et al. Named graphs. http://www.w3.org/2004/03/trix/, accessed 2009-04-02. W3C Interest Group.

[CDG+06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *Proc. 7th USENIX Symp. Operating Systems Design and Implementation (OSDI)*, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[CSD+08] Richard Cyganiak, Holger Stenzhorn, Renaud Delbru, Stefan Decker, and Giovanni Tummarello. Semantic sitemaps: Efficient and flexible access to datasets on the semantic web. In *Proc. 5th Europ. Semantic Web Conf.* Springer, 2008.

[dBLPF05] Jos de Bruijn, Rubén Lara, Axel Polleres, and Dieter Fensel. OWL DL vs. OWL flight: conceptual modeling and reasoning for the semantic web. In *Proc. 14th int. conf. World Wide Web*, pages 623–632, New York, NY, USA, 2005. ACM.

[dcm] DCMI metadata terms. http://dublincore.org/documents/dcmi-terms/, accessed 2009-06-29.

[DH76] Whitfield Diffie and Martin E. Hellman. Multiuser cryptographic techniques. In *AFIPS National Computer Conference*, pages 109–112, 1976.

[FG96] Eric Freeman and David Gelernter. Lifestreams: a storage model for personal data. *SIGMOD Rec.*, 25(1):80–86, 1996.

[Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[FLGD87] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The vocabulary problem in human-system communication. *Commun. ACM*, 30(11):964–971, 1987.

[Fow04] Martin Fowler. Inversion of control containers and the dependency injection pattern. http://martinfowler.com/articles/injection.html, accessed 2008-09-25, 2004.

[Gar05] Jesse James Garrett. Ajax: A new approach to web applications. http://www.adaptivepath.com/publications/essays/archives/000385.php, accessed 2009-04-02, 2005.

[GH06] Scott A. Golder and Bernardo A. Huberman. Usage patterns of collaborative tagging systems. *J. Inf. Sci.*, 32(2):198–208, 2006.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, 1995.

[Gro08] Object Management Group. Mof query / views / transformations. http://www.omg.org/spec/QVT/1.0/, 2008. Version 1.0.

[gui08] Java on guice: Guice 1.0 user's guide. http://code.google.com/p/google-guice/wiki/UserGuide, accessed 2008-09-25, 2008.

[Hal01] Frank G. Halasz. Reflections on notecards: seven issues for the next generation of hypermedia systems. *ACM J. Comput. Doc.*, 25(3):71–87, 2001.

[Hea09] Tom Heath. Linked data tutorial at semantic web austin. http://linkeddata.org/guides-and-tutorials, 2009.

[Hen06] Jim Hendler. The dark side of the semantic web. http://www.mindswap.org/blog/2006/12/13/the-dark-side-of-the-semantic-web/, 2006.

[HKM04] N.F. Noy H. Knublauch, R. Fergerson and M.A. Musen. The Protege OWL Plugin: An Open Development Environment for Semantic Web Applications. In 3rd *Int. Conf. Semantic Web (ISWC)*, 2004.

[HM76] J. W. Hunt and M. D. McIlroy. An algorithm for differential file comparison. Technical Report CSTR 41, Bell Laboratories, Murray Hill, NJ, 1976.

[HPSB$^+$] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: A semantic web rule language combining OWL and RuleML. http://www.w3.org/Submission/SWRL/, accessed 2009-06-26.

[HvOH06] Michiel Hildebrand, Jacco van Ossenbruggen, and Lynda Hardman. /facet: A browser for heterogeneous semantic web repositories. In *International Semantic Web Conference*, pages 272–285, 2006.

[HZL08] Philipp Heim, Jürgen Ziegler, and Steffen Lohmann. gfacet: A browser for the web of data. In S. Auer, S. Dietzold, S. Lohmann, and J. Ziegler, editors, *Proc. int. wsh. interacting with multimedia content in the social semantic web (IMC-SSW)*. CEUR-WS, 2008.

[iso07] Common logic (cl): a framework for a family of logic-based languages. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=39175, 2007. ISO/IEC 24707:2007.

[JSB$^+$08] Simon Jupp, Robert Stevens, Sean Bechhofer, Yeliz Yesilada, and Patty Kostkova. Knowledge representation for web navigation. In *Semantic Web Applications and Tools for the Life Sciences (SWAT4LS 2008) Workshop*, 2008.

[Kuh08] Tobias Kuhn. AceWiki: Collaborative ontology management in controlled natural language. In *Proc. 3rd Wsh. Semantic Wikis*. CEUR Workshop Proceedings, 2008.

[KVV$^+$07] Markus Krötzsch, Denny Vrandečić, Max Völkel, Heiko Haller, and Rudi Studer. Semantic wikipedia. *Web Semant.*, 5(4):251–261, 2007.

[Mag] Patrick T. Magee. Three step creative writing process. http://braindance.com/bdimmap4.htm, accessed 2009-04-02.

[MAK+04] Martin Michalowski, José Luis Ambite, Craig A. Knoblock, Steve Minton, Snehal Thakkar, and Rattapoom Tuchinda. Retrieving and semantically integrating heterogeneous data from the web. *IEEE Intelligent Systems*, 19(3):72–79, 2004.

[Mar06] Gary Marchionini. Exploratory search: from finding to understanding. *Commun. ACM*, 49(4):41–46, 2006.

[MB] Alistair Miles and Sean Bechhofer. SKOS simple knowledge organization system reference. http://www.w3.org/TR/skos-reference/, accessed 2009-06-29.

[McA90] David A. McAllester. Truth maintenance. In *AAAI*, pages 1109–1116, 1990.

[MHRJ91] Catherine C. Marshall, Frank G. Halasz, Russell A. Rogers, and William C. Janssen, Jr. Aquanet: a hypertext tool to hold your knowledge in place. In *Proc. 3rd ACM Conf. Hypertext*, pages 261–275, New York, NY, USA, 1991. ACM.

[MHS06] Eetu Mäkelä, Eero Hyvönen, and Samppa Saarela. Ontogator - a semantic view-based search engine service for web applications. In *International Semantic Web Conference*, pages 847–860, 2006.

[MHS09] Boris Motik, Ian Horrocks, and Ulrike Sattler. Bridging the gap between OWL and relational databases. *Web Semant.*, 7(2):74–89, 2009.

[MM] Frank Manola and Eric Miller. RDF primer. http://www.w3.org/TR/rdf-primer/, accessed 2009-04-02. W3C Recommendation.

[MTEBG07] Christian Morbidoni, Giovanni Tummarello, Orri Erling, and Reto Bachmann-Gmür. RDFSync: efficient remote synchronization of RDF models. In *Proc. 6th Int. and 2nd Asian Semantic Web Conf. (ISWC+ASWC)*, pages 533–546, November 2007.

[New] Richard Newman. Tag ontology design. http://www.holygoat.co.uk/projects/tags/, accessed 2009-06-19.

[NR] Natasha Noy and Alan Rector. Defining n-ary relations on the semantic web. http://www.w3.org/TR/swbp-n-aryRelations, accessed 2009-05-21.

[ODD06] Eyal Oren, Renaud Delbru, and Stefan Decker. Extending faceted navigation for RDF data. In *Proc. Int. Semantic Web Conf. (ISWC)*. Springer, 2006.

[ODM+06] Eyal Oren, Renaud Delbru, Knud Möller, Max Völkel, and Siegfried Handschuh. Annotation and navigation in semantic wikis. In *Proc. 1st Wsh. Semantic Wikis*, 2006.

[omg09] Ontology definition metamodel. http://www.omg.org/spec/ODM/1.0, 2009. Version 1.0.

[PENN07] Matthias Palmér, Fredrik Enoksson, Mikael Nilsson, and Ambjörn Naeve. Annotation profiles: configuring forms to edit RDF. In *DCMI '07: Proceedings of the 2007 international conference on Dublin Core and Metadata Applications*, pages 10–21. Dublin Core Metadata Initiative, 2007.

[PH03] J. Park and S. Hunting, editors. *XML Topic Maps*. Addison-Wesley, 2003.

[Pie] Emmanuel Pietriga. *Fresnel Selector Language for RDF (FSL)*.

[Pro] The LaTeX Project. Latex – a document preparation system. http://www.latex-project.org/, accessed 2009-04-02.

[PS05] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. http://www.w3.org/TR/rdf-sparql-query/, accessed 2009-04-02, July 2005. W3C Working Draft.

[PSHH04] Peter F. Patel-Schneider, Patrick Hayes, and Ian Horrocks. OWL web ontology language: Semantics and abstract syntax. http://www.w3.org/TR/2004/REC-owl-semantics-20040210/, 2004.

[PV04] Benjamin C. Pierce and Jérôme Vouillon. What's in Unison? A formal specification and reference implementation of a file synchronizer. Technical Report MS-CIS-03-36, Dept. of Computer and Information Science, University of Pennsylvania, 2004.

[QHK03] Dennis Quan, David Huynh, and David R. Karger. Haystack: A platform for authoring end user semantic web applications. In *International Semantic Web Conference (ISWC)*, pages 738–753, 2003.

[Qui05] Emanuele Quintarelli. Folksonomies: power to the people. http://www.iskoi.org/doc/folksonomies.htm, June 2005. Presented at the ISKO Italy-UniMIB meeting.

[RAN07] Axel Rauschmayer, Anita Andonova, and Patrick Nepper. Increasing the versatility of Java documentation with RDF. In K. Tochtermann, W. Haas, F. Kappe, A. Scharl, T. Pellegrini, and S. Schaffert, editors, *Proc. Int. Conf. Semantic Technologies (I-SEMANTICS)*, J.UCS. Graz University of Technology, 2007.

[Rau] Axel Rauschmayer. *Hyena manual*. http://www.pst.ifi.lmu.de/~rauschma/hyena/manual/.

[Rau04] Axel Rauschmayer. A recipe for more dynamic OOP: Mix a knowledge representation and prototypes. In *Proc. OOPSLA Wsh. Revival of Dynamic Languages*, October 2004.

[Rau05a] Axel Rauschmayer. An RDF editing platform for software engineering. In *ISWC Wsh. Semantic Web Enabled Software Engineering (SWESE)*, November 2005.

[Rau05b] Axel Rauschmayer. An RDF editing platform for software engineering. Technical Report 0505, Ludwig-Maximilians-Universität München, Institut für Informatik, July 2005.

[Rau05c] Axel Rauschmayer. Semantic-web-backed gui applications. In *ISWC Wsh. End User Semantic Web Interaction*. M. Jeusfeld c/o Redaktion Sun SITE, Informatik V, RWTH Aachen, November 2005.

[Rau08a] Axel Rauschmayer. Lightweight data modeling in RDF. In Jörg Rech, Björn Decker, and Eric Ras, editors, *Emerging Technologies for Semantic Work Environments: Techniques, Methods, and Applications*. Idea Group Inc., 2008.

[Rau08b] Axel Rauschmayer. Next-generation wikis: What users expect; how RDF helps. In Christoph Lange, Sebastian Schaffert, Hala Skaf-Molli, and Max Völkel, editors, *Proc. 3rd Semantic Wiki Wsh. at ESWC*. M. Jeusfeld c/o Redaktion Sun SITE, Informatik V, RWTH Aachen, 2008.

[Rau10] Axel Rauschmayer. Structure your wiki: Improving support for structured data in wikis. Technical Report 1002, Ludwig-Maximilians-Universität München, Institut für Informatik, 2010.

[rdf] RDFa bookmarklets. http://www.w3.org/2001/sw/BestPractices/HTML/rdfa-bookmarklet/, accessed 2009-04-02.

[RK06] Axel Rauschmayer and Walter Christian Kammergruber. A wiki as an extensible RDF presentation engine. In *ESWC Wsh. Semantic Wikis—From Wiki to Semantics*. M. Jeusfeld c/o Redaktion Sun SITE, Informatik V, RWTH Aachen, June 2006.

[RR04a] Axel Rauschmayer and Patrick Renner. Knowledge-representation-based software engineering. Technical Report 0407, Ludwig-Maximilians-Universität München, Institut für Informatik, May 2004.

[RR04b] Axel Rauschmayer and Patrick Renner. Tube: Interactive model-integrated object-oriented programming. In *Proc. IASTED Int. Conf. Software Engineering and Applications (SEA)*, November 2004.

[RR05a] Axel Rauschmayer and Patrick Renner. Tube—structure-orientation in a prototype-based programming environment. In *Proc. Int. Conf. Programming Languages and Compilers (PLC)*, June 2005.

[RR05b] Axel Rauschmayer and Patrick Renner. Tube: a prototype-based programming environment. Technical Report 0502, Ludwig-Maximilians-Universität München, Institut für Informatik, April 2005.

[Rul] The rule markup initiative. http://www.ruleml.org/, accessed 2009-04-02.

[SC] Leo Sauermann and Richard Cyganiak. Cool uris for the semantic web. http://www.w3.org/TR/2008/NOTE-cooluris-20081203/. W3C Interest Group Note.

[SEG+09] Sebastian Schaffert, Julia Eder, Szaby Grünwald, Thomas Kurz, Mihai Radulescu, Rolf Sint, and Stephanie Stroka. Kiwi – a platform for semantic social software. In *Proc. 4th Wsh. Semantic Wikis*, 2009.

[SESH07] Michael Sintek, Ludger Elst, Simon Scerri, and Siegfried Handschuh. Distributed knowledge representation on the social semantic desktop: Named graphs, views and roles in nrl. In *Proc. 4th European Conf. The Semantic Web*, pages 594–608, Berlin, Heidelberg, 2007. Springer-Verlag.

[SHJJ09] Henry Story, Bruno Harbulot, Ian Jacobi, and Mike Jones. FOAF+SSL: RESTful Authentication for the Social Web. In Michael Hausenblas, Philipp Kärger, Daniel Olmedilla, Alexandre Passant, and Axel Polleres, editors, *Proc. 1st Wsh. Trust and Privacy on the Social and Semantic Web (SPOT)*. CEUR-WS.org, 2009.

[sit] Microformats. http://microformats.org/, accessed 2009-04-02.

[SMB+08] Andy Seaborne, Geetha Manjunath, Chris Bizer, John Breslin, Souripriya Das, Ian Davis, Steve Harris, Kingsley Idehen, Olivier Corby, Kjetil Kjernsmo, and Benjamin Nowack. Sparql update: A language for updating RDF graphs. http://www.w3.org/Submission/2008/SUBM-SPARQL-Update-20080715/, 2008.

[TB08] Michal Tvarožek and Mária Bieliková. Collaborative multi-paradigm exploratory search. In *WebScience '08: Proceedings of the hypertext 2008 workshop on Collaboration and collective intelligence*, pages 29–33, New York, NY, USA, 2008. ACM.

[TM08] Giovanni Tummarello and Christian Morbidoni. The dbin platform: A complete environment for semantic web communities. *Web Semantics: Science, Services and Agents on the World Wide Web*, 6(4):257 – 265, 2008.

[Tun06] Daniel Tunkelang. Dynamic category sets: An approach for faceted search. In *Proc. SIGIR Wsh. Faceted Search*. ACM, 2006.

[VG06] Max Völkel and Tudor Groza. Semversion: An RDF-based ontology versioning system. In *Proc. IADIS Int. Conf. WWW/Internet*, volume 1, pages 195–202, Murcia, Spain, OCT 2006. IADIS, IADIS.

[Wel07] Katrin Weller. Folksonomies and ontologies: Two new players in indexing and knowledge representation. In H. Jezzard, editor, *Applying Web 2.0: Innovation, Impact and Implementation. Proc. Online Information Conference*. Learned Information Europe Ltd., 2007.

[wika] Creole: A common wiki markup language. http://wikicreole.org/, accessed 2009-04-02.

[Wikb] Wikipedia. RDFa. http://en.wikipedia.org/wiki/RDFa, accessed 2009-04-02.

[WP07] Katrin Weller and Isabella Peters. Reconsidering relationships for knowledge representation. In *Proc. I-KNOW*, 2007.

[YCM06] Alexander Yip, Benjie Chen, and Robert Morris. Pastwatch: a distributed version control system. In *Proc. 3rd Conf. Networked Systems Design & Implementation (NSDI)*, pages 28–28, Berkeley, CA, USA, 2006. USENIX Association.

[ZR04] Christian Zimmer and Axel Rauschmayer. Tuna: Ontology-based source code navigation and annotation. In *OOPSLA Wsh. Ontologies as Software Engineering Artifacts*, October 2004.

Acknowledgements

I would like to thank the following people who accompanied my professional life while I was writing this dissertation.

- Thanks to my professor, Martin Wirsing, for having provided me with a great job and work environment for several years.
- I would like to thank the primary reviewers of this dissertation: Martin Wirsing, Marcus Spies, and Don Batory. Their comments and their support were very helpful.
- I would also like to thank my third and fourth reviewers, Claudia Linnhoff-Popien and Christian Böhm for their time.
- The whole chair of Programming and Software Engineering was an incredible source of information and inspiration. To name four important people among many: Martin Wirsing (Grundlagen der Systementwicklung, process calculi), Alexander Knapp (calculi, "how academia works", history), Hubert Baumeister (Wikis, eXtreme Programming), Matthias Hölzl (Dylan, Lisp, programming language design).
- I was fortunate to have many students collaborate with me on HYENA. This lead to many fruitful discussions. Among others: Patrick Nepper, Christian Zimmer, Thomas Müller, Anita Andonova, and Philipp Mpalampanis.
- Discussions with and tips from Malte Kiesel concerning wikis and the semantic web were also highly instructional.
- Matthias Palmér provided valuable feedback on drafts of the REMM chapters.
- Michael Hausenblas wrote the wiki page on which the sections of write-enabling linked data are based and answered questions I had.
- Helpful feedback on the wiki survey questions was given by Malte Kiesel, Andreas Schroeder, Philip Mayer, and Hubert Baumeister.
- Finally, members of the mailing list of the "Linking Open Data Project" patiently answered my questions, helping me with understanding linked data principles and ideas.