Integrated Project

Priority 2.4.7

Semantic based knowledge systems

Information Society Technologies

The Social Semantic Desktop

NEPOMUK

Conceptual Data Structures (CDS) Tools

Deliverable D1.2

Version 1.0

15.01.2008

Dissemination level: PP

Nature	P
Due date	15.01.2008
Lead contractor	FORSCHUNGSZENTRUM INFORMATIK AN DER UNIVERSITÄT KARSRUHE
Start date of project	01.01.2006
Duration	36 months

Authors

Max Völkel, FZI
Heiko Haller, FZI
William Bolinder, KTH
Brian Davis, NUIG
Henrik Edlund, KTH
Kristina Groth, KTH
Rósa Gudjónsdóttir, KTH
Mikhail Kotelnikov, COG
Pär Lannerö, KTH
Sinna Lindquist, KTH
Mikhail Sogrin, IBM
Yngve Sundblad, KTH
Bosse Westerlund KTH

Mentors

Ansgar Bernardi, DFKI
Mehdi Jazayeri, USI

Project Co-ordinator

Dr. Ansgar Bernardi
German Research Center for Artificial Intelligence (DFKI) GmbH
Trippstadter Str. 122
67663 Kaiserslautern
Germany
E-Mail: bernardi@dfki.uni-kl.de, phone: +49 631 205 75 105

Partners

DEUTSCHES FORSCHUNGSZENTRUM F. KUENSTLICHE INTELLIGENZ GMBH
IBM IRELAND PRODUCT DISTRIBUTION LIMITED
SAP AG
HEWLETT PACKARD GALWAY LTD
THALES S.A.
PRC GROUP - THE MANAGEMENT HOUSE S.A.
EDGE-IT S.A.R.L
COGNIUM SYSTEMS S.A.
NATIONAL UNIVERSITY OF IRELAND, GALWAY
ECOLE POLYTECHNIQUE FEDERALE DE LAUSANNE
FORSCHUNGSZENTRUM INFORMATIK AN DER UNIVERSITAET KARLSRUHE
UNIVERSITAET HANNOVER
INSTITUTE OF COMMUNICATION AND COMPUTER SYSTEMS
KUNGLIGA TEKNISKA HOEGSKOLAN
UNIVERSITA DELLA SVIZZERA ITALIANA
IRION MANAGEMENT CONSULTING GMBH

Versions

Version	Date	Reason
0.1	17.08.2007	Start of first draft by Max Völkel
0.2	15.11.2007	First draft by Max Völkel and Heiko Haller
0.3	07.01.2008	Second version presented to mentors
1.0	15.01.2008	Final version

Explanations of abbreviations on front page

Nature
R: Report
P: Prototype
R/P: Report and Prototype
O: Other

Dissemination level
PU: Public
PP: Restricted to other FP6 participants
RE: Restricted to specified group
CO: Confidential, only for Nepomuk partners

Executive Summary

In this deliverable we present the idea of Conceptual Data Structures (CDS), a unifying data model for end-user semantic personal knowledge management. We present concepts as well as a software framework.

We describe two main user interface prototypes based on the CDS framework and their evaluations. First, the NEPOMUK hypertext knowledge workbench (HKW) is described. Second we describe the visual knowledge workbench (VKW), consisting of two parts: iMapping and QuiKey. iMapping is graphical UI concept for personal knowledge mapping, QuiKey is a "visual command line" for knowledge articulation and querying and browsing. All prototypes are based on the same CDS API and are hence interoperable. The CDS API is integrated with the NEPOMUK RDF repository. We describe the current and planned integration status.

Additionally, we describe some natural language processing tools and a WikiModel parser framework developed in our workpackage and how they integrate with CDS.

Contents

1 Introduction

Everyday knowledge artefacts from scrap notes to books, from mind maps to websites all use certain very common structures that can be summarised in a few very general relation types between information elements. It is the core idea of the CDS approach, to identify these "Conceptual Data Structures" found by analysis of common knowledge media (Sec. 2.4. They are to serve as a guideline for knowledge workbench tools as well as a common high-level vocabulary that bridges the gap between informally structured personal notes and formal representations of knowledge like ontologies.

In this document we describe CDS as a generic framework for the representation and management of personal knowledge (Sec. 2). CDS offers a set of basic entities and common relations which proved suitable to cover the characteristics of common information tools and structures in every-day use. We describe the underlying data model SWCM (Sec. 2.3) and its realisation on top of RDF (Sec. 2.6).

We developed and evaluated several prototypical user interfaces which support the creation of and interaction with CDS, representing knowledge as a highly interrelated set of small items. They follow two different approaches: The Hypertext-based Knowledge Workbench (Sec. 3) is inspired by the semantic wiki approach. It is a web-interface to a CDS-Model, which can be seen as a semantic wiki with very fine granularity. The Visual Knowledge Workbench (Sec. 4) mainly consists of the iMapping client for visually authoring CDS models. It is inspired by visual knowledge mapping techniques like mind-maps and targeted to allow intuitive authoring of personal knowledge bases. The iMapping approach is based on design principles derived from research in user interface-design and cognitive psychology. We have so far created and evaluated two early prototypes of the iMapping client (Sections 4.1.5 and 4.1.6). The lessons learnt hereby have led to the design of the currently developed full iMapping prototype described in 4.1. It is complemented by the QuiKey tool, a kind of smart semantic command-line that focuses on highest interaction efficiency to browse, query and author CDS Models.

Apart from explicitly entering all formal structures manually, Natural Language Processing (NLP) can be used on plain text like wiki pages, e-mails or external content to extract structures and statements from or to propose related items and keywords. We describe some of the NLP tools developed in WP1000 and how they integrate with CDS in Sec. 5.

Finally, we describe how CDS relates to semantic wikis in a twofold way: by originating from a semantic wiki approach and by how a wiki-like syntax can be used to formulate CDS.

2 Conceptual Data Structures (CDS)

In brief, Conceptual Data Structures (CDS) are a lean model suitable for representing and using personal knowledge in various degrees of formalisation in a uniform fashion, allowing step-wise formalisation.

CDS consists of two layers: A data model layer (called SWCM) and an ontology layer (called CDS ontology). In the remainder of this section we present these two layers, the rationale for the ontology layer and the realisation of a CDS API. The technical details of SWCM have been presented at I-Semantics in Völkel (2007). The core ideas of CDS have been presented in Völkel, Haller & Abecker (2007) and Völkel & Haller (2006).

As a research topic, CDS tackles Personal Knowledge Management (PKM), which we introduce now briefly.

2.1 Personal Knowledge Management (PKM)

In brief, PKM can be seen as the Knowledge Management (KM) perspective on Personal Information Management (PIM) or the personal perspective on KM.

Knowledge Management (KM) In 2000, the European Commission issued the Lisbon Strategy[1] to stimulate economic growth. The first one of three pillars is

> preparing the ground for the transition to a competitive, dynamic, knowledge-based economy. Emphasis is placed on the need to adapt constantly to changes in the information society and to boost research and development.

As our society becomes more knowledge-intensive future progress depends on efficient and effective ways to automate knowledge processing.

Probst, Raub & Romhardt (2006) defines KM with six core and two control processes: (1) identification, (2) acquisition, (3) development, (4) sharing, (5) application, and (6) storage of knowledge . The control processes are *knowledge goal definition* and *evaluation of goal achievement*.

Personal Information Management (PIM) PIM usually means managing contact information, appointments, to-do items and notes. In fact, CDS grew out of frustration encountered with the existing PIM tool-scape and the inability to relate, link and describe PIM items. For example, one can currently not even relate a number of contacts, appointments and tasks to a project. In particular, plain *text notes* are in most PIM tools merely an unstructured, unrelated set of memo items.

The basic processes in PIM have been identified (Jones & Bruce 2005) as: (1) keeping; (2) finding/re-finding and (3) meta-activities like mapping between information and need, maintenance and organisation .

Jones, Bruce & Dumais (2001) introduces the problem of "keeping found things found" which reports on the tension between *knowing* something and merely *storing* something.

Another core problem of PIM could be paraphrased as "keep found things accessible". Marshall (2007) describes how hard it is to keep just all emails over a time span of five years.

Personal Knowledge Management (PKM) In 1958, Peter F. Drucker (Drucker 1985) was among the first to use the term *knowledge worker* for someone who

[1] `http://europa.eu/scadplus/glossary/lisbon_strategy_en.htm`

works primarily with information or one who develops and uses knowledge in the workplace.

> The most important contribution of management in the 20th century was to increase manual worker productivity fifty-fold. The most important contribution of management in the 21st century will be to increase knowledge worker productivity – hopefully by the same percentage. [...] The methods, however, are totally different from those that increased the productivity of manual workers.

Management of personal information and personal knowledge becomes more important as more people work as knowledge workers, where their capital is their knowledge.

The term personal knowledge has also been used in Polanyi (1958), reprinted in Polanyi (1998). Frand & Hixon (1999) was among the first to use the term PKM in an academic context, followed by Avery, Brooks, Brown, Dorsey & O'Conner (2001), Mitchell (2005).

North (2007) defines knowledge work as "work based on knowledge with an immaterial result; value creation is based on processing, generating and communicating knowledge.[2]" According to North (2007), core value creation processes in knowledge work are (1) searching, analysing, structuring, and reflecting; (2) planning, strategy development, and organisation; (3) learning; (4) communication and documentation; (5) combination, reconfiguration, designing, and synthesizing.

CDS help mainly with step (1).

Conceptual Data Structures (CDS) CDS are a framework for PKM. In particular, they can be seen as a building block for *semantic* personal knowledge management (SPKM) tools. SPKM (Oren, Völkel, Breslin & Decker 2006) is PKM supported by semantic technologies.

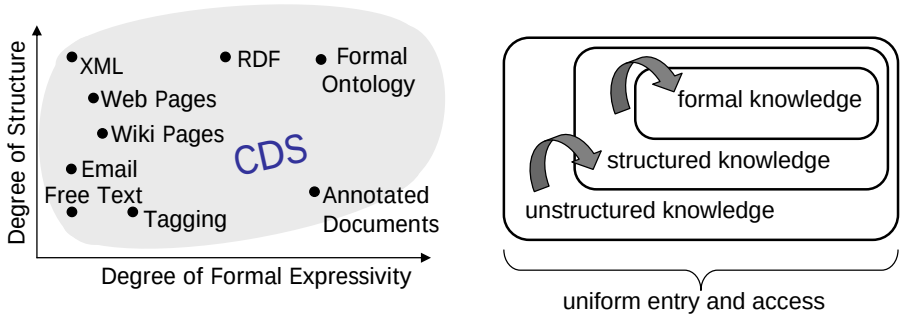

Figure 1: CDS unifies different levels of formality

PKM requires the uniform management of unstructured, structured and formal knowledge We assume that a typical user has the majority of the content in unstructured form, some content will be a least structured and only few content will be fully formalised, as depicted in Fig. 1, right side. We expect this distribution simply for cost-benefit-reasons: It would be too costly to structure or formalise all content. On the other hand, formalising *some* content lets the user profit e. g. from inferred content types. So when content is typed with "Researcher", a search for "Person" would also return it – if the relation between researcher and person has been formally defined as `cds:hasSubType`.

One can distinguish between domain-specific data models such as the data model of e. g. Microsoft Outlook, which limits the user to speak about persons and their addresses. There is neither a way to state the relation to other *persons*, nor to represent music collections or relations from persons to other *objects*. This is

[2] translation by Vi¿ $\frac{1}{2}$lkel

not only a limit of PIM-tools but of any domain-specific tool: They support data modelling in a given domain well but do not allow to extend the model or link to other objects in other data models. Domain-free data models such as the file system or the model behind mind map applications allow to model *any* domain – but not in a structured way. It is, e. g. not possible to export a set of persons created in e. g. a Mind Mapping application to an address book application. CDS is intended to unify popular domain-free data models and allow at the same time to represent structured, domain-specific data in such a way that domain-specific semantics can be used for inference, search and export.

Wikis and concept maps are two examples of popular, domain-free tools. CDS is based on concepts from wikis (e g. easy editing and browsing), concept maps (e. g. typed links) and semantic web technologies (e. g. formal statements, globally addressable entities). Wikis offer better linking abilities but come at the cost of not being able to represent specific types of items, such as appointment or contact data. Semantic wikis meet both requirements but are still to cumbersome to use, i. e. it is hard to get an overview of the emerging structure and refactoring is very costly.

CDS is fusing the two concepts (informal knowledge tools and formal ontologies) and shifting the focus from documents to knowledge models. The goal of CDS is to handle a mix of knowledge in different degrees of formality (c.,f. Fig. 1, left side) and allow *stepwise migration* from less structure to more structure and more formality.

2.2 Requirements

CDS was not derived directly from cognitive or psychological works. Instead, an indirect approach was chosen in which existing user interfaces, data structures, and document models have been analysed.

The data sources are selected to represent structures which humans are used to use. This deliverable tries not to model how humans think or which structures would cognitively be the best choice. The approach is instead to consolidate existing structures under the assumption that those have been "proven" to work. A system using a model resembling existing structures should at least be easy to get used to.

The steps undertaken to arrive at the CDS are: (a) select a number of categories (Sec. 2.2.1), (b) in each category choose the most popular instances (Sec. 2.2.2), (c) analyse data model structures and features (Sec. 2.2.3), (d) find common relations and create a subsumption hierarchy of relations (Sec. 2.4.1).

We found some structures to be inherent to most knowledge artefacts ranging from vague paper notes to highly structured documents. The next sections present the analysis steps in depth.

2.2.1 Relevant Categories

What approaches are used in practice to organise information and knowledge? What approaches deal with the representation of vague knowledge, allow stepwise formalisation or expose a modelling language to and end-user? A comprehensive list is hard to define precisely, yet we believe the following category list is broad enough, to cover all relevant (vague knowledge, stepwise formalisation, exposing modelling language) approaches used in practice to organise information and knowledge:

- note-taking,
- documents,
- hypertext

- desktop information management tools,
- data structures in programming languages
- data exchange formats and content languages
- creativity tools,
- more advanced PKM tools
- web sites for collaborative information organisation
- knowledge representation languages
- argumentation tools

2.2.2 Most Popular Instances for each Category

In this section we select the most popular or otherwise relevant approaches of each category.

Note-Taking Popular examples of notes on paper are to-do lists, shopping lists, diaries, and lab books.

Documents Documents exist on paper as well as in electronic form. The conceptual model behind them is the same in both cases.

Hypertext By far the most prominent example of hypertext in use is the *World Wide Web*. Before the broad use of WWW there were a number of more sophisticated hypertext systems that might be harder to deploy on a global scale but which might also contain a number of interesting concepts for PKM. Therefore this deliverable looks also into the *Xanadu* data model.

Desktop Information Management Tools The most popular tool for desktop information management is probably the Microsoft Windows File Explorer. Similar file browsers exists for less popular operating systems.

Implicitly the structure of the file system is exposed via these tools.

Data Structures in Programming Languages A popular all-purpose programming language is Java, hence we look into the *Java Collection Framework*, which is the foundation for managing collections of programming objects. Additionally, we look into Python, a scripting language with dynamic typing, and its built-in handling of data structures. The assumption is that structures that are deeply built into a programming language worked well in the past to moderate between humans and computers.

Data Exchange Formats and Content Languages The *structures* used in data exchange languages must work for all use cases of that data format. Many parts in a data format vary, but some structures cannot be changed. This deliverable analyses theses built-in structures. In particular, XML and RDF are analysed.

Due to the high popularity of the web, the structures built into HTML are also worth looking into.

Creativity Tools Popular creativity tools are mind-maps and concept maps. Both are popular graphical approaches for structuring information and helping the user to get an overview. Mind maps can also be used to present information to others.

The most popular mind mapping tool is *Mind Manager* from *MindJet*. For concept maps, the most popular tool is the *CMap Tools* suite, which recently became a commercial product.

More Advanced PKM Tools Research prototypes for more advanced PIM and PKM systems exist. Among the more popular ones are Haystack Quan, Huynh & Karger (2003) and Chandler [3].

Web Sites For Collaborative Information Organisation With the advent of "Web 2.0", some web site dedicated to collaborative information management became popular. Among these, *del.icio.us*, for organising bookmarks and *flickr*, for organising pictures have been chosen for further analysis. Both have been bought by Yahoo.

Knowledge Representation Languages The personal knowledge model is intended to represent formal knowledge. Therefore an investigation of existing knowledge representation languages is useful¡ $\frac{1}{2}$.

This deliverable looks into OWL – RDF is handled already as a data exchange format and Cyc – because the Cyc system has over 15,000 different types of links. Furthermore, this deliverable analyses RDFS, Topic Maps, Conceptual Graphs and SKOS.

Argumentation Tools Argumentation is used to achieve a consensus among diverging opinions, e. g. in court or expert discussions. Established forms of argumentation are pro and contra arguments, based on verifiable evidence. A formalisation of the basic structures has been done by Kunz & Rittel (1970) as *IBIS*, issue-based information systems.

2.2.3 Analysis of Data Models and their Relation Types

In this section we look at popular tools within each category and distill the essentials of their data model concepts.

Note-Taking People tend to use their own notation in paper notes.

There are very few published studies on the structure of personal notes. A study by Dienel (2006) examines personal note books used by engineer around 1850 and later. The study concludes that typical entry types were ideas, projects, addresses, to-do lists, meeting minutes, data from measurements, and appointments. Most engineers used one or two diaries at the same time, one for factual knowledge and the other more for process knowledge. The study also remarks the prominent use of different colors and carefully added table of contents. The artefacts in the notebooks are either text snippets belonging to one of the given types, or drawings or tables.

Features of paper-based note books which are requirements for CDS:

- Do not restrict the user to a domain. ($\mapsto$ Req. 1)
- Let the user work on the same knowledge model for several years without loosing structure ($\mapsto$ Req. 2)
- Can use text and images ($\mapsto$ Req. 3)

Documents A document contains a number of knowledge items. This act of "packaging" together a set of knowledge items influences the interpretation of each item by the reader. A document is a knowledge artefact consisting of several layers.

A document has a visual structure, i. e. is not only a stream of sentences, but uses type-setting, i.e. bold, italics, different font styles and size, and placement of figures. Instead of focusing on the visual properties of documents such as distribution of content on printed pages, this deliverable looks at the logical structure encoded by visual properties.

Features of information in a document are:

[3] `http://www.osafoundation.org/`

Reference-ability : Once a document is published, the reference can act as a placeholder for the content expressed within. A reference to a document can act as a meta-symbol on top of the symbols (information atoms) the document contains. The usage of document references as symbols allows a document to "participate" in conversations, which lead to scholastic methods and modern academia. (↦ Req. 4)

Metadata : Each document is written by a number of authors for a certain audience with a certain goal. By sending this process metadata along with the document the reader has the ability to put the document in context and interpret it better. Such metadata is used by the reader as a frame of reference for interpretation and for search. (↦ Req. 5)

Sequential Navigation : A document can typically be read from start to end by navigating through all contained information items. This is a simple yet effective strategy to scan completely over a body of information. However, during document creation, the final order is most likely not yet defined. Ordering a collection of ideas or text snippets into a coherent flow is one of the main tasks of authoring (Esselborn-Krumbiegel 2002). (↦ Req. 6).

A user should be able to create order gradually, e. g. by stating order between some sections, but not requiring a total ordering. (↦ Req. 7)

Logical structure : The visual structure is used to encode a logical structure consisting of i.e. paragraphs, headlines, footnotes, citations, and title. The logical structure makes it possible to reference smaller, meaningful parts within a document, i.e. "Sec. 4.2". This logical structure most importantly consists of a hierarchy (↦ Req. 8) of text parts such as book part, chapter, section, sub-section, paragraphs, sub-paragraph and sentence. This hierarchy, together with a linear order (Req. 6) creates a tree. The user can navigate e. g. from section up to chapter, down to sub-section, forward to next section, or backwards to previous section.

Argumentative structure : On top of the linear content, a document follows an argumentative structure to convey its content to the reader. Argumentative structures appear on all scales. A typical structure is the "Introduction - Related work - Contribution - Conclusion"-pattern of scientific articles. On smaller scales, patterns like "claim-proof" and "question-answer" are used. (↦ Req. 9)

Content semantics : Documents content's mean something. Building upon logical and argumentative structure, the author encodes statements about a domain within the content. CDS should allow to represent a documents semantic content in a formal way. (↦ Req. 10)

Documents are an established form of communication, carrying a lot of structure to be exploited.

Hypertext The main concept in hypertext and the WWW is the notion of a *hyper-link* which moves the focus away from the document the user is looking at towards another document. Hyperlinks in WWW are directed, have a single source and a single target. (↦ Req. 11)

A central idea in another hypertext system called *Xanadu* is the notion of *transclusion*, as defined in Nelson (1995):

> The central idea has always been what I now call *transclusion*, or *reuse with original contexts available*, through embedded shared instances (rather than duplicate bytes).

Transclusion allows to use a piece of content inside many document. If the content piece is changed, it changes in all documents that use it. (↦ Req. 12)

Desktop Information Management Tools In 1981, the Xerox Star Workstation, one of the first personal computers, was released (Friedewald 2000). It pioneered the WIMP-metaphor (window, icon, menu, pointing device) and placed digital documents, represented as little icons, in the heart of the user interaction. Files in the computer were modelled close to physical documents. Since then, documents remained the dominant paradigm for information exchange and archival.

The popular operating systems today (Windows, Mac OS, Linux) still follow the WIMP-metaphor. All of them contain a desktop with icons and a file browser (Windows: Explorer, Max OS: Finder, Linux: e. g. Konqueror). The file browser basically shows a strict hierarchy of directory names and the files contained therein. File and directory properties such as creation date, size and access rights are also shown in this tool. Users can browse a tree (Req. 6, Req. 8) and thus narrow down their search and discover related, yet unexpected items. It is important for a user to be able to group seemingly unrelated content ($\mapsto$ Req. 13) together so that retrieval of one item triggers retrieving of the others, too (Jones, Phuwanartnurak, Gill & Bruce 2005). It should be easy to place new items into a named container ($\mapsto$ Req. 14).

Data Structures In Programming Languages Popular programming languages have built-in support for certain data structures, while others are built by composing the simpler ones.

- *Python* has built-in support for arrays (lists), sets and maps (dictionary).
- *Java* has only native support for arrays. The included *Java Collections Framework* adds support for sets, lists and maps.

The relation types needed to represent these data structures are:

Unordered collection Does not imply or require an order among items. This is the same as Req. 13.

Lists Membership in a collection (Req. 13) plus a total order (Req. 6) among its elements.

Maps A map is a collection of items which has for each item a defined corresponding item. The functionality of a map can thus be realised via Req. 6 and Req. 11.

Data Exchange Formats For data exchange, XML is probably one of the most popular languages today. The data model used in XML is the so-called XML infoset. This is a strict tree of elements. Elements may contain text and a number of attribute-value pairs. Basically XML encodes a labelled tree with ordered children.

Representing XML requires hierarchical nesting of elements (Req. 8), unique names for the elements and attributes (Req. 14), and modelling of attributes. An attribute is basically a key-value-pair connected to an element. As an example, the image element `IMG` of XHTML has an attribute `SRC` to state the source URL of an image. The attribute can modelled as *part-of* the element, a case of hierarchical nesting (Req. 8). Furthermore, the key of the attribute denotes the semantics of the attribute. In fact, not the complete key-value-pair is part of the image, only the URL value is. The key is only stating the role or type the value plays. More formally, the semantics of

```
<element key="value"> ... </element>
```

can be modelled as three formal statements: (element, hasPart, x), (x, hasValue, "value"), (x, hasType, key). As requirements we get the need for representing part-whole-hierarchies (Req. 8) and for assigning types to items ($\mapsto$ Req. 15).

A new data exchange language, the Resource Description Framework (RDF), is also gaining popularity. The strengths of RDF are a well-defined process for merging

several data sources and the ability to represent arbitrary graphs. Dealing with RDF directly requires quite a technical mind set, e. g. thinking about the distinction between literals, blank nodes and URIs. As requirements we get ability to represent typed links between entities (↦ Req. 16).

Most aspects of (X)HTML have already been described in the general section on documents and hypertext.

Creativity Tools Besides standard office document format like text document, spreadsheet, presentation slides, and diagrams, mind maps are becoming a popular format, too. Mind maps were invented by Buzan (1991) to help people learning new material faster and better. As a computer application, mind maps are mostly used to (re-)structure items (↦ Req. 17) or help in capturing ideas in discussions.

Conceptually, a mind map is a tree of nodes (Req. 8), centered around a central root node. In many popular tools such as *Mind Manager* and *FreeMind* only the nodes can carry labels, the arcs cannot.

In *CMap Tools* the user can label nodes and links. There is not a single central node. The data model of *CMap Tools* is very similar to the SWCM model described in Sec. 2.3, but *CMap Tools* lack semantics and queries.

More Advanced PKM Tools Existing prototypes for PKM tools such as MITs *Haystack* (Adar, Karger & Stein 1999, Quan et al. 2003) allow the user to manage a large set of information items in a homogenous way. A central idea in Haystack is the notion of a collection, which may contain items of different kinds. CDS should allow to group items in arbitrary collections. (Req. 13)

All items are rendered in a similar fashion. CDS tools should be able to render all kinds of items. (↦ Req. 18)

Web Sites For Collaborative Information Organisation Many web sites allow users to "tag" their content. *Delicious*[4] assigns single-term keywords to bookmarks, *flickr*[5] labels digital images. Both systems allow users to browse the implicitly created sets of items which share a common tag. Many other popular web 2.0 site for collaborative information organisation also offer tagging as a lightweight means to structure information. Some systems even allow to add structure to the tags themselves (e. g. Soboleo[6] and Bibsonomy[7]).

The data model consists of items which can have zero, one or more tags. CDS should allow to tag items. (↦ Req. 19) Similar to Bibsonomy, users should be able to create structure between tags. (Req. 10)

Knowledge Representation Languages Existing knowledge representation languages are very general (RDF, RDFS, Topic Maps, OWL, Conceptual Graphs) and feature few semantic relations suited to directly model and structure personal knowledge.

Existing vocabularies and ontologies (SKOS[8], IBIS Kunz & Rittel (1970)) are too domain-specific (c. f. Req. 1) to model arbitrary personal knowledge.

RDFS does not distinguish between class and instance. Every resource can be used as a type of another resource. OWL is stricter, and mandates a clear separation of modelling layers. CDS should be able to assign types to items (Req. 15). CDS should allow to do meta-modelling, i. e. assign types to types ((↦ Req. 20)).

Both RDFS and OWL have inheritance hierarchies of classes and properties. Both have the notion od domains and ranges for properties; they are used in a reasoner to

[4] http://del.icio.us
[5] flickr.com
[6] soboleo.com
[7] bibsonomy.org
[8] http://www.w3.org/2004/02/skos/

infer types of instances on which theses properties are used. OWL offer a construct to state equality between concepts, RDFS lacks such a construct.

2.3 Data Model Layer – Semantic Web Content Model (SWCM)

The *Semantic Web Content Model* (SWCM) is the data model of CDS. Although SWCM has been inspired – among other things – from RDF, it follows completely different goals. RDF is a rather technical data format for information integration and data exchange. SWCM is a conceptual data model designed for end-users. As an analogy, compare the technical level of the file system, which consists of nodes, blocks, node tables and file allocation tables. From a users point of view, a file system is a tree containing folders and files. As such, SWCM describes also some user interface aspects, since SWCM can be used without the CDS ontology.

2.3.1 SWCM in a Nutshell

Semantic Web Content Model (SWCM) consists of five kinds of entities.

Model A knowledge model consists of *Items*.

Item Every entity in SWCM is an item. Each item has a unique URI which makes it globally addressable. Additionally, each item *may* have content attached to it. There is no content not attached to an item. Content may be textual or binary. Binary content is defined as on the web, i. e. having an encoding, mime-type and length in bytes. Textual content has by default UTF-8 encoding. Each *Item*belongs to a model.

NameItem A *NameItem*is a special kind of item, with two restrictions on its content. First, a *NameItem*may have only textual content. Second, this textual content must be *unique* within a knowledge model. This naming concept (inspired from wiki naming) allows to address items via human type-able names. Note that SWCM does not mandate the name-content-pairs that form a typical wiki page.

Relation A *Relation*is a special kind of *NameItem*. Relations can be used to state statements, as explained in the next paragraph. In addition to a *NameItem*, each *Relation*p has a *mandatory inverse* Relation $-p$. This allows to render all statements of the form (s,p,o) additionally as (o,-p,s). This is very handy for user interfaces which allow browsing of items.

Statement A *Statement*is always of the form (*Item*, *Relation*, *Item*). As a statement is itself an *Item*, the user can annotate statements as well – a handy feature e. g. for discussion systems.

2.3.2 Semantics

Similar to the RDFS semantics Hayes (2004), SWCM is used to assert facts about a universe. Instead of repeating the RDFS specification, we highlight the differences.

Items Each item has a URI. Two items with the same URI are the same *Item*.

Statements In SWCM, *Statements* are *Items* themselves. As such, they are addressable. From each *Statement* x we can assert the fact (x.source, x.relation, x.target) in the universe. Note that the identity of the *Statement* does not influence the asserted facts. It is possible that difference statements with the same URI assert

the same facts. Each statement has always exactly one source item, one relation and one target item. Therefore it is not possible that two different statements (differing in source, relation and/or target) have the same URI.

NameItems SWCM has the consistency axiom that no two *NameItems* have different URIs and the same content. Formally, for two *NameItems* a and b the following holds:

$$a.content = b.content \Leftrightarrow a = b \tag{1}$$

There may be normal *Items* (those that are not *NameItems*) having the same content as a *NameItem* or as another *Item*.

Inverse Relations Every *Relation*p has an inverse *Relation*$-p$. The inverse of the inverse of a *Relation*p is p:

$$-(-p) = p \tag{2}$$

For every *Statement*(s, p, o), the inverse *Statement*$(o, -p, s)$ is inferred, where $-p$ is the inverse of p. That is:

$$\forall s, p, o : (s, p, o) \Mapsto (o, -p, s) \tag{3}$$

Comparing SWCM and RDF Although, technically, SWCM is *implemented* using RDF, from a users point of view this does not matter. Many semantic web applications hide the complexity of RDF completely – for a good reason. Most users do not want to deal with URIs, blank nodes, literals, data-types for literals, language tags for literals and other RDF subtleties such as the `rdf:List` construct. Some early semantic web applications *do* show URIs to the end user – such user interfaces are now considered immature and prototypical. The core idea of SWCM is to expose the same *expressivity* to the user that RDF has, basically a graph with typed links.

The expressivity of SWCM is comparable to RDF. Again, SWCM is not meant to replace RDF. Instead, it is a data model for the end user, which is implemented in RDF. It could also be implemented using other encodings besides that. Yet, SWCM has a similar expressivity as RDF, therefore we now compare SWCM with RDF.

SWCM is an extended subset of RDF. The following entities of RDF have been removed:

Blank nodes: All entities in SWCM have a URI. This URI is never shown to the user.

Data-typed Literals: All literals in SWCM are plain literals. SWCM is meant to record personal thoughts, not to store technical data.

Language-tagged literals: SWCM is meant to be used by one person. There are no multiple values in different languages.

Literals : SWCM has no "plain literals". Instead, each item (identified by a URI) has exactly one literal attached.

SWCM also extends RDF. The following *features* have been added, compared to RDF:

Addressable Literals: The fundamental concept of SWCM is the *Item*. Each item is first addressable via a URI and second it may contain zero or one content. No content can appear outside of *Items*. Each piece of content is thus addressable, which is used e. g. to record creation date and authorship of each *Item*. RDF does not allow to address literals.

Addressable Statements: As *Statements* in SWCM are special *Items*, all *Statements* also have a URI. Hence all SWCM *Statements* are addressable. In RDF, statements are *not* directly addressable and reification and named graphs do not solve this either, out of the box.

Inverse relations: RDF and RDFS do not define inverse relations. OWL does define inverse relations, but they are not mandatory. In SWCM, inverse relations are mandatory.

Name items: SWCM has the notion of *NameItems*, which allow the user to address items via a memorable string. This was inspired by the usage of WikiWords in wikis. *NameItems* are items where the content has naming characteristics. RDF does not have a *naming concept* for humans.

2.4 CDS Ontology Layer

SWCM allows to represent knowledge in various degrees of formality. CDS extends SWCM with an ontology of relations. CDS is represented in the SWCM data model. A typical user is expected to use SWCM together with the CDS ontology.

CDS has the central concept of a relation hierarchy (the CDS ontology), which lets the user orientate even in large knowledge models.

We found that there is a relatively small set of types of structural relations between information items, which occur very often in all of the tools analysed in Sec. 2.2. We grouped these relation types according to different dimensions that they describe. Finally the more specific ones of these relations were subsumed under the more general ones depending on how generic they are. This hierarchy of relations is the basis of the CDS ontology.

2.4.1 A Subsumption Hierarchy Of Common Relations

Based on the analysis of data models in Sec. 2.2.3 we distilled a small set of relations sufficing to represent them. We arranged these relations in a subsumption hierarchy, from general to specific.

The four core relation types are depicted in Fig. 2. Relations are listed with their name and the name of their inverse relation. Fig. 3 shows the CDS relation hierarchy. The resulting ontology, expressed in RDF/N3 can be found in the appendix. The complete CDS relation ontology is now discussed in detail.

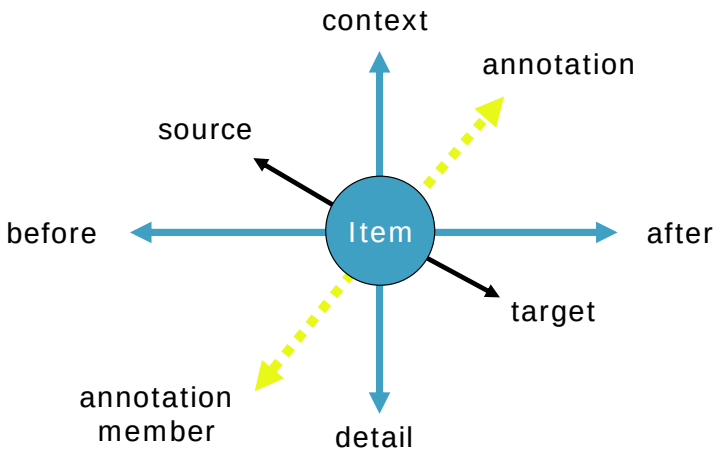

Figure 2: The four core CDS relations

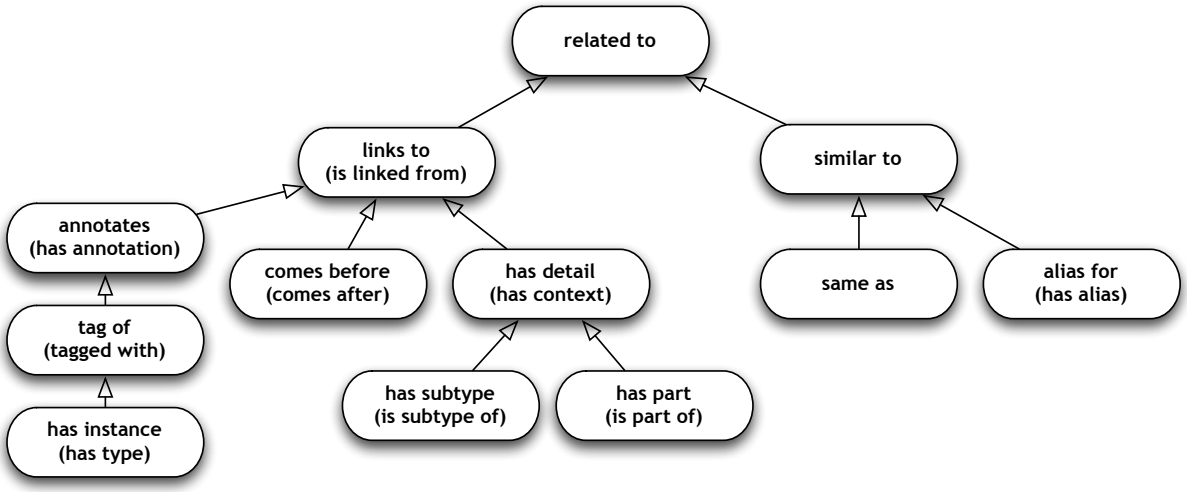

Figure 3: Relation Hierarchy in CDS

Related `related/related` is the most basic relation. Every item is related to another item, if any kind of relation has been stated. All relations are sub-relations of this relation by default.

Similar, Same, Alias and Replacement If one Item A is an alias of another Item B, then all Statements about A are considered to be Statements about Item B. If one Item is a replacement for another one, then an occurrence of A's content is replaced at edit-time with B's content, e. g. when editing text.

Hyperlinks `cds:hasTarget` and its inverse `cds:hasSource` model generic, directed linking. This can be found in WWW hyperlinks, references in documents, or links in the file system. The semantics of a link are pretty generic: A link refers to a target Item.

Order `cds:hasAfter` and its inverse `cds:hasBefore` model any kind of ordering relation. It might be order in space, time or by other means, e.g. priority or rank. Sequences such as arrays and lists are used in virtually any information system.

`has before/has after` is a sub-relation of `has target/has source`. Different from rdf:Lists, this allows to represent partial order or even cyclic order. Such freedom is important to let a user experiment with different orders, e. g. for his tasks.

Hierarchy `cds:hasDetail` and its inverse `cds:hasContext` represent any kind of hierarchy and nesting. Hierarchies are very common information structures present in documents, organisational charts, file systems, and user interfaces. This relation models hierarchies in a generic way. Part-whole relations or type hierarchies are considered special cases of this relation. `cds:hasDetail` is a sub-relation of `cds:hasTarget` from which we can follow (due to inverse relation semantics) that `cds:hasContext` is a sub-relation of `cds:hasSource`.

Annotation, Tagging and Typing `cds:hasAnnotation` and its inverse `cds:hasAnnotationMember` models annotations of Items. An annotation is typically a statement about an item-taking a meta perspective. This relation indicates a difference in the modelling layer. Annotating items covers everything from virtual

sticky notes up to tagging and formal typing. `cds:hasAnnotation` is a sub-relation of `cds:hasTarget`. `cds:hasAnnotationMember` is a sub-relation of `cds:hasSource`.

Tagging Together with thy hype around "Web 2.0", tagging became popular for assigning easy-to-type keywords on items. CDS considers tagging as a special form of annotation. Hence `cds:hasTag` is a sub-relation of `cds:hasAnnotation`. `cds:hasTagMember`is the inverse of `cds:hasTag`. `cds:hasTagMember` is a sub-relation of `cds:hasAnnotationMember`.

As tags inherently have the characteristics of being a unique name, users in CDS *should* tag with *NameItems*. As there are some approaches tagging e. g. images with images [9], CDS does not *restrict* tagging to *NameItems*.

Typing The next formalisation step is possible by assigning types to *Items*by referring to any *NameItem*with the built-in relation `cds:hasType`.

Classifying an item with `cds:hasType` is modelled as a special case of tagging. Tagging has no formal semantics, but types are inherited via the `cds:hasSubType`-Relation-which is in turn a special form of `cds:hasDetail`. `cds:hasType`is a sub-relation of `cds:hasTag`. `cds:hasType`has the inverse `cds:hasInstance`. `cds:hasInstance`is a sub-relation of `cds:hasTagMember`. This allows all types to be used as tags as well and allows applications using CDS but not understanding formal types to give the user still some benefit of the formal data.

2.4.2 Semantics

In this section we explain briefly the semantics of CDS. Besides the semantics listed in this section, the other CDS ontology *Relations*have no formal semantics besides them being *Relations*in the sense of the SWCM model.

Subsumption (`cds:hasSubRelation`, `cds:hasSubType`) Types form a subsumption hierarchy, exactly like in RDFS. A cycle in the subsumption graph denotes that all relations equal. Type subsumption in CDS is indicated by `cds:hasSubType` and its inverse `cds:hasSuperType`.

Relations form a subsumption hierarchy, exactly like in RDFS. A cycle in the relation subsumption graph denotes that all relations equal.

Together with inverse relations we get:

$$\begin{aligned} & (s,p,o),(p,hasSubRelation,q) \\ (3) \Rightarrow \quad & (o,-p,s) \\ \Rightarrow \quad & (-p,hasSubRelation,-q) \end{aligned}$$

In CDS, relation subsumption is bound to the built-in relation `hasSubRelation` with its inverse relation `hasSuperRelation`.

Aliases (`cds:hasAlias`) The CDS *Relation* `cds:hasAlias` links a *NameItem* to another *Item*, acting as a shorthand for navigation and data entry. Given an alias a pointing to an item i (a, hasAlias, i), a *Statement* about a implies the same statement about i. Formally,

$$\begin{aligned} (a,hasAlias,i)(a,p,o) &\Rightarrow (i,p,o) \\ (a,hasAlias,i)(s,a,o) &\Rightarrow (s,i,o) \\ (a,hasAlias,i)(s,p,a) &\Rightarrow (s,p,i) \end{aligned} \tag{4}$$

[9] See the ImageNotion project `http://www.imagenotion.com/`

Comparing CDS to RDF Schema and OWL CDS is intended to be used by end-users, not for data exchange between machines. But as RDFS has been a strong inspiration for the semantics of CDS, we present a brief comparison.

Class and Instance: CDS does not clearly distinguish between the two. Rather, every item can be used as a type for another item. This is the same in RDFS, but different from OWL.

Domain and Ranges: CDs has no notion of domains and ranges. Every relation can be used with every kind of item.

Class hierarchies: CDS has a *hasSubType* relation, similar to RDFS' `subClassOf`.

Property hierarchies: CDS has a *hasSubRelation* relation, similar to RDFS' `subPropertyOf`.

Same As: Different from RDFS, but more similar to OWL, CDS has several ways to state similarity (*has similar*) or equality (*same as*).

2.5 Using CDS

Whereas the SWCM allows modelling content snippets (*Items*) and arbitrary relations between them, the CDS ontology defines a simple schema language to classify, relate and describe relations. It is designed to allow for soft migration from unstructured to structured knowledge. The user is free to create any *Relation* or *Item* types he needs. CDS only demands from the user to classify all *Relations* according to the CDS ontology. This is not really a constraint, however, since unspecified *Relations* can safely be made a sub-*Relation* of the top-level *Relation* `cds:related`. The CDS ontology is a taxonomy of *Relations*, each lower-level *Relation* implies the higher-level *Relations*, just like in RDF Schema (RDFS). Every *Item* that has any kind of *Relation* to any other node is at least `cds:related`.

The usual way to use CDS is: A user creates a number of text items, comparable to brainstorming with sticky notes on a white-board. Then she groups these snippets and connects them with arrows. Later she specifies these relationships by labelling the arrows. After a while, she might see that some items share common characteristics and assigns them to one or more types such as "Person", "Idea" or "Todo". These types can be exploited for search, e. g. "give me all Persons in Karlsruhe". Arcs are classified in a similar fashion and can be typed with Relations such as "knows" or "part of".

2.5.1 Evaluation wrt. Requirements

In this section, we give list how the requirements gathered in Sec. 2.2 are met. First we look at requirements solved by the design of SWCM:

- Req. 1: SWCM is not limited to any domain. Users can create any number and kind of *Items*and relation types as they need.
- Req. 4: Every item in SWCM has a URI, as such it is globally reference-able.
- Req. 3: SWCM items can also contain binary content with a given mime-type. This enables them to contain images as well.
- Req. 10 and Req. 16: *Statements*in SWCM represent formal statements between items.
- Req. 14: *NameItems*allow users to address items via human-type-able names.

- Req. 5: Using formal SWCM *Statements*, *Items*to describe the metadata of an *Item*can be represented.
- Req. 20: As each *Statement*and each *Relation*are *Items*themselves, the user has full meta-modelling abilities in SWCM.
- Req. 18: In fact, neither SWCM or CDS can really ensure this requirement. It is up to the user to use renderable content for the non-*NameItems*. *NameItems*and *Relations*can always be rendered as their content is restricted to a plain string.
- Req. 13: Grouping of items can simply be achieved by linking several items e. g. a, b, c to the same target t via any kind of relation e. g. p: (a,p,t),(b,p,t), and (c,p,t). Then the query (*, p, t) returns the group of items.
- Req. 7: Step-wise formalisation is addressed by SWCM insofar, as a user can explicate her knowledge as the content of items in unstructured form, or structured via wiki-syntax (see 6). The migration from unstructured text to wiki syntax is smooth. Knowledge can also be represented as formal statements. Sec. 6 describes also how such statements can be derived from wiki syntax to smooth the transition.

Other requirements are addressed by features of the CDS ontology:

- Req. 11: A simple, directed hyperlink in CDS is modelled as `cds:hasTarget`/ `cds:hasSource`. Any kind of resource can be linked with any other resource (e. g. ideas, persons, files, issues, tags, types, ...). The direction and type of the link can be specified.
- Req. 6: Order in CDS is represented as `cds:hasAfter`and `cds:hasBefore`.
- Req. 8: All kinds of hierarchies, including type-hierarchies are modelled in CDS as `cds:hasType`(inverse: `cds:hasInstance`) or sub-relations of these relations. This allows generic tools to browse all kinds of hierarchies.
- Req. 15: In CDS, each item can be used as anothers items type via `cds:hasType`. This allows to add each item a type and is required to enable full meta-modelling, i. e. assign types to types (Req. 20).
- Req. 19: Is realised in CDS via `cds:hasTag`.
- Req. 7: A central idea of the CDS ontology layer is way how relations are placed into the relation hierarchy: The most general relations (e. g. `cds:related`and `cds:hasTarget`) are at the top, more specific ones are further down. We expect users to refine existing statements as they see fit with more specific relation types.

These requirements can only be met by tools based on CDS:

- Req. 17: Refactoring of content is probably a key strength of mind maps and a key problem of wikis. Good CDS-based tools will allow for easy refactoring of knowledge. The current HKW prototype is still limited in this respect, in the future we plan to add drag & drop editing.

Some requirements are currently not met at all:

- Req. 9: Support for argumentation can easily be added by any end-user simply by creating the appropriate relation types.
- Req. 12: Transclusion has to be enabled in the wiki syntax. This is planned for the future.

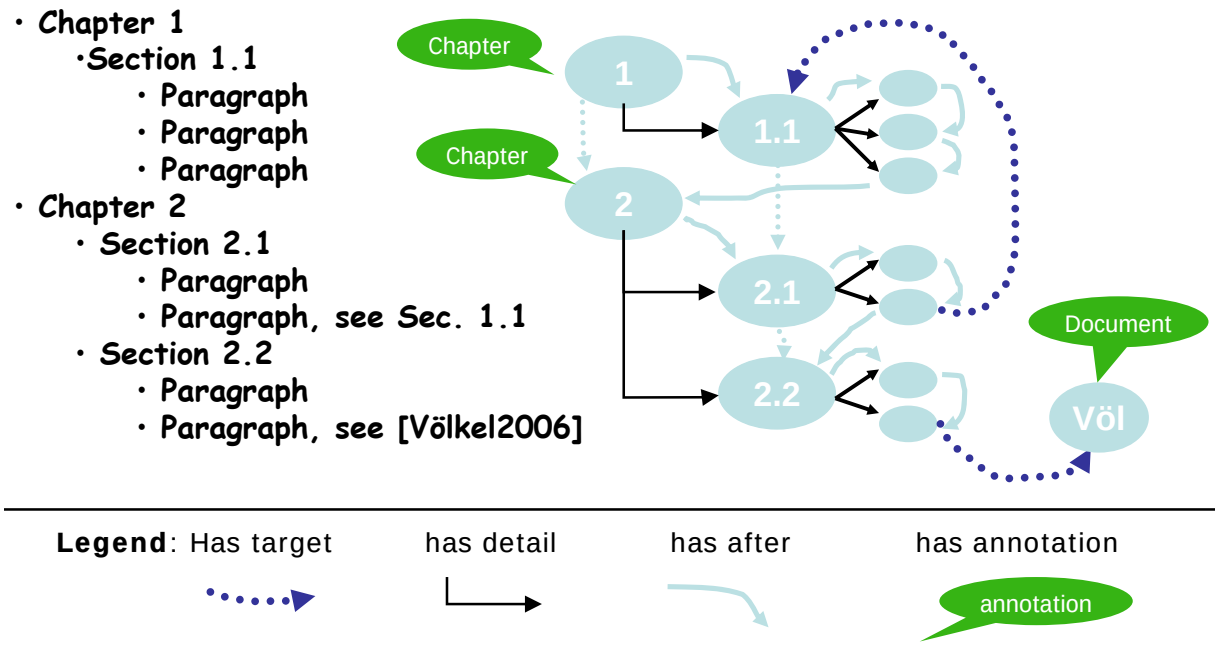

Figure 4: A document represented in CDS

- Req. 2: Whether CDS-based tools can be used successfully by a user over many years cannot yet be said.

CDS offers by its design three parallel ways to work with personal knowledge:

1. content of *Items*, e. g. simple keyword search for item retrieval, using structural and formal knowledge only to improve ranking,
2. Relation structure for retrieval by associative browsing as well as for composing documents from existing items, and
3. semantics of *Items*and *Relations*for reasoning.

Representing Documents in CDS As an example, consider Fig. 4 which shows a mapping of document structures to a CDS knowledge model.

To represent a document in CDS, we can use one knowledge item to represent the root of the document and store the title as the content of it.

Document metadata is modelled as additional items linked to the document root, similar to the way how RDF is used.

Linear navigation is modelled via an arc of type *hasNext* between the items holding the document parts.

Following the approach of Groza, Handschuh, Möller & Decker (2007), we model a document structurally as a tree consisting of root, sections nested into each other, paragraph and sentence. Sections can also contains figures and tables, which are not further modularised. The relation *hasPart* is used to model the different kinds of containment. To distinguish the different types of structural unit we use a relation *hasType* and nodes representing types such as *section*, *paragraph*, etc.

Hennum (2006) describes ways to encode argumentative structures in RDF. One basic observation is that modelling a document as a strict tree, i. e. as in XML, doesn't allow to model overlapping regions. It does not discuss the discourse structures themselves in much detail. This gap is filled by Groza, Handschuh, Möller & Decker (2007) which describes a small yet expressive ontology for argumentative structures, which is based on Rhetorical Structure Theory Taboada & Mann (2006). Groza, Handschuh, Möller & Decker (2007) models argumentation at the sentence

Tour Code			DP9LAX01AB	
Valid			01.05. - 30.09.04	
Class/Extension			Economic	Extended
Adult	P R	Single Room	35,450	2,510
		Double Room	32,500	1,430
		Extra Bed	30,550	720
Child	I C E	Occupation	25,800	1,430
		No occupation	23,850	720
		Extra Bed	22,900	360

Figure 5: A Complex Table

Tour Code	Tour Code	Tour Code	DP9LAX01AB	DP9LAX01AB
Valid	Valid	Valid	01.05 - 30.09.04	01.05 - 30.09.04
Class/Ext.	Class/Ext.	Class/Ext.	Economic	Extended
Adult	PRICE	Single Room	35,450	2,510
Adult	PRICE	Double Room	32,500	1,430
Adult	PRICE	Extra Bed	30,550	720
Child	PRICE	Occupation	25,800	1,430
Child	PRICE	No occupation	23,850	720
Child	PRICE	Extra Bed	22,900	360

Figure 6: A Complex, Normalised Table

level. In CDS argumentation can be modelled by relating content items via their logical structure and at the same time encode the argumentative structure.

Finally, formal statements can be represented as CDS statements.

Representing Tables in CDS An important data structure for information organisation is a table or grid. At a first glance, a table seems to be a simple structure, but their encoded semantics are often more complex. Hurst (2000) distinguishes access and data cells. Pivk, Cimiano & Sure (2005) analyzes how concepts with certain attributes belonging to certain classes are encoded in tables.

A simple table consist possibly only of a headline (access cell) and a number of cells below (data cells). The headline describe e. g. a common type for the instances below. Figure 5 shows a complex table example, taken from (Chen, Tsai & Tsai 2000). It shows a number of prices for different configurations of room bookings. Figure 6, taken from Pivk et al. (2005) shows the same table in a normalised form. Spanning cells have been split into single cells. Now it is easier to see that e. g. "35,450" in fact refers to a value with "Tour Code=DP9LAX01AB", "Valid=01.05-30.09.04", "Class/Ext.=Economic", "Adult" and "Single Room".

In CDS, one can represent this with a plain item (a) and its content "35,450". This item has a special kind of context, e. g. "has tour code" pointing to a *NameItem*with the content "DP9LAX01AB", assuming tour codes are unique names within the model. If not, a plain item can be used instead. In a similar way, a has a relation "has class" to "Economic". Furthermore, a can be tagged with "Adult" and typed with "PRICE". Besides concepts, tables also form a visual grid, implying a kind of order on cells. Cells are arranged in columns and rows. CDS can also represent the visual properties of a table. A table can be seen as a context for a number of columns and a number of rows. Between each column and each row we can model order using "has before" and "has after". Inside each column, we find a number of cells, linked via "has context" to its column and via "has before" and "has after" to its neighbours.

Taking the example from Fig. 5, we would have an empty *Itemt* typed with "'Table".

The table item has two items linked via "has detail", namely an item c typed with "columns" and an item r typed with "rows". Item c has two children, c_1 and c_2. We record furthermore a statement (c_1,"has after",c_2') to model the order of the columns. Column c_1 has 5 details: "Tour Code", "Valid", "Class/Extension", c_{1a} and c_{1b}. Among each consecutive cell we record again the order via "has after". In the end, each cell can be reached on multiple paths from the root item t.

Gradual Formalisation In current applications for recording and restructuring knowledge, people *do* use a lot of structure to add semantics to their content. E. g. they use different colors in a document to indicate its editing status. Or the use different fonts to differentiate source code samples from descriptive test. Or they use bold and italic font shapes to indicate emphasis or proper names. In mind maps, they use little icons to differentiate e. g. ideas from tasks. Such semantics can be communicated out-of-band to a human, but not yet to a computer.

It is the aim of CDS to let the user profit from formal annotations more easily. CDS makes mapping of user-semantics to formal semantics easier by allowing the user to start with less expressive, vague semantics.

There are several way for step-wise formalisation possible in CDS:

- The simplest thing a user can do is to create a plain item. This is equivalent to take a piece of paper and write the date on it.
- Next the user can write plain text in this item. Or set the content to contain an image. In any case, some piece of content is now addressable in the model.
- The user can turn the item into a *NameItem*. The system has to check if another *NameItem*with the same name (content) exists already, and if yes, the user has to decide what should happen: Delete the new item? Rename it? Merge the two? As we see, even small formalisation steps can come at some cost.
- A user can link any two existing items via one of the built-in relations. The simples possible link is a simple, undirected *related* link between two items. It is analogous to drawing a line between to pieces of paper. E. g. sometimes it is easy to say that two items are related but it is hard to say how.
- The user can *refine* any existing relation. For *related* relations, the user might as a next step choose a directed hyperlink (hasTarget) or express a kind of similarity (similar to).
- The user can also create new relations, as they are needed. Browsing a knowledge model is easier when the most suitable parent relation is chosen from the existing relations (c. f. Fig. 3).
- The user can structure the content of an item using wiki syntax, e g. format text in sections, lists and tables.
- The user can add formal statements to the wiki content (c. f. Sec. 6.3).

Queries in CDS Queries in CDS are basically SPARQL (Prud'Hommeaux & Seaborne 2007) SELECT queries with only one projected variable. This allows them be used in set-like operations like intersection, union and set difference. CDS queries also need a kind of full-text query. Luckily, a component developed by DFKI and Aduna, the *LuceneSAIL*[10] can execute combined semantic queries, i. e. queries involving SPARQL and full-text parts.

The formal query language is based on atomar triple *patterns* $p \in P$ composed of items i, relations r and the wildcard $*$. Note that all items and relations in CDS

[10] `http://dev.nepomuk.semanticdesktop.org/wiki/LuceneSail`

are identified with a URI. Formally,

$$P = \{(i, r, *), (i, *, i), (i, r, *)\}, i \in I, r \in R$$

where the first parameter of the pattern denotes the source item of a statement, the second the relation of a statement and the third parameter the target item of the relation.

A query $q \in Q$ is then either an atomar pattern, a negated query ($\neg$), or the intersection ($\cap$) or union ($\cup$) of two queries. Formally,

$$Q = \{p, \neg q, q_a \cup q_b, q_a \cap q_b\}; p \in P; q_a, q_b \in Q.$$

As an example, *all friends of Dirk living in Karlsruhe that do not work at SAP*, would be represented as the query

$$\begin{aligned} q_1 &= q_2 \cap q_3 \\ q_2 &= q_4 \cap q_5 \\ q_3 &= \neg p_1 \\ q_4 &= p_2 \\ q_5 &= p_3 \\ p_1 &= (*, worksAt, SAP) \\ p_2 &= (Dirk, knows, *) \\ p_3 &= (*, livesIn, Karlsruhe) \end{aligned}$$

or in one query

$$q = ((Dirk, knows, *) \cap (*, livesIn, Karlsruhe)) \cap (\neg(*, worksAt, SAP)).$$

This query language (CDS-QL) can be mapped to SPARQL as follows: CDS-QL patterns are mapped to SPARQL patterns by replacing each $*$ with `?var` and each item or relation with its URI put between angle brackets, that is <URI(item)>.

Negation can be represented in SPARQL via OPTIONAL and FILTER. For any pattern $p = (x, y, z)$ where one of the components is a wildcard $*$, the mapping to SPARQL is

```
OPTIONAL { x y z } .
FILTER( !bound(?var) )
```

Intersection of two patterns is expressed as simply separating the two corresponding SPARQL patterns via a single dot (“`.`”).

Union in CDS-QL is mapped to SPARQLs “UNION” keyword.

Assuming the following URI mapping:

$worksAt \mapsto$ `http://example.com/worksAt`
$SAP \mapsto$ `http://example.com/SAP`
$Dirk \mapsto$ `http://example.com/Dirk`
$knows \mapsto$ `http://example.com/knows`
$livesIn \mapsto$ `http://example.com/livesIn`
$Karlsruhe \mapsto$ `http://example.com/Karlsruhe`

the above example results in the SPARQL patterns:

$p_1 \mapsto$ `?var, <http://example.com/worksAt>, <http://example.com/SAP>`
$p_2 \mapsto$ `<http://example.com/Dirk>,`

```
     <http://example.com/knows>,
     ?var
p3 ↦ ?var,
     <http://example.com/livesIn>,
     <http://example.com/Karlsruhe>
```

The full CDS-QL example query as a SPARQL query is

```
SELECT ?var WHERE {
  <http://example.com/Dirk>
    <http://example.com/knows>
      ?var .
  ?var
    <http://example.com/livesIn>
      <http://example.com/Karlsruhe> .
  OPTIONAL {
    ?var
      <http://example.com/worksAt>
        <http://example.com/SAP>
  } .
  FILTER( !bound(?var) )
}
```

We extend SPARQL SELECT queries with two important concepts for CDS users:

Transitivity The user needs the option to treat each *Relation*as a transitive relation. This feature is not possible in standard SPARQL.

Equality Sometimes it is desired to merge the semantic links of two (ro more) items linked via *sameAs*. The CDS sameAs is not the same as the owl:sameAs, as the cds:sameAs is not always in effect. This allows e. g. to browse two different items linked via sameAs as two different items. A necessary feature to be able to separate them again.

2.6 Realisation

As the CDS ontology is represented in an SWCM model, the CDS API is realised as an extension (technically a decorator) of the SWCM API. As such, a user can use the CDS API to perform arbitrary SWCM operations plus convenience support for the specific CDS ontology relations.

The SWCM API uses a layer called *swecr.core* as its persistence layer. An overview of different CDS tools accessing the CDS API can be found in Fig. 7. Note how the CDS tools operator on the conceptual CDS model, which is implemented by reusing some parts of RDF. The complete CDS API description can be found online at `http://semweb4j.org/site/cds.api/apidocs/`.

swecr.core.api [11] This is a really simple API layer, offering access to other persistence parts:

`getModelSet()` returns an RDF2Go ModelSet, which is a set of named RDF graphs.

`getBinStore()` returns an *IBinStore*[12]. This is a simple component allowing to store and retrieve binary content addressed by URIs.

`getTextIndex()` returns a thin wrapper interface around a full text index, typically *Lucene*.

There exist currently two implementations: *swecr.core.simple*[13] which di-

[11] `http://semweb4j.org/site/swecr.core.api`
[12] `http://semweb4j.org/site/binstore`
[13] `http://semweb4j.org/site/swecr.core.simple`

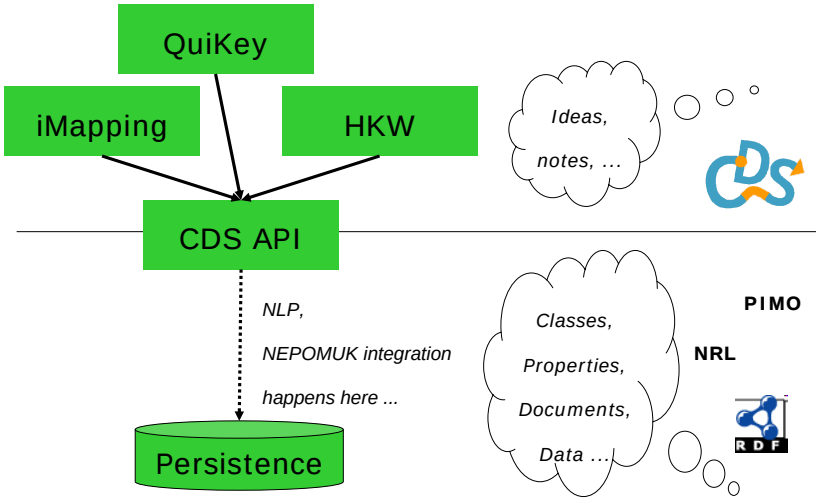

Figure 7: Overview of CDS API usage

rectly stores data in *Sesame*, *Lucene*, and a simple binary store. This implementations offers no semantic queries (using RDF and full text combined) yet. The second implementation, *swecr.core.nepomuk*[14] stores content via the *RDF2GoRepository*, tunnelled over HTTP, in NEPOMUKs RDFRepository. This implementation cannot store binaries yet.

swecr.model [15] This is the data model part of CDS. It implements the model described in Sec. 2.3.

cds [16] The `ICdsModel` extends the SWCM `IModel` interface. The main differences between a plain SWCM model and a CDS model are: Support for inferencing and a built-in hierarchy of relations.

Mapping to RDF We need to map only the data model part (SWCM, c. f. Sec. 2.3). The ontology layer of CDS is represented via the data model. In this paragraph we briefly explain how diverse SWCM constructs are currently mapped to RDF. We use the following namespace bindings (Turtle syntax):

```
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#> .
@prefix swcm: <http://purl.org/net/swcm#> .
```

Note: The implementation currently uses random URIs, encoding the current system time and a random number, prefixed by `urn:rnd:`, resulting in URIs such as `urn:rnd:252f391c:1167c5d5744:-7fd7`.

Item Each item with URI i is represented as i `rdf:type swcm:Item`. If the item has content attached to it, it is represented as i `swcm:hasContent` *content*. Additionally, each item has a change date (`swcm:hasChangeDate`) and an author (`swcm:hasAuthor`).

An example of an item with content:

```
<urn:rnd:252f391c:1167c5d5744:-7f44> rdf:type swcm:Item
; swcm:hasContent "Lorem ipsum dolor sit amet,
consectetuer adipiscing elit. Proin id enim a
velit cursus tempor. Aenean non erat. Mauris
imperdiet, sem in iaculis interdum, velit libero
```

[14] http://semweb4j.org/site/swecr.core.nepomuk
[15] http://semweb4j.org/site/swecr.model
[16] http://semweb4j.org/site/cds

```
aliquam nisi, scelerisque tincidunt leo dui eget pede."
; swcm:hasChangeDate "2007-11-26T14:48:12Z"^^xsd:dateTime
; swcm:hasAuthor swcm:author-unknown
.
```

NameItem *NameItems* are represented almost exactly like items. In addition, they have the type `swcm:NameItem` assigned. An example:

```
<urn:rnd:252f391c:1167c5d5744:-7fd7> rdf:type swcm:Item
; rdf:type swcm:NameItem
; swcm:hasContent "Dirk Hageman"
; swcm:hasChangeDate "2007-11-26T14:37:09Z"^^xsd:dateTime
; swcm:hasAuthor swcm:author-unknown
.
```

Relation A *Relation* is a special kind of *NameItem*. It has an additional inverse relation specified via `swcm:hasInverse`. And it has the type `swcm:Relation` assigned. An example:

```
<urn:rnd:252f391c:1167c5d5744:-7fd0> rdf:type swcm:Item
; rdf:type swcm:NameItem
; rdf:type swcm:Relation
; swcm:hasContent "writes PhD at"
; swcm:hasChangeDate "2007-11-26T14:38:09Z"^^xsd:dateTime
; swcm:hasAuthor swcm:author-unknown
; swcm:hasInverse <urn:rnd:252f391c:1167c5d5744:-7fcf> .
.
```

Statement A *Statement* is also an item, but has the additional type `swcm:Statement` assigned. A *Statement* may not be a *NameItem* or *Relation* at the same time. A *Statement*, just like a normal *Item* may have content. Below is an example of a statement without content attached. Note that the statement is in RDF terms "reified", to allow addressing it. The plain, un-reified statement is also recorded in the RDF to simplify answering of SPARQL queries:

```
<urn:rnd:252f391c:1167c5d5744:-7fcd> rdf:type swcm:Item
; rdf:type swcm:Statement
; swcm:hasChangeDate "2007-11-28T15:18:19Z"^^xsd:dateTime
; swcm:hasAuthor swcm:author-unknown
; swcm:stmtSource   <urn:rnd:252f391c:1167c5d5744:-7fd7>
; swcm:stmtRelation <urn:rnd:252f391c:1167c5d5744:-7fd0>
; swcm:stmtTarget   <urn:rnd:252f391c:1167c5d5744:-7fd1>
.

<urn:rnd:252f391c:1167c5d5744:-7fd7>
  <urn:rnd:252f391c:1167c5d5744:-7fd0>
    <urn:rnd:252f391c:1167c5d5744:-7fd1> .
```

Current Level of Integration in NEPOMUK The CDS API has an implementation which depends on a `swecr.core` API. Both levels are NEPOMUK-independent. The `swecr.core` API requires three services: An RDF triple store, a full-text search engine and a binary store.

Two implementations of the `swecr.core` API exist: First, a stand alone implementation, based on Sesame, Lucene and BinStore.

A second implementation is based on NEPOMUKs RDF2GoRepository which connects via HTTP to a running NEPOMUK server. The NEPOMUK repository contains already a combination of RDF triple store and full-text search engine. A setup using this implementation is depicted in Fig. 8.

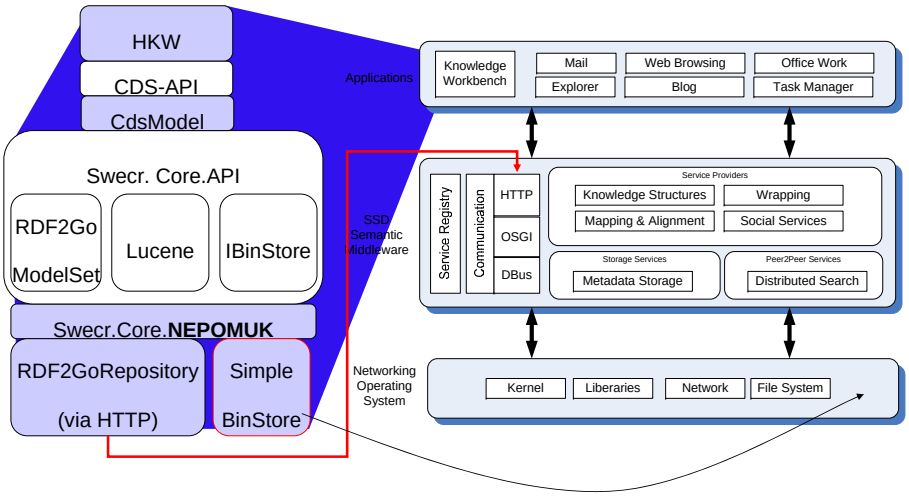

Figure 8: Current Level of Integration

3 Hypertext-based Knowledge Workbench (HKW)

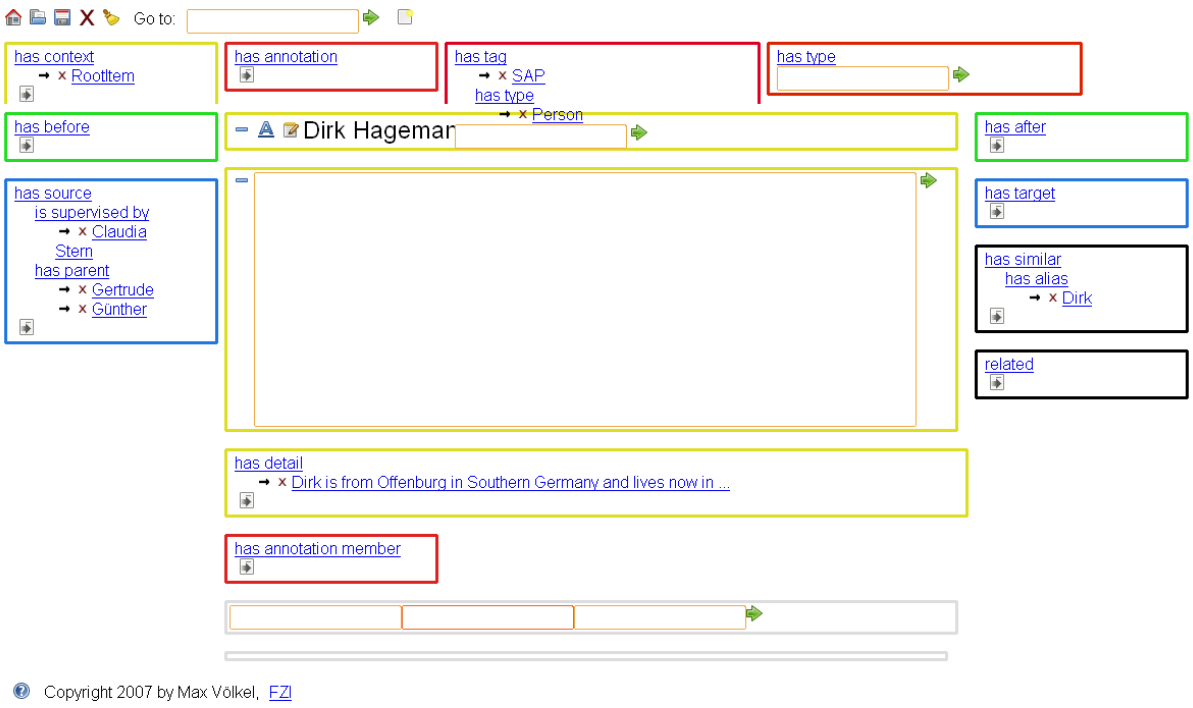

Figure 9: HKW prototype screen shot, focusing on *Dirk Hageman*

This section describes the current prototype of the Hypertext-based Knowledge Workbench (HKW). HKW allows to create, browse and author CDS models.

HKW is the first of two prototypes currently under development in NEPOMUK, completely based on CDS ideas. HKW exposes all CDS ideas to the user, iMapping (the other prototype) adds the notion of space to CDS.

Although HKW has among other things, been influenced by the idea of semantic wikis, HKW is not like an ordinary semantic wiki, as it has no notion of "document". Instead, it puts the focus more on micro-content and its relations. As a direct result, searches and navigation do not bring up long wiki documents, but short fragments of text with its relations to other parts.

3.1 Design

The HKW prototype has been designed based on the feedback collected on the early NEPOMUK semantic wiki prototypes, described in Kotelnikov, Polonsky, Kiesel, Völkel, Haller, Sogrin, Lannerö & Davis (2006).

Semantic Pad has the ability to create customised views that include business logic. The review feedback evaluated the programming as too hard for a casual end user. Therefore we did not include this functionality into the HKW prototype. There were no complaints on the wiki syntax used by Semantic Pad. As the syntax is parsed via an re-usable, powerful component (WikiModel, see Sec. 6.1), we plan to integrate it into HKW.

Kaukolu got positive evaluations for the auto-completion mechanism. Hence HKW uses auto-completion almost everywhere. Kaukolu got negative feedback for unclear instant reward of adding semantic statements on wiki page. HKW makes creation of statements more explicit, by adding items into different boxes statements are created. Additionally, the new items are shown,

where they have been created, offering instantly better ways for navigation and orientation.

Semantic MediaWiki (SMW) got the best feedback, mostly because of its maturity. The strongest point is that SMW can be used just like a normal wiki, completely ignoring the semantic features. As the current HKW cannot parse wiki syntax yet, this is not yet true for HKW. But it is out goal to achieve the same simplicity of use ad SMW (allowing the user to ignore semantic features). SMW is simpler, which comes at a cost of expressivity, compared to Kaukolu. HKW will additionally allow to state statements via wiki syntax, which is described in Section 6.3. Auto-completion will include links to desktop resources.

Requirements stated in Kotelnikov et al. (2006):

- Composition of documents using fragments of different documents: Create a new document based on fragments of different documents of previous work. – needs multi-level transclusion
- Ontology refactoring: The ontology properties and types should be refactorable. – Done.
- Template system: The system should let users define advanced rules defining the way resources should be displayed, using templates. – HKW has chosen a path requiring to dedicated templates. The view is configured by classifying relations.
- Visual refactoring: Users with appropriate rights should be able to refactor the contents visually, changing their names, their status, their meta-data. – Done for contents.

Compared to the semantic wiki prototypes, the CDS-based tools aim for

- A tighter integration with desktop data
- Intuitive, efficient graphical user interface (iMapping)
- A switch at the user interface level from RDF to CDS, to reduce cognitive load for knowledge articulation. RDF will be use behind the scenes for implementation purposes only.
- Overcome segmentation of knowledge into pages – instead in CDS knowledge is represented as a highly networked set of small items.

3.2 User Guide

HKW is an editor and browser for CDS models. The GUI runs in a web browser (currently only the open-source browser *Mozilla Firefox* is supported). The core concept of the GUI is the notion of focus on a given element. Fig. 9 shows a screenshot of the HKW GUI focusing on the *NameItem* "Dirk Hageman". Below this selected item we have a large area for quick text input, related to "Dirk Hageman". The different axes of CDS are arranged around the centered concept. Each axis has a different color. The most important axis, the hierarchical `cds:hasContext` / `cds:hasDetail` is rendered in yellow. Context is at the top, the details are below. Similar, the axis for order, `cds:hasBefore` and`cds:hasAfter` are arranged left and right of the center, rendered in green. Annotations, tags and types are rendered in red. Their inverses are subsumed under "has annotation member" at the bottom. Arbitrary, directed links are rendered in blue, incoming links are left in "has source", outgoing links are right under the headline "has target". The black boxes are relations of the type "related" and "similar" which are both undirected relations. The GUI shows relations always in their most specific box. Items are only rendered in different boxes at the same time if the user assigned multiple super-relations to a relation.

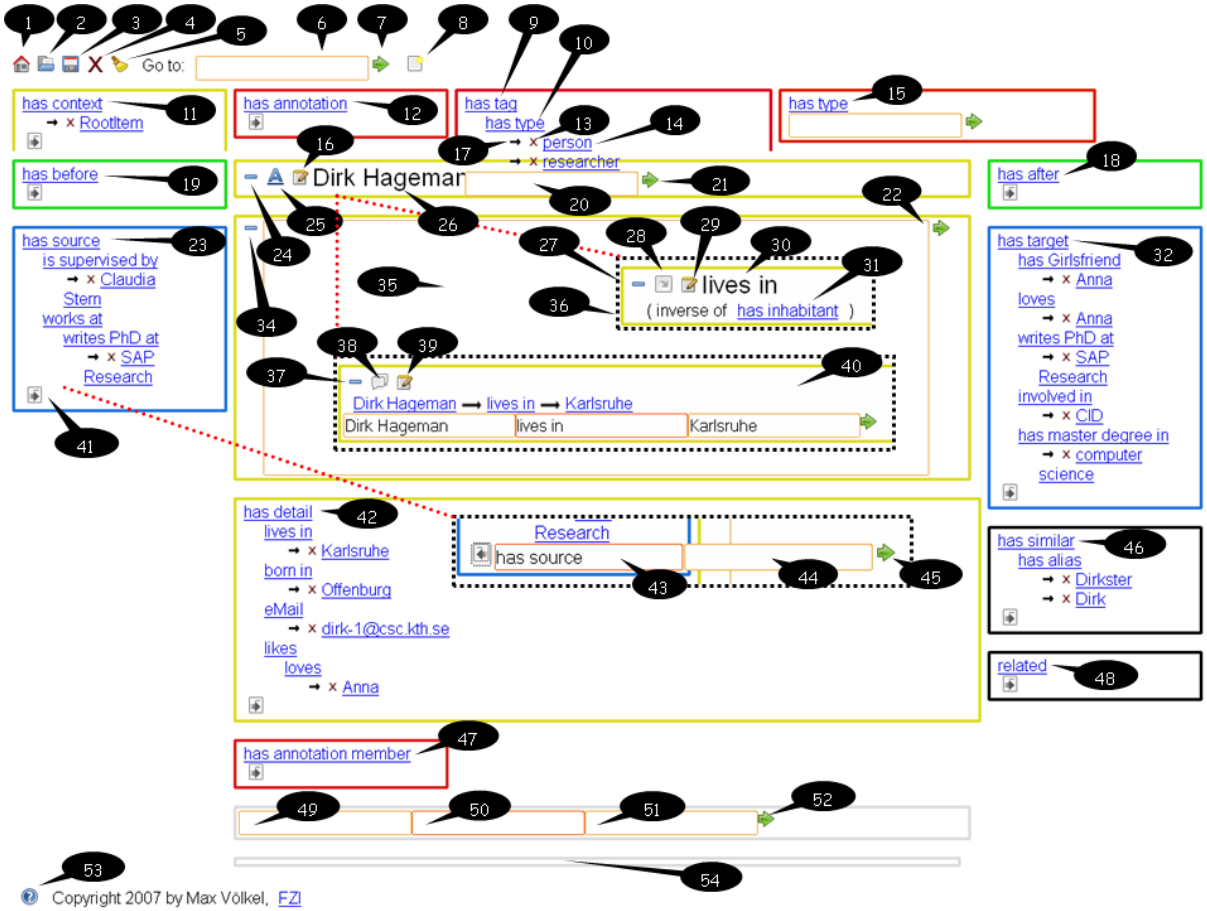

Figure 10: HKW GUI reference

Figure 10 shows a screen shot of the current HKW prototype, with all GUI parts numbered. In the remainder of this section we describe the functionality of each part.

1 Home – focus on "RootItem"

2 Load – there is one slot to store models. This button replaces the current state with the one stored on disk.

3 Save – store the current state on disk. Overwrites an existing stored state.

4 Clear – deletes the current state, adds the built-in relations and focuses on RootItem.

5 Clear – this button deletes the current state and adds a built-in dummy model for testing purposes. This button will be removed in the future.

6 Address bar – this bar supports auto-completion. The user can enter a known name item and press (7) to navigate to it. Or she can enter a new name for a name item. Pressing (8) creates a new name item with the given name and navigates to it. It is not possible to navigate to non-name items via this address bar.

7 Navigate to name item – see (6)

8 Create name item – see (6)

9 has tag – this panel allows the user to use tags on the focused item (26). It shows that "has type" (10) is a sub-relation of "has tag" and that there are two items connected via the relation "has type" to the current focus item ("Dirk Hageman"). In fact, this is a bug - the relation "has type" should be shown in box (15). Other boxes show sub-relations in a similar way.

10 "has type" – a sub-relation of "has tag".

11 "has context" – showing all items which act as a context (opposite of detail) to the focused item. "Dirk Hageman" is only in the context "RootItem".

12 "has annotations" – Shows all annotations of the current focus item. The little arrow (described in 41) allows to add new annotations.

13 delete statement – this red X allows to delete the statement ("Dirk Hageman" "has type" "person"). None of the three items themselves are deleted, only the statement.

14 Navigate to "person". All items in the view, including the relations ("has tag", "has type") can be clicked on and will then be the focus item.

15 "has type" – shows all types of the focus item. New types can be added via the box and the green arrow.

16 "edit item" – allow to switch the focused item to edit mode. in edit mode, the user can change the string associated with the item. No links ever break when doing this.

17 focus statement – allows to navigate to the statement ("Dirk Hageman" "has type" "person"). (40) shows a focused statement.

18 "has after" – showing all items that are "after" the focus item.

19 "has before" – shows all items "before" the focus item.

20 Create or add new item – allows to create a new *Item* (or re-use an exiting *NameItem*) which will be added as a *tag* of the current focus item. In other words, a statement (current item, has tag, new *Item*) will be added. This entry box does not allow to create new relations. A similar entry field, allowing to create new relations, can be found in 43. The field 20 supports auto-completion.

21 This button (green arrow) adds the item that has been entered in 20. Alternatively, the user can type Ctrl-Return for the same effect.

22 This green arrow allows to add the text entered into the main box (35) as a new detail. In the future, the text in this box will *additionally* be interpreted as wiki syntax, possible creating further *Items* and *Statements*.

23 The box "has source" shows all *Items* x that are connected via (x, has target, current focus item) or the inverse (current focus item, has source, x). More figuratively, it shows all items that have a hyper-link pointing *towards* the current focus item. The opposite directions is shown in 32. This box also shows all sub-relations of "has source", in the screen-shot those are *is supervised by* and *works at*. The latter has a further sub-relation, *writes PhD at*. The statements expressed by the box are thus (Dirk Hageman, is supervised by, Claudia Stern) and (Dirk Hageman, writes PhD at, SAP). 41 allows to add ne members to this box.

24 This button deletes the current focus item. Recursively all statements *about* this item are also deleted. Deleting a relation (by first focusing on it, then pressing this button) deletes all statements using this relation.

25 This symbol (an underlined A) shows that the current focus item is a *NameItem*. Pressing the button turns the *NameItem* into a normal *Item*. A normal item is symbolised with an icon showing a piece of paper with three lines on it (not shown in the screen shot).

26 Focus item – here a representation of the current focus item is shown.

27 Same as 24.

28 This icon indicates that the item is *read-only*. Therefore the user is not allowed to change the type. Relations are always *NameItems*.

29 Same as 16.

30 Same as 26.

31 Showing the inverse of the relation. The user can navigate to it.

32 The "has target" box. This is the counterpart to 23. In this box (32) the user sees all outgoing links, that point away from the current focus item. 23 and 32 show the statements as the user entered them, although each statement has a corresponding inverse statement that could be rendered instead. The relation *writes PhD at* has been considered by the user to *both* point toward the focus item, hence it is shown in 23, and pointing away from the focus item, hence it is also shown in 32. Most relations have only one direction.

33 Does not exist.

34 This icon collapses the box 35. It turns into a "plus"-sign which expands the box again.

35 Main entry panel to quickly add new details of the current focus item.

36 This view shows (26) when the user has navigated (focused on) a relation.

37 Same as 24.

38 Icon showing that the user has navigated to a statement. Statements are always normal *Items*.

39 Same as 16. Allows to edit a textual content of the statement.

40 View when the user navigated to a statement. Inside, the statement the user focused on is shown twice. In the blue line, the user can see the current statement and navigate to each item that constitutes it. Below, the same statement is shown as auto-completion text boxes. Here the user can *change* the current statement, e. g. set a new source, target or relation. Creation of new items for source or target is also possible.

41 Pressing this expand-button shows the entry-panel with 43, 44, and 45.

42 This box shows all details of the current focus item, similar to 23 and 32. The opposite direction of "has detail" is "has context", which is shown in 11.

43 Allows to create new relations, but only together with adding items to the current box. The relation will automatically be a sub-relation of the relations main box. E. g. when the user creates a relation in 23, the new relations is automatically a sub-relation of "has source". The system will automatically create an inverse relation, named *name of relation*`-inverse`. 43 supports auto-completion on existing relations.

44 Allows to select existing items (via auto-completion) or create new items. Together with 43, the user enters a statement of the form (focus item, relation entered via 43, item entered via 44).

45 Pressing this button (or hitting Ctrl-Return) commits the new statement (43, 44) to the system.

46 Showing "similar" items. This is another box like 23.

47 Shows the "annotation members". Those are the opposite of 9, 12 and 15. In other words, here the instances of types, the items tagged with a tag, or the items annotated with an annotation are shown.

48 Showing related items. This is a box similar to 23, but slightly different. The relation "related" has no direction. Hence the statements (x, related, y) and (y, related, x) are equal.

49 Together with 50 and 51, this allows the user to state queries to the system or enter free-form statements. If all three boxes are filled (all boxes support auto-completion) and the user presses the green arrow (52), then a new statement is added to the system. If on of the boxes is left blank, it is interpreted as a wild-card and a query is performed.

50 See 49.

51 See 49.

52 See 49.

53 Help icon. Navigates to the *NameItem* "Help".

54 Is the box showing the results of queries posed in 49.

3.3 Realisation

The HKW is based on the CDS API. As such, it has the same integration with NEPOMUK as the CDS API.

The HKW prototype has been realised with the *Google Web Toolkit* (GWT), an open-source AJAX-enabled web user interface toolkit by Google Inc. AJAX web GUIs run a certain part of the program code in the end-users browser. Traditional web applications ran all code on the server side, resulting in a flickering user experience. Each time the user clicked, she had to wait for a complete page reload from the server. Network latency thus really added up as user wait times. With AJAX, some code is running as JavaScript in the browser, often not requiring a page reload as all for navigation operations. If a page reload is required, JavaScript requests only the needed data from the server and changes the loaded pages in the browser – without a reload. The result is a much more fluid user experience. The GWT toolkit runs only stable in Firefox and Internet Explorer. The same is true for many web applications provided by Google Inc. As Firefox is a popular, stable browser available for free on all platforms (Windows, Linux, Mac) this was not considered a draw-back.

Programming JavaScript, a dynamically typed script language, is not easy. For dynamically typed languages, there are no IDEs with e. g. refactoring support. The different APIs of different browsers make the problem even harder. The unique approach of GWT is to write the code in Java, a statically typed programming language, for which many mature IDEs exist. GWT includes a compiler, which translates the Java code into JavaScript code. Additionally, GWT includes a widget library, similar to desktop GUI frameworks such as Swing or SWT (from the Eclipse project). Programming with GWT thus relieves the user from dealing with browser differences and JavaScript.

The HKW prototype uses GWT widgets and some custom-build widgets. Styling in GWT is done via *Cascading Stylesheets* (CSS). The server side sits on top of a CDS API, which is described in Sec. 2.6.

3.4 Evaluation

3.4.1 Expert Evaluation – Method

Our usability experts from KTH evaluated both the Hypertext-based (Sec. 3.4.2) and a prototype of the visual (Sec. 4.1.5) Knowledge Workbench following an integrated method for evaluating interfaces developed by McQuaid & Bishop (2001) with a few changes. The method consists of five steps:

1. Gathering domain knowledge.
2. Conducting the heuristic evaluation.
3. Categorizing the issues.
4. Prioritizing the issues.
5. Writing the report, including recommendations for solving the problems.

The first step, gathering domain knowledge, was considered already done since we, through previous activities, have good knowledge of what NEPOMUK and WP1 wants to achieve. Though, before we performed the evaluation of each prototype, we defined the different prototypes' function and aim and also decided which persona would be most relevant for each prototype. We then performed the evaluation as a group, using a projector showing the prototype. This is different from McQuaid's method where different people perform the evaluation individually. Instead, we went through, group wise, the different steps of performing tasks that

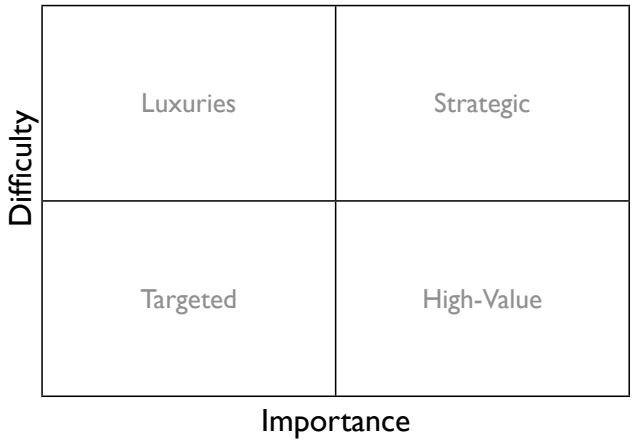

Figure 11: Prioritizing chart

the prototypes help you with. Related to heuristics and design guidelines (Nielsen 2005) as well as our own experience as usability professionals we wrote down our judgments and suggestions on post-it notes.

We then sorted the post-it notes in low-level and high-level problems and concerns. Low-level problems are on the screen-level and refer to the usability of buttons, dialog boxes, and other elements that appear on a single screen. High-level problems refer to the usability of the overall interaction, including problems with navigation and task sequence. We also noted other concerns like valuable features that already work. When the post-its were sorted we prioritized the problems according to how important it is to fix them, from the users' perspective, and how difficult it is to fix them. The latter is from the developers perspective and since we are not developers it was rather difficult to make a correct judgment. Our technical partners should reconsider this prioritisation.

The prioritisation was made on the chart in Figure 11. The y-axis represents the difficulty of making the improvement and the x-axis is how important the improvement is for the users. The different fields can be defined in the following way:

- High-value: very important issues that require less effort to fix
- Strategic: very important issues that require more effort to fix
- Targeted: less important issues that require less effort to fix
- Luxuries: less important issues that require more effort to fix

This report is the result of the fifth and final step in the evaluation process; writing the report, including recommendations for solving the problems. The report describes the results of the evaluation of each of the prototypes, the identified problems as well as recommendations for solutions to the problems.

3.4.2 Expert Evaluation – Results

The purpose of the HKW prototype that we tested is to show information and relations between resources as well as editing the relations between them. All the NEPOMUK personas are relevant for the prototype, but we decided to focus on Dirk and Claudia during the evaluation.

The HKW prototype that was evaluated had no visible indication of version[17].

[17] The evaluated version can be downloaded from `http://semweb4j.org/repo/de/xam/cds.gwt/0.0.3/` and will run permanently at `http://octopus13.fzi.de:8888/cds.gwt-3/`

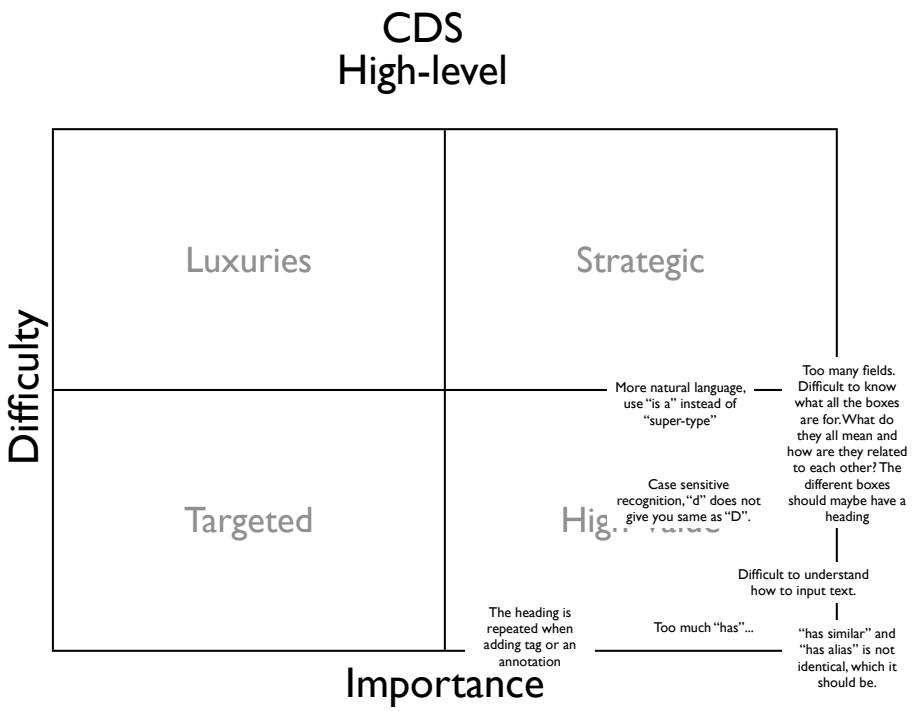

Figure 12: Problem prioritisation - High-level

High-Level Results Users in the different case studies are asking for a better way to browse and search for information, as well as to see the connections between different resources, in their knowledge base and this is what the CDS prototype is aiming to provide.

Powerful semantic modelling tools (e. g. *Protégé* and *TopBraid Composer*) are loaded with features and very flexible. For a demanding modelling professional, they have much to offer. But they are not easy to use. HKW could make semantic modelling easier by its approach to emphasize some of the most important semantic dimensions, giving them dedicated positions on the screen. Relevance at the cost of flexibility.

The auto complete is very helpful for the uses when searching and adding information.

The prototype it very thorough, which is both positive and negative. The negative aspect is that it is more difficult to evaluate. But it is positive that the prototype can do so much and gives such a comprehensive picture of the different resources.

More natural language, use "is a" instead of "super-type" To exemplify: the relation "super-type" may not be a natural concept for describing. For instance, in everyday language people would probably say that Dirk is a person, instead of Dirk has super-type person. Using natural language the annotation tools would probably be used to a greater extend when people can understand the relations.

Too many fields Many people claim that Google is used worldwide just because it is the interaction with it is so simple. There is no question what to do when the search engine is rendered in the web browser. With the HKW prototype it is difficult to know where to start and what all the boxes (c. f. Fig. 13) are for. What do they all mean and how are they related to each other? The different boxes should have a heading that indicates what the box is used for and be grouped according to functionality. Also, the HKW could have two different modes, simple and advanced mode. We recommend one field for just the most basic relations and another for a power user who wants to

Figure 13: Example showing all the fields

manipulate the data in a more sophisticated manner.

Case sensitive recognition, "d" does not give you same as "D". Dirk and dirk should be the same item. If they are different, or the name for two different items is the same (e. g. Paris (the city) or Paris (the celebrity)) what separates the items should be explicitly explained.

Difficult to understand how to make text input The text in a text field should disappear when the field is activated. It makes the user aware of that the field is meant for her.

The heading is repeated when adding tag or an annotation Separate the heading and the tag or the annotation. Either delete one of them (reducing the cognitive load on the user) or use them to complement the information to exemplify and bring more meaning to the functionality. For example "has target" and "has source" are superfluous for the users to understand that Dirk has a girlfriend name Anna or that Claudia is his boss.

Too much "has"... To the user, the concept "Context" works as well as "has Context". All "has" presented in the prototype are not needed to bring clarity and increase usability. Instead, it has the opposite effect and makes the user uncertain and insecure and wonder about the application, which is not recommended if you want to develop an application with high usability.

Identity "has similar" and "has alias" is not identical, there should be a possibility to add a relation that treats objects as identical. "Dirk Hageman", "Dirk" and "dirk" should all "love Anna" and "like running" because they are the same person. Adding or editing a relation to "one of them" should effect "them all".

Low-Level Results

Layout Overlapping text boxes when adding text, e. g. in "has tag"

Colors Is there color-coding? If so, make that explicit. What do the colors mean?

Layout The "X" is too close to the link, too easy to delete (see Fig. 15). And if you do delete it is difficult to undo.

Layout What is the purpose of the large area in the middle? When you add text there it appears as a detail.

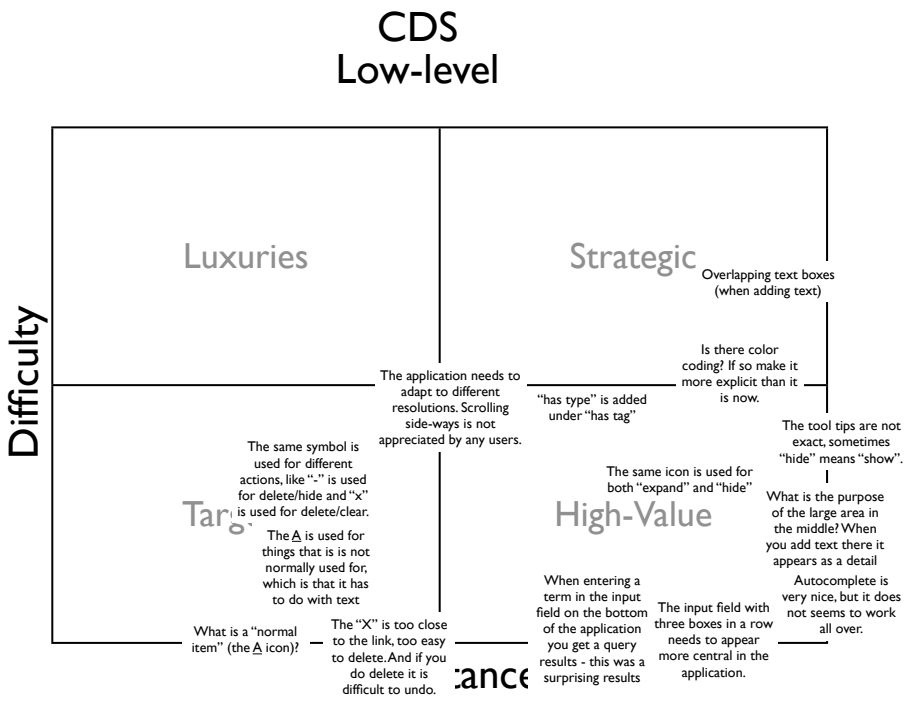

Figure 14: Problem prioritisation - Low-level

Figure 15: An example showing the "X" being too close

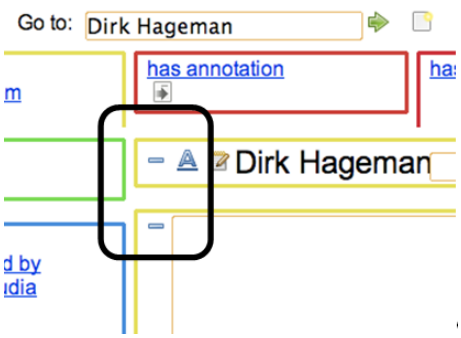

Figure 16: Same icon for "expand" and "hide"

Availability The application needs to be dynamic and adapt to different resolutions. Scrolling side-ways is generally not appreciated by any users that we have observed.

Consistency The same symbol is used for different actions, like "-" is used for delete/hide (see Fig. 16) and "x" is used for delete/clear.

The auto-complete functionality is good, but it does not seem to work globally in the whole prototype.

The tool-tips are not exact, sometimes "hide" means "show" – e. g. the plus for the main entry area.

Icons The A is not used for text, which it normally is, and that is confusing to the user. The same icon is used for both "expand" and "hide"

Language What is a "normal item" (the A icon)?

Surprises When entering a term in the input field on the bottom of the application you get a query result - this was a surprising result. A nice surprise but this could just as well happen if you input the same text in the field at the top and instead of "Go to:" can choose "Query:".

"has type" is added under "has tag".

3.4.3 End-User Feedback

FZI also performed a first internal round of gathering end-user feedback. We demonstrated the prototype (same version as for KTH expert evaluation) and let the user test it. We collected the following points (random order):

- Dates should have a special support. In a personal diary I will use a lot of dates. Ideally, a kind of calendar pop-up would ensure that I can only enter dates for certain relations. . . . maybe entering other data should be possible, too, but with a warning. It's better to guide the user than to force him.
- Relations should have icons instead of weird names. Why not have a relation like *has icon* and the ability to upload image files?
- I would like to have some kind of macros in the wiki syntax, so that I could e. g. type `[htp]` and the system would expand it to `hasType Person`.
- The big box in the middle should allow to enter statements in wiki syntax.
- I need a way to change a *NameItem* into a normal *Item* and vice versa.
- In general, there could be a context-menu with a right-click, for operations on the item. Then one would not have to focus on another item for certain operations.
- The UI should allow to report new bugs and feature requests right from the main screen
- Ability for import of mind-maps. First, each relation is mapped to *has target*, but then I can easily refine it.

We will perform further similar studies with end-users, collect more requirements and prioritise them. These requirements will round up the results of the expert evaluations from KTH to drive further development of the HKW prototype.

4 Visual Knowledge Workbench

Like the Hypertext-based Knowledge Workbench, the Visual Knowledge Workbench has the goal to support knowledge workers to enter and maintain contents in a semantically structured way. They both specialise in letting the user create her own structures in various degrees of formalisation, i. e. supporting the construction of hybrid knowledge models in CDS. These CDS models can encompass parts that are fully formalised semantic models, but at the same time casual text notes, as well as many weakly structured *semi-semantic* elements.

The Visual Knowledge Workbench mainly consists of the iMapping client for visually authoring CDS models. It is build to allow intuitive authoring of personal knowledge bases using mainly graphical mouse interaction. The iMapping client is described below in Sec. 4.1. It will be complemented by the QuiKey tool, a kind of smart semantic command-line that focuses on highest interaction efficiency to browse, query and author CDS Models. QuiKey is described in Sec. 4.2. During the integration efforts scheduled for 2008, some additional components might be created to enrich the Visual Knowledge Workbench.

4.1 iMapping

iMapping is a technique for visually structuring information objects. It supports the full range from informal note taking over semi-structured personal information management to formal knowledge models. With iMaps, users can easily go from overview to fine-grained structures while browsing editing or refining the knowledge base in one comprehensive view.

An iMap is comparable to a large white-board where information items can be positioned like post-its but also nested into each other. Spatial browsing and zooming as well as graphical editing facilities make it easy to structure content in an intuitive way. iMapping builds on a zooming user interface approach to facilitate navigation and to help users maintain an overview in the knowledge space.

The iMapping approach is described along with its motivations and foundations in more detail in Haller (2006) – the accompanying poster is depicted in Fig. 17. In the following, we describe some more technical details that go beyond what is published there.

4.1.1 iMapping Design principles

Criteria for Visual Mapping Environments The core design principles of iMapping has arisen from prior work on visual mapping techniques for the organisation of knowledge (Haller 2003): It deals with the psychological foundations of visual mapping techniques, compares existing approaches and applications and deduces a set of criteria, which – from a cognitive psychological point of view – should be met by cognitively adequate mapping techniques and tools.

This is the list of criteria and how they are met by the iMapping design. Some of them refer more to the mapping technique itself while others are rather to be met by the corresponding software tool. However they are not treated separately here. For a more detailed description of these criteria, see Haller (2003).

Free Placing Any information item can be freely placed anywhere on the canvas or inside any other item.

Free Relations Links between Items can be set in any level of formality: They are not mandatory at all an can be omitted; explicit but unspecified links simply representing a general relationship are possible as well as informally labelled links and specific links carrying formal semantics.

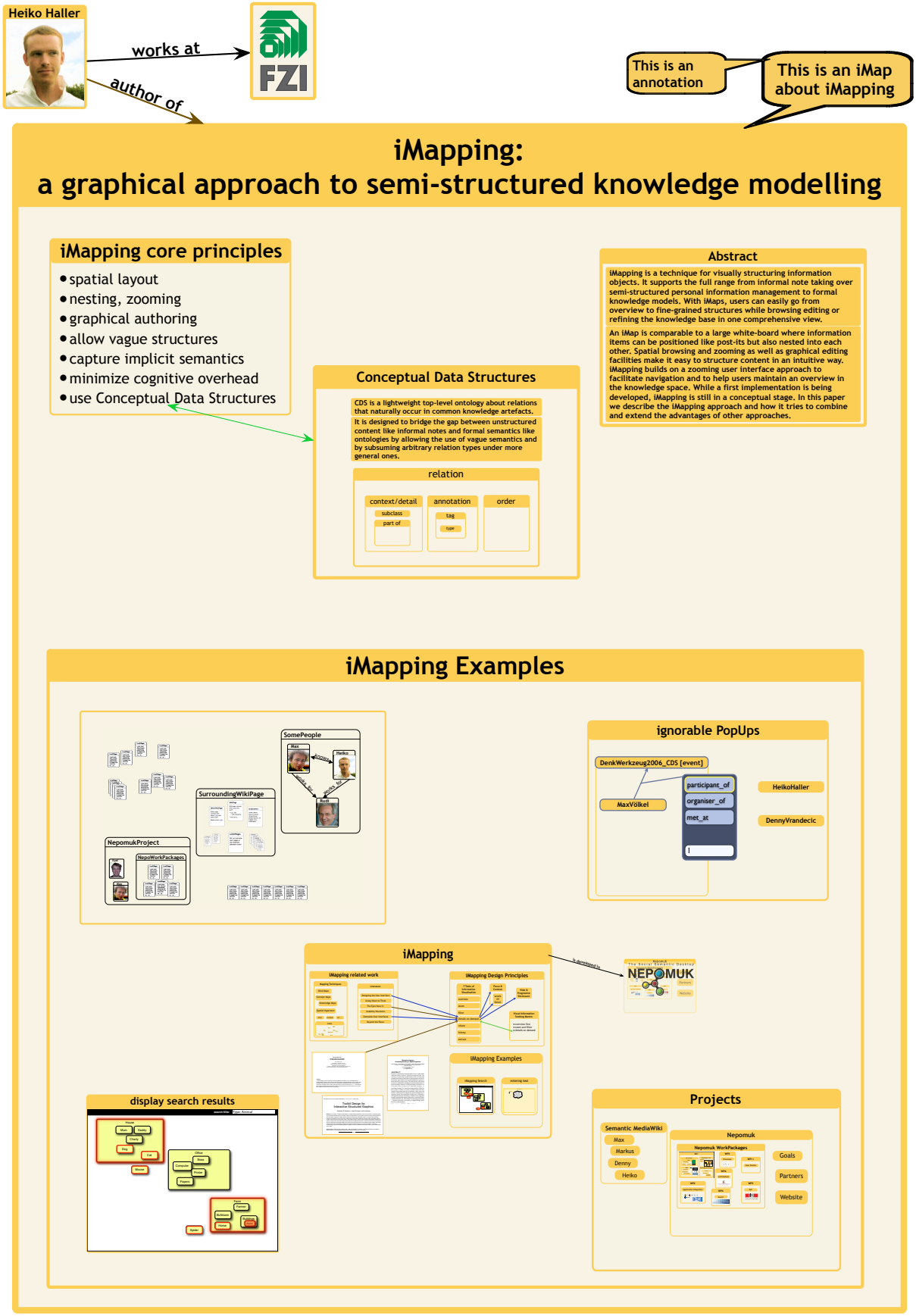

Figure 17: An iMap about iMapping – Poster presented at the Semantic Web User Interaction Workshop (ISWC 2006)

Annotations Annotating Items is again possible on every level of formality: with an informal text note, tagged with another NameItem, but also using any formal relation in the sense of semantic annotation.

Macro Structure / Facility of Inspection Inspecting the overall macro structure of a topic or subtopic in a way that enables the inspection of the topic by grasping an overview before drilling down into the details is facilitated by the iMapping approach in a unique way through its deep-zooming facility.

Hyperlinks To be able to explicitly relate Items that are not visually close together, in-line Hyperlinks can be used as well as generic CDS statements e.g. entered with the QuiKey tool described in Sec. 4.2.

Easy Edit Imposing only the least necessary cognitive overhead onto the user is one of the highest design goals of the iMapping tool. This reflects e.g. in items being able to be created by simply clicking and writing anywhere without needing further specifications of type, meaning or file name. Texts can be edited in WYSIWYG and items can be restructured by drag-and-drop. Whether this is intuitive enough, later user studies will show.

Integration of Detail and Context Navigating complex information environments often leads to a loss of orientation, because when drilling down into detailed views of a matter, context information often gets out of reach. To aid mental integration of details and context, the iMapping tool provides facilities to easily zoom out of and back into a spot, like taking a step back from some detail work to become aware of the surroundings. Additional Levels-Of-Detail approaches like to semantic zooming[18], or Fisheye Views (Furnas 1986) integrate well with the iMapping approach. However it is unclear how much of that we will be able to implement within the nepomuk project.

A number of other visual mapping approaches and tools have been evaluated with these criteria (Haller 2003, Richter, Völkel & Haller 2005) and to the best of our knowledge, there is not a single one that fulfils all of them.

Ben Shneiderman's Seven Tasks of Visual Information Environments, first published by Shneiderman (1996), are well described by Shneiderman & Plaisant (2004). Here is how the iMapping Client supports these tasks.

Overview To gain overview, a user can always just zoom out to see the context of where she has been before. Zooming back in right to where she was before is especially easy because the 'way out' is tracked.

Zoom Zooming is the native browsing method for iMaps.

Filter Several ways of filtering iMapping content to reduce complexity are being discussed. We are currently experimenting with a flash-based prototype.

Details-On-Demand This criterion is met a) by the general zooming approach, b) by the ability to collapse/expand every item on demand and c) by showing Links only on demand to reduce visual clutter, unless they are explicitly made permanent.

Relate To interrelate information items is the core business of semantic knowledge management. This is supported in several ways like dragging items into each other, drawing Links between items or interlinking even remote items with the QuiKey Tool described in Sec. 4.2.

[18] "With a conventional geometric zoom all objects change only their size; with semantic zoom they can additionally change shape, details (not merely size of existing details) or, indeed, their very presence in the display, with objects appearing/disappearing according to the context of the map at hand" (Boulos 2003), for more definitions see *InfoVis Wiki* (n.d.).

History For the iMapping client, we have chosen a transactional design that allows for undo and history functionality. However – since this is not the main focus of this research prototype – it has low implementation priority.

Extract With extracting, Shneiderman means the ability to isolate a subset of items which is to be handled separately or distributed independently. The highly modularised nature of iMaps, allows this conceptionally. In what way the actual sharing will be supported in nepomuk is to be decided.

4.1.2 iMapping GUI

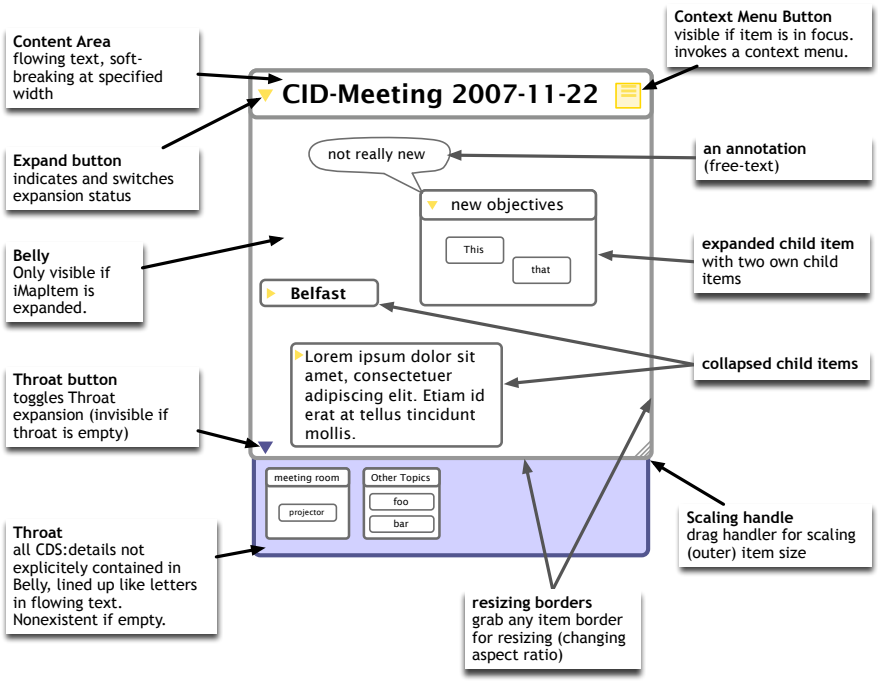

Figure 18: Functional elements on an Item widget in the GUI

A basic text item consists of a content area and a "belly" that can be "collapsed" away. The Belly is the area that holds all the child-items of an item. All such child-items are CDS:details of the item. If there are additional CDS:details that were not explicitly placed in the item's Belly, they appear in the "Throat" (A throat is something where things are before they reach the belly. It's a working label – the name might change in the future.) the throat is normally collapsed, it can be expanded on demand. A current sketch of such a GUI-widget of an iMapping item with some functional elements is shown in Fig. 18.

More details like e.g. on mouse and keyboard interactions can be found in the iMapping development wiki at `http://dev.imapping.info`.

4.1.3 iMapping data Model

All content and the conceptual structure of an iMap is stored in CDS. Every iMap Item represents one CDS:Item whose content it displays and every iMapping Link represents a CDS:Statement (c. f. Figs 20 and 21).

The iMapping data model used in the iMapping back-end however is the result of quite some discussions and refinements, since we aimed at two additional uncom-

mon goals: We wanted to

- store everything, even the visual metadata like item-positions, sizes and other details in an explicit schema in the nepomuk rdf back-end.
- be able to model various kinds of *sameness* between items: a) different iMapItems representing the same CDS:Item, because in different contexts one might want to have completely different-looking representations of one and the same thing. b) *mirrored* iMapItems: Different iMapItems in different contexts that always look the same and 'mirror' each others' changes (see Fig. 19).

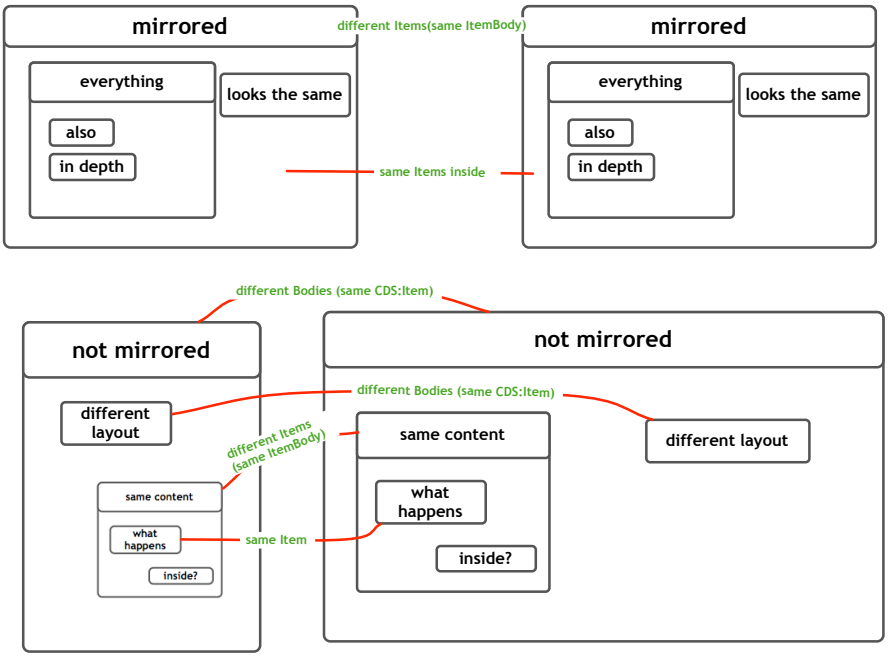

Figure 19: Mirrored Items (top) and not-mirrored items that still representing the the same CDS:Item

For that we had to introduce an intermediate object in the model: the Item-Body. While the *Item* itself carries all the 'outer' properties, which are unique to the item, like its position and (outer) size, the item's *Body* carries the 'inner' properties that are related to the content of the item, like its text or binary content defined by the CDS:Item it represents, but also everything concerning it's 'Belly', like its inner dimensions and references to all its children items. Like that, two 'mirrored' items can share the same content and layout by using the same Body while being displayed in different places since they are still distinct items on their own.

Fig.20 shows examples for these different levels of sameness on all three modelling levels: A and B are different items (different on all three layers) containing a common detail item C. While on CDS-level an item with two different context-items (parent items) poses no further complications, in an iMapping GUI this means that the item is rendered in different locations (inside A and inside B). Because it might be desirable to have C looking different, depending on its context (note its different proportions in the example), the two iMapItems C.1.1 and C.2.1 have different bodies (C.1 and C.2) that both represent the same CDS:Item C.

Item C contains the detail item D, which in our example is to look the same everywhere. So the two Items D.1.1 and D.1.2 are mirrored because they share the same Body D.1.

Note that in figure 20 all objects including the widgets are labelled with IDs, not with the actual content they represent so we can refer to them in this text.

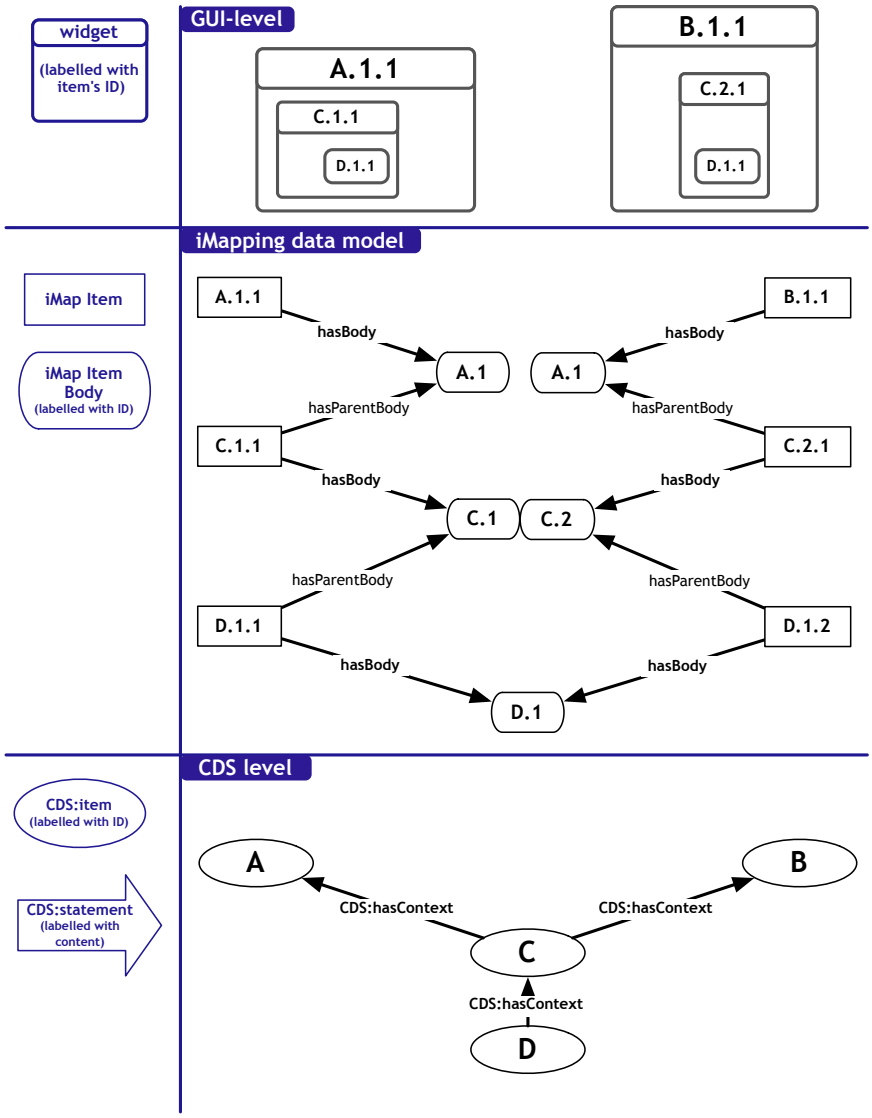

Figure 20: Simple example of two different items containing two mirrored items on all three modelling levels: the CDS model (bottom), the iMapping back-end model (middle) and how it would look in the GUI (top)

Links between iMap Items correspond to CDS:Statements between the respective CDS:Items. However in order to specify where exactly a Link is drawn and whether it should be visible by default or only on demand, iMapping-Links are objects on their own in the iMapping data model. They have specific iMap Items as start and end points.

Like this, a connection between two CDS:Items can be made explicitly visible in one context although it might be hidden because it is irrelevant in another one[19].

iMapping on CDS While this node-and-link-representation would suffice to generically display any CDS:Model as a concept-map-like graph structure, in iMapping some types of Relations are visually represented in more individual ways:

[19] E.g. The statement that Dirk likes Claudia might be relevant for planning the seating arrangements of the SAP Research Christmas celebration, but not wanted to be seen in an organisation chart

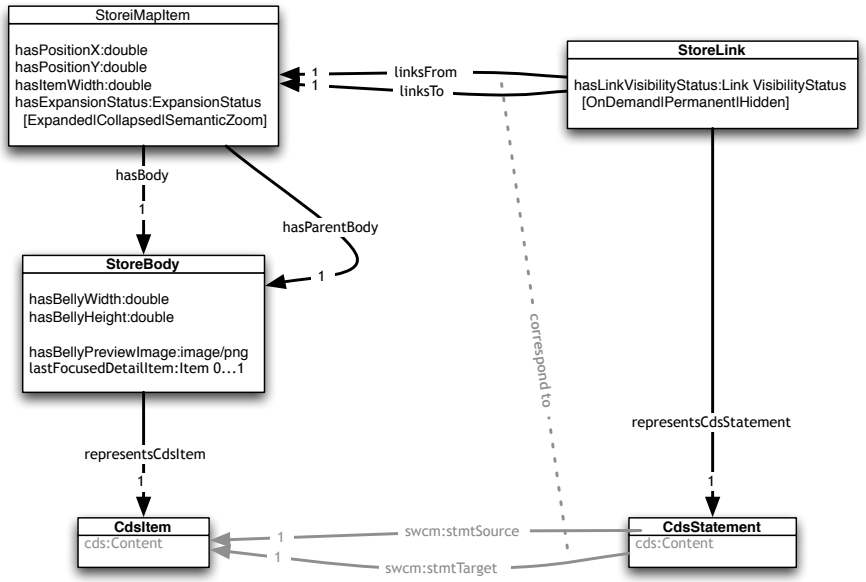

Figure 21: The imapping store data model, representing the mapping from visual, iMapping-specific meta-data to an rdf-graph.

CDS:hasDetail correspond to the nesting structure of iMaps. every child-Item is a CDS:Detail of its parent-Item.

CDS:hasType wherever possible through a suitable plug-in, items of different types will be rendered in a different manner, suitable for the type. So e.g. an Item representing a task could be recognised as such not by having a link pointing at the task-type item but because of its distinct look (c. f. Fig. 22).

CDS:hasAnnotation and CDS:hasTag could also be displayed in distinct ways e.g. as shown on the top of Fig. 17.

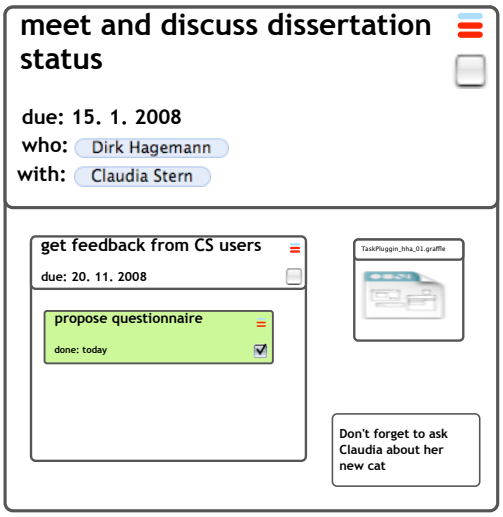

Figure 22: Mock-up sketch of how a task item with sub-tasks could look like.

What could be perceived as an inconsistency between the CDS-way and the iMapping way is, that we want to allow different items to look differently although they represent the same CDS:item. E.g. The Item "Karlsruhe" in the context of a

project should maybe have different child items (maybe "SAP Research", "University of Karlsruhe") than in a cultural context (maybe "ZKM", "Badisches Staatstheater"). While in CDS, all four would just be details of "Karlsruhe" regardless of the wider context, in iMap it is possible to show only some CDS:details as child-items in the item's belly. However to be able to access all other details as well, we added an extra container-area that is expandable on demand and that always carries all 'left over' CDS:detail items that do not have a place in the belly (see Fig. 18). The working title for this area is the "Throat" – In the UI it will later be called something like "additional details".

4.1.4 Implementation Status

There are currently three iMapping implementations:

Flash GUI prototype In order to test different ways of displaying relations between items, we built a GUI prototype in Flash. It has very limited functionality but allows to switch between different modes that are distinguished by background colour (c. f. Fig. 25).

This prototype has been analysed in an expert evaluation described in Sec. 4.1.5.

Java GUI prototype To be able to develop and test the iMapping GUI design without being dependent on the CDS back-end that has only recently reached the required stability, we began implementing a java prototype that omitted all semantic features and just used a classic (hibernate) data base in its back end. Its GUI is already based on the zooming UI framework *piccolo*[20], that is developed by Ben Bederson in Ben Shneiderman's HCI group at the University of Maryland.

Basic Features realised in this prototype include

- creating text items by simply clicking anywhere and starting to type
- moving items around
- nesting items into each other (including automatic adaption of the size of nested items)
- resizing items
- expanding and collapsing items
- zooming / panning freely and deeply
- zooming / panning exactly to any specific item

This prototype has been evaluated in usability tests described in Sec. 4.1.6. It is depicted in Fig. 26

Full version (Java, CDS-based) This is the version that is currently being developed and that is described in the above Sections 4.1.2 and 4.1.3. It is built from scratch, based on the lessons learnt from both designing and evaluating the above mentioned iMapping prototypes. The Full version now actually uses the CDS-API described in Sec. 2.6 to store its content and rdf-reactor[21] for additional visual metadata as described in Sec. 4.1.3. Since most of the work went into the more sophisticated back-end so far, many GUI Features are not implemented yet. However it will provide the following features – additional to the ones listed above for the Java GUI prototype:

[20] http://www.cs.umd.edu/hcil/piccolo/
[21] `http://wiki.ontoworld.org/wiki/RDFReactor`

- Content and structures reflected in CDS
- drawing and typing arbitrary links between items
- scaling items
- additional details area a.k.a. the 'throat'
- rich text editing
- removing / erasing items
- mirrored items
- task-items representing task-objects (c. f. Grebner, Ong, Riss, Brunzel, Bernardi & Roth-Berghofer (2006), Fig. 22)

4.1.5 Expert Evaluation (Flash GUI Prototype)

With the method described in Sec. 3.4.1, we also tested an iMapping prototype realised in Adobe Flash. The purpose of the iMapping prototype that we tested is very limited compared to the vision for the iMapping system under development. It was its purpose to test five different ways to show relations between items. The different versions of this prototype are distinguished with colours: yellow, orange, red, green and blue. We focused the evaluation on these five ways. We also decided that all the NEPOMUK personas would be relevant for the prototype, but during the evaluation we focused on Dirk and Claudia.

The prototype version tested is iMapping Flash Prototype 5 which is available online at `http://proto5.flash.imapping.info`.

High-Level Results Users in the different case studies are asking for a better way to browse and search for information, as well as to see the connections between different resources, in their knowledge base and this is what the iMapping prototype is aiming to provide. It is very good that the iMapping prototype allows for alternative ways to browse the knowledge base. This is exactly what we observed during our field-studies. People behave differently and use their computer in different ways and the software we develop should therefore allow for alternative ways of interaction. It is also positive that it is easy to get an overview and focus on details iMapping with the help of the zooming functionality.

Recommended improvements are:

- **Easy configurability of the way to interact:** It should be easy to configure how you want to interact with the application; e.g. by clicking, mouse-over, etc. The user should also feel secure to make mistakes to enhance the chance of finding an efficient way of interacting.
- **Arbitrary (nested) relations:** The user can put items inside other items in an arbitrary way which is a different solution compared to the highly defined relations shown by lines, movement or highlighting. This could be seen as closeness in different ways, or somehow clustering items. However, it needs to be clearly shown when for example one resource is a part of another resource. One way of showing that could be a drop down list for often used relations within the parent window.
- **Explain and configure the annotation:** Allow the user to rate the annotations. The user should also be able to see who added the annotation and other history of the annotations.
- **Ability to choose the level of the relations shown:** There is a trade-off in showing all relations versus showing none, regarding the information

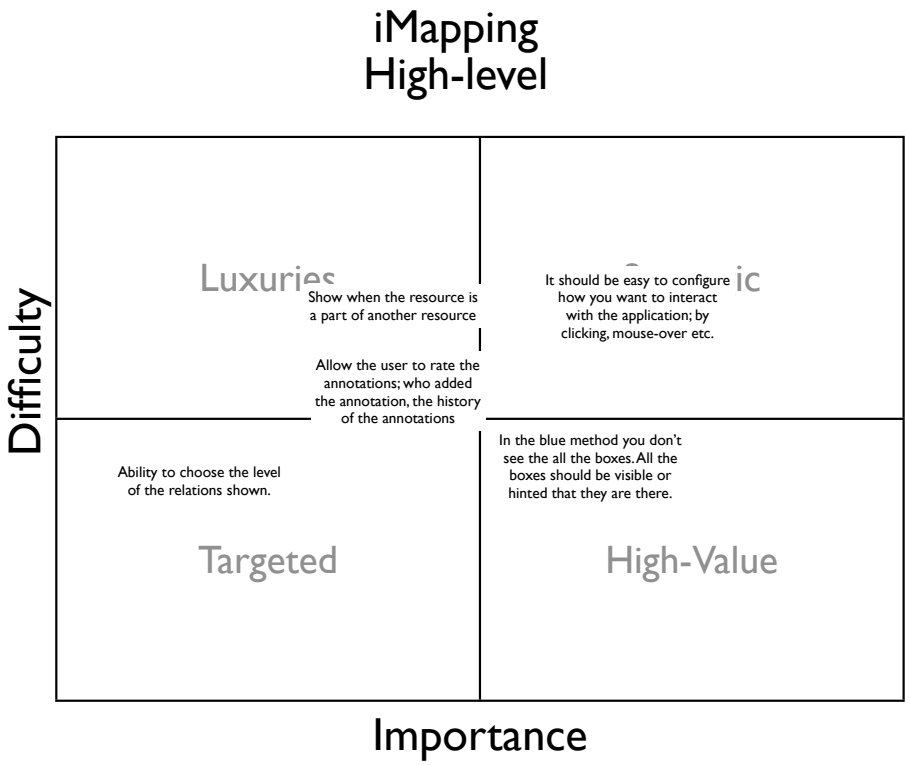

Figure 23: Problem prioritisation - High-level

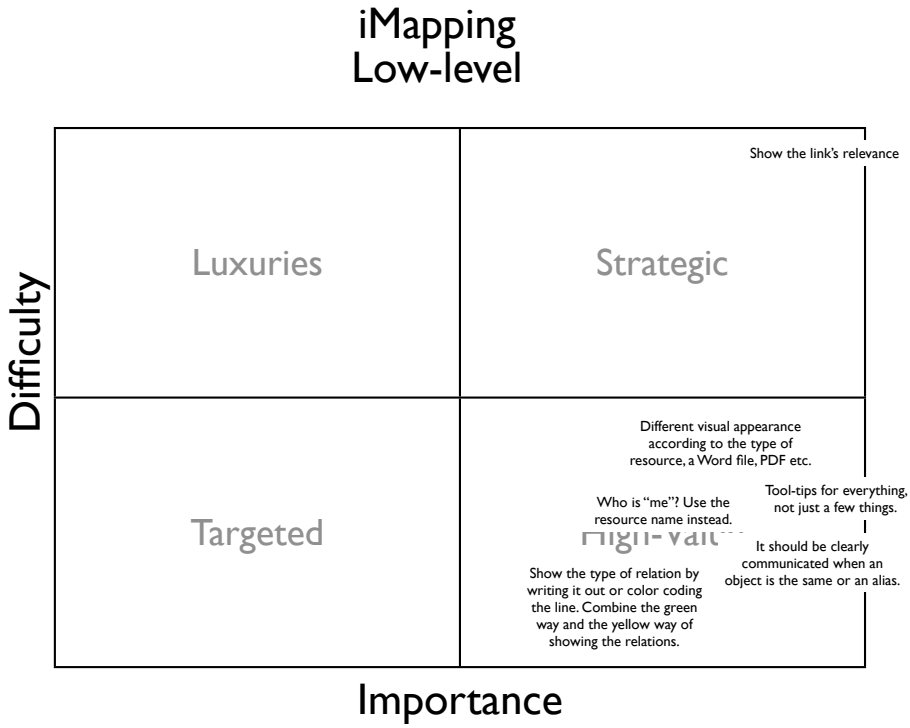

Figure 24: Problem prioritisation - Low-level

available and the cognitive load. A way to deal with this problem would be to let the user dynamically choose or control how many levels of relations that are shown. For instance; show me only my friends, not my friends friends.

- **Hint for hidden objects:** In the blue and red method you don't see the all the boxes if zoomed in. It could mean that related boxes are out of bounds. All the boxes that are related to an object should be visible or the user should be hinted about the fact that there are more relations and boxes.
- **Box-sizes:** What does the sizes of the boxes mean? Find a nice metaphor to make this meaningful or make 100% clear that the size means nothing.

Low-Level Results

- Show the link's relevance, e. g. by thickness (thicker line = stronger relation) or transparency (more transparency = weaker relation).
- There should be different visual appearance according to the type of resource. A word file must be visually different from a PDF, mp3 or a contact.
- In the green version: Who is "me"? Use the name of the object instead. In the prototype "me" refers to the very object. The personal pronoun "me" usually refers to a persona and not a resource that can represent a word document or a hotel booking.
- Usage of tool-tips is helpful for the users. But it needs to be consistent and used for all icons.
- It should be clearly communicated when an object is the same or an alias.
- Show the type of relation by writing it out, different line-types and/or colour-coding of the line, preferably in combination with text (at least initially).
- Combine the green way and the yellow way of showing the relations (c. f. Fig. 25). This could also be implemented together with the high-level issue of configurability.

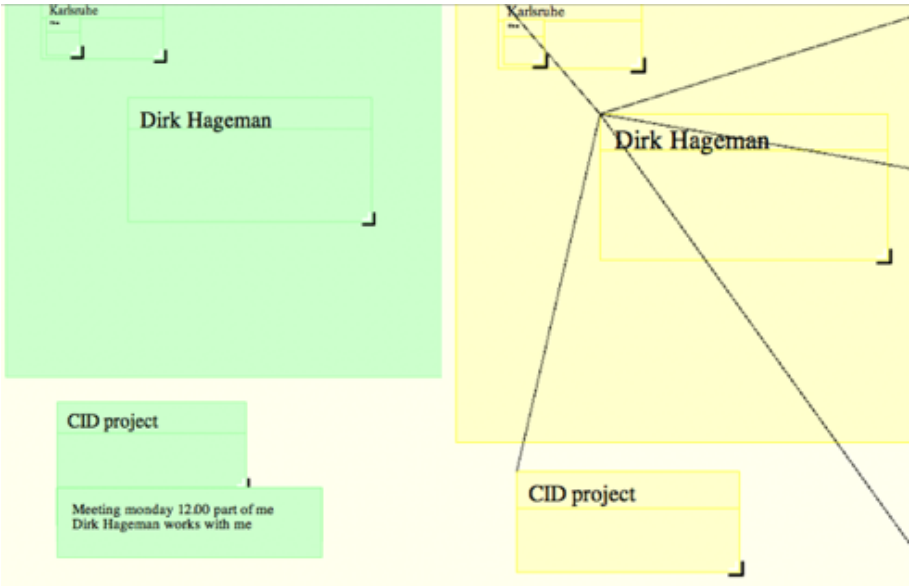

Figure 25: The green way and the yellow way of showing relations

4.1.6 End-User Evaluation (Java Prototype)

Additional to the expert-evaluation of the Flash prototype we conducted usability tests by exposing the Java GUI prototype to potential end users. The sample consisted of nine subjects between age 25 and 40 most of them with an academic background, none of them computer scientists.

After explaining the basic notion of a semantic desktop and of the iMapping approach to the testers, we showed them the running Java prototype pre-populated with some simple content, similar to Fig. 26. They were asked to play around with the program and were watched interacting. Special attention was paid to unsuccessful interaction attempts, since they reveal how a user would have expected certain interactions. When users asked how a certain function can be used or whether it is implemented, they were first told to try and find it on their own before getting a straight answer, so their intuitive actions could be observed. When a tester asked how an item could be opened, he was shown the mock-up of the new item design depicted in Fig. 18 and asked how he would interact with it.

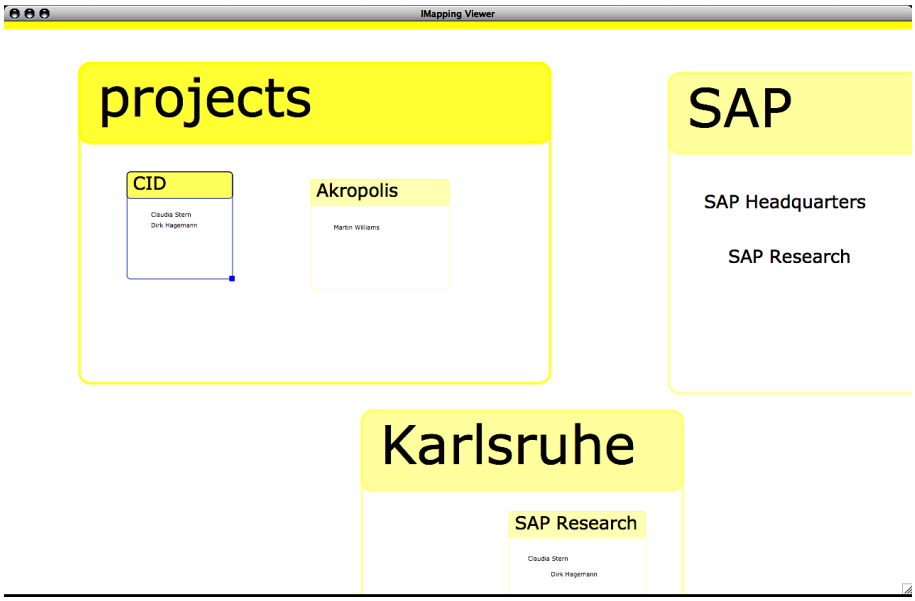

Figure 26: Screenshot of the Java GUI prototype used for usability tests

Results Apart from results that are due to the incomplete implementation and that will be addressed anyway within the limits of feasibility in nepomuk, these are some of the aggregated results concerning the interaction design:

What did not pose any problems and was usually found intuitive was

- zooming in general, esp using the scroll-wheel
- moving items around
- resizing items using the drag handle
- expanding / collapsing items using the triangle button of the new design (c. f. Fig. 18)

Problems, that frequently occurred were

- wish to gain overview (e. g. hierarchical tree, convenient zoom-out-action)
- lost in zoom: to far in / too far out to navigate reasonably

- expected to somehow 'open' (either expand or edit) an item on double click
- wish to automatically align items
- expected context menu on right-click
- wish to specify type of relationship between items and their children (consistent with findings in Sec. 4.1.5)

These findings, along with many other comments and suggestions we gained, will of course be considered in the UI design of the new Java-based full version.

4.2 QuiKey

QuiKey is a light-weight tool that can act as an interactive command-line for the Knowledge Workbench.

It combines some simple established interaction techniques like auto-completion, command interpreters and faceted browsing. QuiKey forms a generic, extensible UI, that can be used to browse, query but also author a knowledge base. Despite its versatility, QuiKey needs very little screen space, which makes it a candidate for possible future mobile use.

QuiKey is inspired by *quicksilver*[22], a kind of advanced application launcher for the Mac that has gained a lot of popularity due to its versatility and efficiency. Quicksilver can open files and applications and trigger a large variety of common actions not only on any files but also on specific information objects: Depending on the plug-ins installed, it can e. g. manage play-lists in iTunes, send a file via e-mail or dial a contact's phone number. A Semantic Desktop system however opens up some new applications areas for such tools, and QuiKey is being built to cover some of them.

Since, on a semantic desktop in general and in CDS in particular, knowledge can be modelled in a formal and fine granular way, it might be convenient to have tools that support browsing and editing the knowledge base in such fine-granular ways. It also aims at bringing simple ways of construction structures queries to the not-so-technically-advanced users.

4.2.1 Interaction

QuiKey is organised around the notion of *parts*. A part can be an existing item, a relation, a new text string or a command. Depending on the number, order and types of parts entered, it is decided what action to take.

Authoring To add a new text item to the knowledge base, it is enough to just type the text into the QuiKey console and press Enter.

To make CDS:statements about existing items, the statement can just be entered in a subject-predicate-object fashion, separated by tab-keys. So. e. g.

```
Claudia Stern→works for→SAP Research[enter]
```

would just add that statement. Only that the user would not even have to type in the whole sentence because all three parts are already known to the system which would in an auto-completion manner suggest the best fitting *NameItem* for each step. So for this example it is actually enough to type in

```
Cl→wor→SAP R[enter]
```

[22] http://blacktree.com/?quicksilver

In this way, the personal knowledge web can be woven in single simple steps in an ad-hoc fashion. Cognitive overhead is reduced to a minimum since additional actions and decisions that are not part of the actual content, like starting an application, opening a new document, choosing a file name and location, are not necessary anymore.

Browsing Simply navigating the knowledge base through its graph structure is possible with QuiKey without even changing into a different mode: when a part has been selected, before the user types anything new to select the next part, existing contents that fit the part pattern are already displayed in the suggestion area and can be browsed.

Queries / Search Semantic Desktop Systems bring the opportunity for complex structured queries, that can directly return a desired answer instead of just pointing out possible resources like conventional search applications. The problem is, that constructing complex, possibly nested queries is difficult, especially for non-expert users and a slight error in the syntax of the query makes the whole query fail or return unintended results. QuiKey tackles both of these problems:

Typos in the text should be easily recognised because there is an immediate feedback whether the typed-in string matches an existing NameItem or Relation. Besides typos are avoided to a certain extent because not many keystrokes are needed until the desired item can be picked from the suggestions. Syntax errors are avoided because instead of requiring the user to write a whole query in some complicated syntax which is then later parsed, in QuiKey the query is constructed interactively. Because of this **interactive construction of queries**, there are no syntactical characters.

To facilitate the construction of complex queries, we allow the **modular construction of queries**. A query can be saved or referred to as a special QueryItem. Simple QueryItems can be constructed with the simple pattern shown the first two query examples in Fig. 27: `KA inhab→lives in→Karlsruhe[enter]` creates a new QueryItem that represents a query about everything living in Karlsruhe. More complex queries can be constructed by combining existing QueryItems like in the examples 3 to 5 in Fig. 27.

4.2.2 Current State of Implementation / Future Work

The QuiKey idea is quite young and not very far implemented yet. Simple functionalities are already implemented and include:

- Creating and deleting *NameItems*
- Creating and deleting *Relations* with their inverse Relations
- Creating and deleting *Statements*
- searching and displaying *Items*, *Relations* and *Statements* by partial string matches.

Browsing and querying functionality is expected to be available during the first half of 2008.

4.2.3 Integration / Possible Applications of QuiKey

While the general approach of QuiKey could be extended to more of less all of a semantic desktop, or tried to be integrated with an existing non-semantic tool like Quicksilver, this would add a lot more complexity – especially when it is used

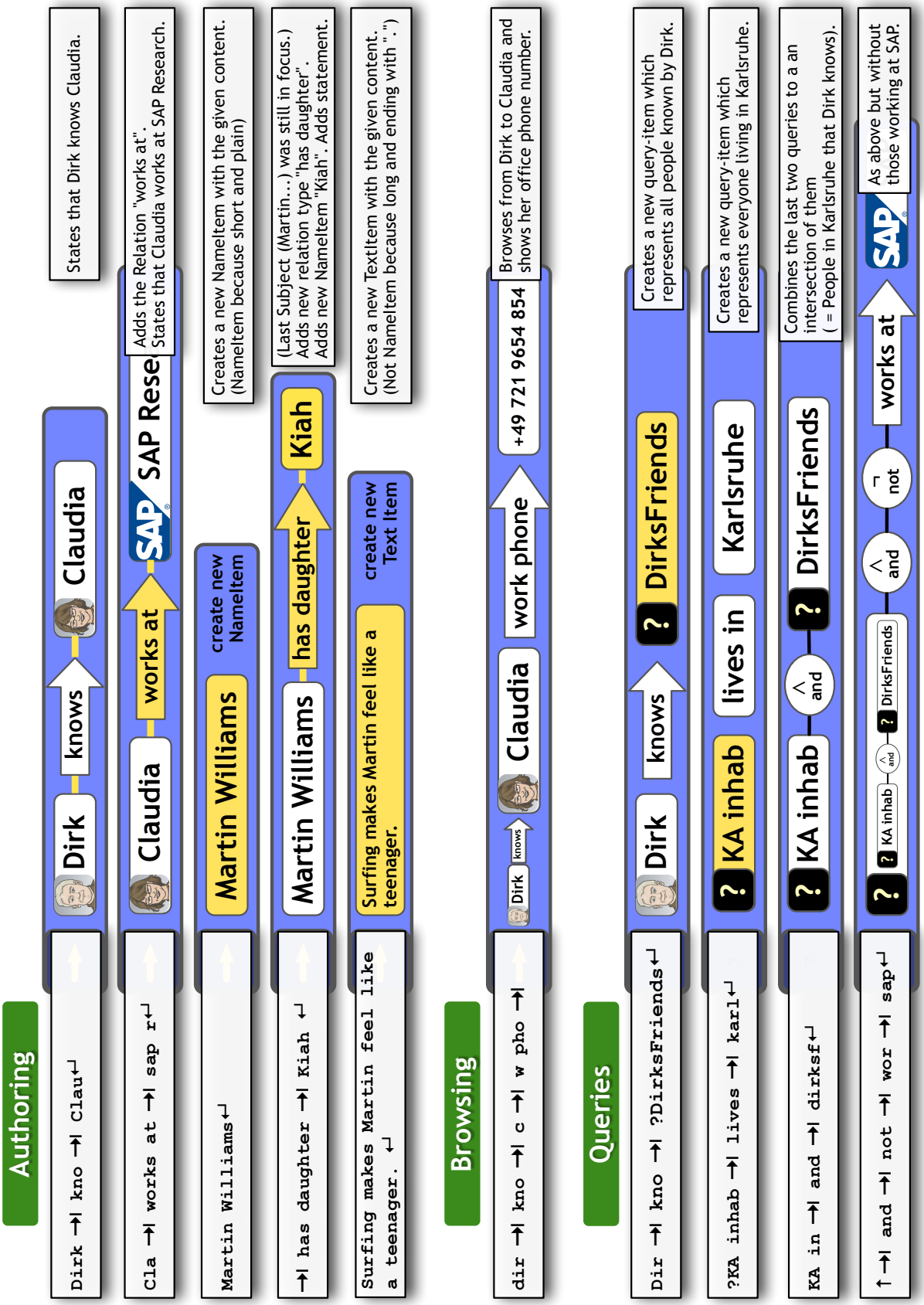

Figure 27: Mock-up of some QuiKey actions including keystrokes necessary (left) and explanation of the action (right).

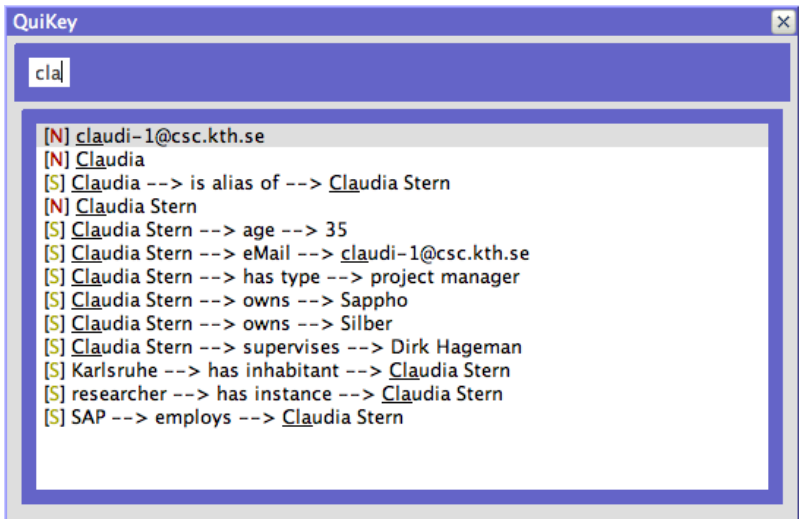

Figure 28: Screen shot of the current QuiKey implementation showing a list of matches to the string "cla".

for authoring and it is not clear in which of the connected data sources the newly added content should go. So, for now, QuiKey is designed to run on CDS only.

Where useful, we will also integrate with the iMapping client, e.g. to open an existing Item directly in an iMap or to 'summon' an existing item into a specific place in an iMap.

Some modules of QuiKey will also be used in the iMapping client e. g. to select a relation type when connecting two items visually. There the auto-completion and ranking mechanisms of QuiKey can be reused.

For current information on QuiKey see `http://quikey.info/`

5 Natural Language Tools

"Applied Natural Language Processing or Applied NLP involves "the construction of intelligent computational artifacts that processing natural languages in ways that are useful to people other than computational linguists"Cunningham (1999), Gazdar (1996). Applied NLP is quite close to the definition Language Engineering (LE) or Natural Language Engineering (NLE) which is the "applied component of computational linguistics which focuses on the practical outcome of modelling human language use" Cunningham (1999). LE also implies the instrumental use of Natural Language Processing within a larger system with some practical goal Cunningham (1999).Knowledge Acquisition via Semantic Annotation and Ontology authoring is vital to the growth and success of the Semantic Web Initiative. More importantly, the majority of web documents as well as legacy intra and inter -desktop data in the context of Nepomuk contain either free or partially structured text. Consequently the application of LE plays a crucial role in easing the constriction inherent to the knowledge acquisition bottleneck by providing support for the Ontology authoring and Semi-Automatic/Automatic Semantic annotation.

5.1 Text Miner and Semantic Analysis Component

Text miner and semantic analysis component is used to process natural language texts and generate metadata based on the semantic relations within the text.

Due to architecture requirements the decision was made to consolidate the design of the semi-automatic semantic analysis component (WP1) and the text mining and semantic extractor component (WP2) into one Text Miner and Semantic Analysis component.

The text analysis component can be used in a variety of ways:

- In WP1, it is currently being used to conduct semi-automatic semantic processing of Wiki documents and furthermore enhance their presentation to the user with links to descriptions of detected concepts.
- In WP2, it is used for automatic extraction of mentions of semantic concepts and the enrichment of document's metadata. It can be used to facilitate document and resource navigation and search.
- However, the usage of the text analysis component is not limited to these scenarios, and it may be used by any future NEPOMUK component.

Within the NEPOMUK implementation itself, text analysis is implemented by "Structure Recommender", component (*Comp-StrucRec* [23]).

The text analytics component consists of 2 subsystems:

1. *Language Processor API* is used to perform linguistic analysis of natural language texts. It includes generic functions like text tokenization, sentence detection, detection of lexical expressions, including multi-word expressions, part of speech detection, morphological normalization, etc.

   Lexical information associated with semantic concepts is extracted from RDFRepository.

2. *Semantic Processor API* is used to perform semantic analysis of lexical expressions in natural language documents discovered by Language Processor. This semantic analysis is utilising ontological knowledge stored in RDFRepository. Semantic analysis provides the following:

   - Automatic disambiguation of lexical expressions found in the text. The result is the mapping from the words or phrases in the text to the concepts in the ontology these words denote.

[23] http://nepomuk.semanticdesktop.org/xwiki/bin/view/Main/Comp-StrucRec

- Detection of a topic or focus of the analysed text document. The focus is a set of concepts from the ontology which are found to be the most central to the document. The focal concepts may not be directly referred in the text, but instead implied from context. The focus of a document may be used to perform classification or tagging of documents, or locate the most important keywords related to the document (even those keywords which did not appear in the text).

The Semantic Processor API uses an algorithm based on the spread of activation technique (Quillan 1966) used in semantic networks. Activation mapping is typically provided by the Language Processor, however the Semantic Processor API can be called directly. The spread of activation in the ontology is affected by a number of parameters, all of which can be controlled allowing it to be tuned to suit specific tasks and needs. The parameters which we use to configure the spread of activation are:

- The minimum activation level required for activation to spread onwards
- The maximum distance (number of link traversals) activation can spread
- The decay rate of the activation as it spreads. This can be further customised on a per link/node basis allowing certain types of links to be considered weaker/stronger or furthermore allowing nodes to act as sinkholes for activation to prevent it from spreading further.
- Link direction; directed and undirected links can affect the spread in different ways
- A parameter that sets the averaging of activation over outgoing links from a node. This parameter can be enabled, disabled or used with a particular coefficient.

Initial activation to nodes in the ontology (which sets the whole process in motion) is provided by language processing of text, which maps term mentions within text to concepts.

In text analysis initial activation supplied by language processing provides starting points for the algorithm which represent concepts mentioned in a text. As this spreads around the ontology other nodes receive activation and the activation accumulates in certain nodes. This provides a basis for determining a conceptual focus, which in the analysis of a long text or discourse is used to help disambiguation of lexical entries which correspond to multiple senses in the ontology.

By varying the parameters used during spreading activation, different results can be achieved, which allows the semantic processing to be adapted to different tasks and to be used by other components as necessary. For some more complicated tasks such as Socio-Semantic Analysis a layered use of the technique can be applied (Kinsella, Harth, Troussov, Sogrin, Judge, Hayes & Breslin 2007) cascading the results of one analysis run onwards over the next, we can abstract away from individual concepts and concentrate on the focus for the next iteration of processing.

Therefore, by processing documents in this manner, we create an abstract semantic model of the document where activation in nodes represents the level of importance of certain concepts to the overall discourse. Using this modelling as a substitute for keyword analysis and combining it with standard information retrieval similarity metrics like Cosine Similarity Measure we have implemented Comp-StrucRec, which performs a sort of concept-based search. This component can return results based on similarity to concepts which are semantically close to those found in documents/queries but which are not explicitly present. This allows us to increase recall of relevant documents and to make a more accurate assessment of the "aboutness" of a document when searching for "more like this".

5.1.1 Language Processing Support Services

The Language Processor API aims to first of all to provide linguistic
textual analysis of text documents for the aforementioned Semantic Processor API. In addition other specialised linguistic processing and language generation functionalities are also available services for WP1 and WP2 such as:

1. The identification of most import keywords in a document5.1.2,
2. The identification of speech acts within a given text5.1.3. Speech acts consist of questions, statements, orders, requests for permission, requests for information or an action, etc.

5.1.2 Keyword Extraction

Keyword Extraction Service Keyword Extraction is the identification of most import keywords in a document. The Keyword Extraction service can cater for various use cases such as: the generation of candidate semantic tags or heuristic support for the Semantic Processor API. Two algorithms are currently available[24]. In NEPOMUK the implementation of text analysis is implemented by '"Related Item Recommender" component (*RelItemRec*). **Algorithm 1** The following algorithm is implemented purely in Java and is independent of any Open source NLP platform, consequently it can be embedded directly into any NEPOMUK application or the Semantic Analysis Component 5.1

1. Read a URL address input by the user.
2. Open an active http connection.
3. Remove the stopwords[25] from the textual content of the website provide by the http connection.
4. Read in the remaining content as a string.
5. Strip all of the html code from the string which was read in.
6. Tokenize this string.
7. Place each word in the string and its frequency into a hash-table. The frequency is the word's weight.
8. Increase the frequency (and therefore weight) of words that are in the title and words which consist of one or more uppercase characters, by adding half the word.s frequency to its original frequency.
9. If the frequency is greater than 4, return the word and its frequency to the user

Algorithm 2 - Kea - Automatic Keyphrase Detection

Kea is a tool for automatic detection of key phrases developed at the University of Waikato in New Zealand. The home page of the project can be found at [26]. To promote interoperability with other Language Processing services within Comp-StrucRec and the "Related Item Recommender" component (*RelItemRec*). and across NEPOMUK, the KEA tool as had been made available using the GATECunningham, Maynard, Bontcheva & Tablan (2002) framework. GATE contains a UIMA Interoperability layer. UIMA (Unstructured Information Management Architecture) is a platform for Natural Language Processing developed by IBM. It

[24] These services have been provided by Hewlett-Packard Galway and DERI NUIG

[25] *stop words* are those words which are so common that they are useless to index or use in search engines, i.e a, the, in, of, is performed.

[26] http://www.nzdl.org/Kea/

is very similar to the GATE architecture in that it represents documents as text and annotations. Processing Resources in UIMA are called *analysis engines* than can manipulate UIMA documents or CAS (*Common Analyis Structure*) as they known within UIMA terminology. UIMA analysis engines behave similarly to GATE Processing Resources. GATE's UIMA interoperability layer permits the embedding of GATE PRs within UIMA Text analysis Engines (*TAEs*)such as IBM's Text Analysis and Semantic Analysis Component 5.1 described earlier. In the context of KEA - outputted GATE annotations can be mapped over to CAS annotations to be included as additional parameters to tune *spreaded activation* within IBMs Semantic Analysis Component5.1.KEA is realised as a GATE Processing Resource(PR). Kea is based on machine learning and must be trained before it can be used to extract keyphrases. In order to do this, a corpus is required where the documents are annotated with keyphrases. KEA is available as part of the Keyword Extraction Service. A preselected corpus suitable for NEPOMUK has been compiled and used to construct a training model for the KEA NEPOMUK service. A stand-alone version of the tool, as a GATE KEA Processing Resource is also available, should users wish to tune the training model and subsequently the KEA tool to their needs. As mentioned earlier a corpus is needed to construct a training model for KEA. Corpora in the Kea format (where the text and keyphrases are in separate files with equivalent names but different extensions) can be imported into GATE using the "KEA Corpus Importer" tool. The usage of this tool is clearly documented in the GATE user Guide[27]. Furthermore, manually annotated corpora for such training purposes are available from the KEA Project Page above.

5.1.3 Speech Act identification

When provided a given text, the Speech Act Identification service will return a set of Speech Act Annotations. This service is was designed specifically to support the prototype for the Semantic Email Client - *Semanta*7.

Text is annotated at the sentential level using the GATE IE engine – ANNIE (A Nearly New Information Extraction Engine, Cunningham et al. (2002)). Finite State Parsing is applied over annotations and text tokens - given a set of hand-coded JAPE grammars. Sentences are annotated within email text as instances of speech acts. In addition a collection of hand written gazetteers or lists of words and key phrases are used to aid the parsing process. A Speech Act Ontology is used to model the Knowledge Acquisition Rules (KA) which from the basis of the Languages Resources used in GATE. The KA rules are refined on an iterative basis by the Language Engineer and Knowledge Engineer and changes are integrated into the next version of the Speech Act Service. The initial development was inspired be Khosravi & Wilks (1999) based on earlier versions of GATE. The service is currently at the alpha stage and will released simultaneously to NEPOMUK in conjunction with D1.2. A beta release will occur in Month 25 in conjunction with first "Semanta" Prototype Scerri (2007).

5.1.4 Text Analysis and CDS

To verify that the text analysis componentcan deal with CDS, we created a CDS demo model that covered most information of the persona descriptions of the personas Dirk, Claudia and Martin. We took several texts and let the text analysis componentanalyse them against this demo CDS model. Among these texts were the original persona descriptions, some scenario descriptions related to these personas and some short texts written to test specific features of the component like disambiguation or the recognition of not explicitly mentioned foci.

The disambiguation feature can be very useful for ambiguous vocabularies. But

[27] http://gate.ac.uk/documentation.html

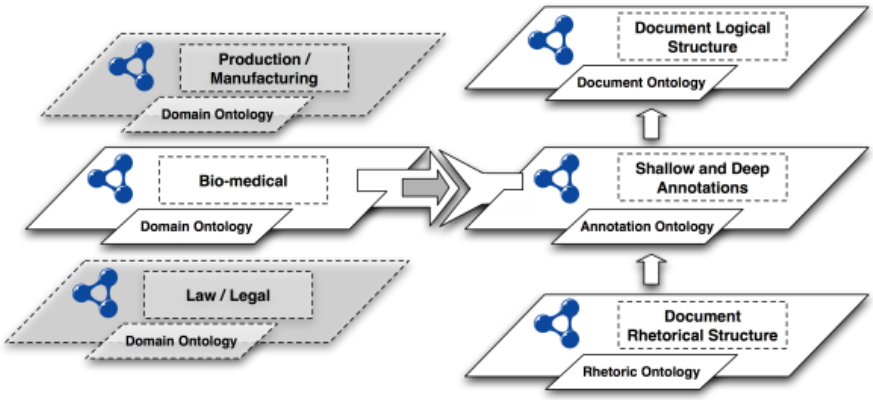

Figure 29: Ontology Layers in SALT

it is not needed for use with cds, since NameItems are already unique. Our tests showed that, with the right parameter settings, the text analysis componentcomes up with text foci, even if they were not explicitly mentioned in the text. While these foci still contained some errors, the results were good enough e. g. to act as proposed tags, which is envisioned for an NLP-based tagging tool. However, to make full use of the CDS-semantics like aliases and inverse relations, it would need either need some internal adaptions to CDS or a special export prior to generating the vocabulary.

5.2 Semantic Authoring with SALT

Semantic Authoring aims to a framework for semantically annotating scientific publications. The first approach that we will follow will be based on the Latex writing environment and will produce Semantic PDF Documents. This instantiation of the framework is called 'SALT", for Semantically Annotated LaTeX. It is an authoring framework for creating semantic documents for scientific publications.

SALT, described in Groza, Handschuh, Möller & Decker (2007), Groza, Möller, Handschuh, Trif & Decker (2007), is a system which allows authors to explicate several kinds of information (see Fig. 29 and see also Sec. 2.5.1).

SALT comprises two layers: (i) a syntactic layer, and (ii) a semantic layer. The syntactic layer represents the bridge between the semantic layer and the hosting environment, i. e. LaTeX. It defines and series of new LaTeX commands, while making use of some of the already existing ones in order to capture the logical structure of the document and the semantics present in the publication's content. As a remark, we chose LaTeX because is one of the most widely used writing environments in the scientific world. In a similar way, the SALT framework can be used also together with, for example, Microsoft WinWord.

The semantic layer is formed by a set of three ontologies: (i) The Document Ontology – modelling the logical structure of the document (SALT extracts the existing structure of a LaTeX document and converts document parts into addressable snippets), (ii) The Rhetorical Ontology – modelling the rhetorical structure of the publication, and (iii) The Annotation Ontology – linking the rhetorics captured in the document's content and the document itself; it also models the publication's shallow metadata. Detailing a bit further the Rhetorical Ontology, it has three sides: (i) a side modelling the claims and supports present in scientific publications and the rhetorical relations connecting them, (ii) a side modelling the rhetorical block structure of the publication, and (iii) the last one, modelling the argumentative discourse span over multiple publications, and seen as a communication between different authors, the current stage being described in the publication modelled

by the current ontology instances. As LaTeX (like almost all document creation systems) allows only to represent consistent strict trees, it might be beneficial for a user to be able to start editing the structure of a document in a CDS tool and export to SALT later. To be able to continue editing in SALT allows to keep and refine all formal structures.

5.3 Human Language Technology(HLT)

HLT for Knowledge Creation implies the use of Human Language Technology (synonymous with Applied NLP) specifically Controlled Natural Language Technology to author Ontologies and create metadata via Controlled Language Semantic Annotation within specific user cases such as status report writing, note taking and recording meeting minutes on the semantic desktop.

HLT for Knowledge access implies the reverse process, whereby Natural Language Generation (NLG) can be applied to

- Ontology authoring and the Ontology development life-cycle – to provide feedback to the domain expert for the purposes of implementing quality assurance over a newly created Ontology or text generation of Ontology version "diffs" to allow user friendly inspection and finally
- automatic text summary generation from Knowledge within the semantic desktop data store based on user specified query. Use cases include - report summary generation based on semantically annotated status reports or goal specific publication summarisation for users conducting a literature review or search across SALT annotated publications on the desktop.

NLG can be targeted to CDS. The initial research described above was developed as part"Related Item Recommender" component ST1240 (*RelItemRec*) in Year 1 and 2 of the NEPOMUK Project, primarily due to the overlap with NLP related work. This work has resulted in the creation of a new component ST1430 Semantic email, blogging and Authoring to be completed for WP1000 Deliverable 1.3

5.3.1 Ontology authoring using Controlled Natural Language

A form of Simplified English or Controlled (Natural) Language (CL) is used to create and author an ontology. In addition, instances of concepts can be described to populate the ontology using CL. The CL text will then be translated into the targeted underlying ontology. The input will consist of CL text input only and the type of ontology to be targeted too. The output will consist of a reference to newly created and populated ontology. The CL syntax analyzer in conjunction with the in vivo ontology will guide the user with respect to legal CL input. See Tablan, Polajnar, Cunningham & Bontcheva (2006) for further reading.

5.3.2 Text generation of Ontologies

Given a reference to an ontology the inverse of 5.3.1 will be applied. Thus a text summary, generated with Controlled (Natural) Language or CL, of a given ontology and its populated instances will be returned to the user. This will involve traversing the ontology and generating CL text given a selection of XML templates, which will contain snippets of CL that will, given a concept or instance, combine into a CL sentence and finally a CL based textual summary of a the ontology. This process in conjunction with 5.3.1 can be used to edit a CL textual summary of an existing ontology in order to regenerate or amend/edit an existing ontology repeatedly to achieve the desired output. The process of merging 5.3.1 and 5.3.2

is called round-trip ontology authoring Tablan et al. (2006).

The technology used for 5.3.1 and 5.3.2 is based on ClOnE - Controlled Language for Ontology Editing and CLIE - Controlled Language Information Extraction

These technologies have been made available as part of the existing collaboration between DERI Galway, National University of Ireland, Galway and the Sheffield NLP group, University of Sheffield. An initial evaluation was conducted by DERI Galway in collaboration with the Sheffield NLP group involving a comparison between an existing Ontology Engineering Tool - Protege and CLOnE. The results were favourable and further details with respect to the implementation and evaluation can found here Funk, Tablan, Bontcheva, Cunningham, Davis & Handschuh (2007).

6 CDS and Wikis

Many ideas in CDS stem from wiki concepts. Wikis are fast editors to create hypertext, due to their simple naming scheme and the ability to link to not-yet-existent pages. Wikis also focus on structure, not visual formatting of content. Most wikis use wiki syntax, which is a convenient way to type text and structures at the same time.

For the next version of CDS and the HKW we plan to integrate wiki syntax. Wiki syntax in CDS is used to enter text, structures and formal statements. The WikiModel, an open source component developed by Cognium Systems, fulfils all needs. In this section we first describe the WikiModel and its syntax in depth, so that this deliverable can act as a reference handbook for the syntax.

The Structured Text Interchange Format (STIF) is presented. It will be used in future versions of HKW to persist the content of items.

We present extensions to the WikiModel syntax to allow entering formal statements.

Finally, we describe BounceIt, a Web 2.0 service built with WikiModel.

6.1 WikiModel 2.0

WikiModel is a data model defining the structure of wiki documents. As such, it defines wiki document elements and their containment rules (like a DTD or XML Schema for XML documents).

Second, *WikiModel is an API* providing access to the structure of wiki documents. This API gives access to and control over the internal structure of individual wiki documents. Usage of this API guaranties that the accessed wiki documents respect the structure defined by the model.

WikiModel is not a complete "wiki engine", it does not contain data storage, versioning or access rights. Further, WikiModel does not check or validate references between documents.

WikiModel can be used as a

- rendering engine to transform various wiki syntaxes to formatted content (HTML, PDF, TEX, ...), or
- parser for semantic annotations.

6.1.1 Design

In this section we briefly review the design and evolution of WikiModel. There are currently two versions. Figure 30 shows the general way how WikiModel works:

- Wiki source syntax is parsed and generates a stream of wiki events (WEM).
- This stream can be converted by a wiki document builder into an in-memory representation, the wiki object model (WOM). This is similar to the generation of an XML document object model (DOM) from a stream of SAX-events.
- Alternatively, a stream of wiki events can be converted into different target syntaxes. One possible target syntax is back to source wiki syntax. This allows round-trip editing.
- The WOM can also be converted back to a WEM stream. This allows to create rich-text editors to work on the WOM, that still remain compatible with wiki syntax.

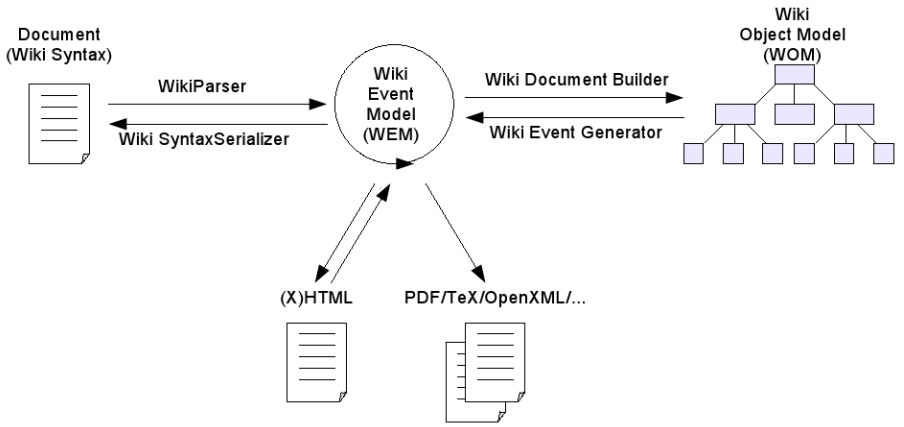

Figure 30: WikiModel – The Big Picture

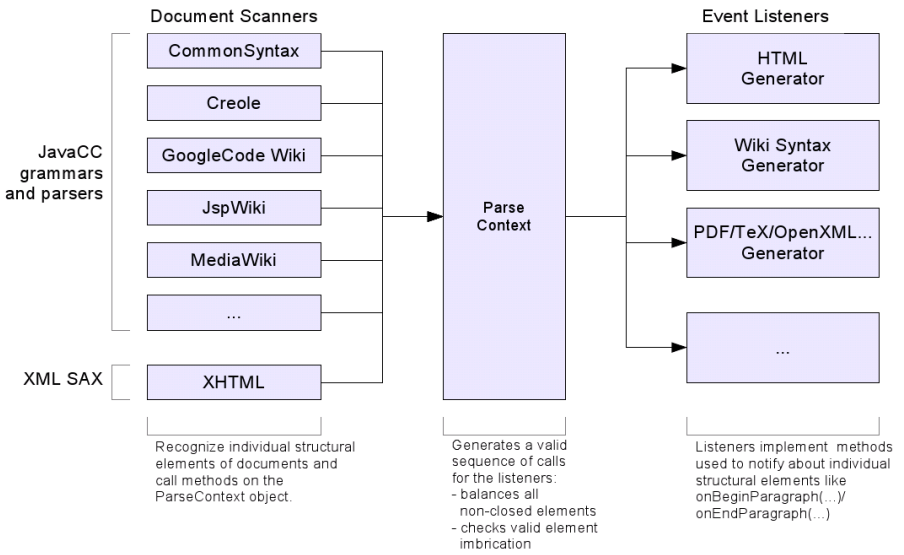

Figure 31: WikiModel Version 2

The design of WikiModel can be described on three levels:

WikiModel Schema: The data model or document schema is the conceptual core. It defines what elements exist and how they can be nested. The WikiModel schema is sufficiently flexible to simulate almost any HTML formatting (e. g. even tables with embedded lists, headers and paragraphs). This goes beyond the expressivity of most existing wiki syntaxes.

Yet the resulting structure is much simpler than XHTML, which greatly simplifies the validation and manipulation of documents.

WikiModel can represent semantic statements about complete documents and parts of a document.

The document schema is a super-set of structural elements existing in others wikis, so the information from almost any wiki can be imported without loosing information or structure.

WikiModel Syntax: The WikiModel CommonSyntax has been extended to handle a greater number of structural elements. It allows to manipulate *all* elements defined in WikiModel. The CommonSyntax is described in detail in the next section.

WikiModel Parsers: Parsers for multiple wiki syntaxes are available (JspWiki, XWiki, MediaWiki, Creole, GoogleCode wiki, ...). Authors may thus use their syntax of choice. An overview of the parsing is given in Fig. 31: A source syntax is interpreted in a parser context and routed to an event listener.

All parsers give access to a valid structure of documents, i. e. if a document contains non-valid elements (non-closed markup or overlapping elements) then such errors will be fixed automatically on-the-fly.

6.1.2 Complete Syntax Description

The CommonSyntax of WikiModel provides support for normal wiki structures (e. g. lists, tables, inline formatting), embedded documents, semantic properties and macros.

The syntax of WikiModel allows for basic inline text formatting like most wikis do (c. f. Fig. 32). To give a user the option to prevent WikiModel parsers from interpreting a symbol, every symbol can be escaped via a preceding backslash character ("\").

To allow for serious usage of wiki syntax, WikiModel supports e. g. a number of typographic symbols as well as mathematical symbols. A complete list of special symbols is depicted in Fig. 33.

Text can be structured with various kinds of lists, as depicted in Fig. 34 and tables. Support for tables is pretty extensive: A user can use header cells, body cells as well as adding properties to individual cells or rows (c. f. Fig. 35). Different from most wikis, tables can contain other nested tables or other arbitrary sub-document, as shown in Fig. 36.

To support editorial processes, WikiModel supports so called information blocks (see Fig. 37), which are snippets of text rendered with a special attention-catching symbol. These information blocks can contain sub-documents on their own.

Links to external entities or other wiki pages are made via square brackets ("[","]"). WikiModel does not define how such links are interpreted. Words separated via a colon (":") with without any whitespace (e. g. like in `wikipedia:Berlin`) are reported as links automatically as well. WikiModel distinguishes between these automatic links and manual links, which use square brackets – possibly also using a colon.

Text Formatting

`*bold*`	**bold**
`__italic__`	*italic*
`text~sub~`	text$_{sub}$
`text^sup^`	text^{sup}
`++big++`	big
`--small--`	small
`@@insert@@`	insert
`##delete##`	~~delete~~

Examples:

`H~2~O`	H_2O
`H~2~SO~4~ <=> 2H^+^ + SO~4~^2-^`	$H_2SO_4 \Leftrightarrow 2H^+ + SO_4^{2-}$

Escaping

Any symbols (including special ones) can be "escaped" using the `\` (back slash) sign. To put the backslash in the text the sequence `\\` (two consequtive backslashes) should be used.

Figure 32: WikiModel Syntax for Basic Inline Formatting

Syntax Example Fig. 38 shows an example using most features. The example starts with three lines stating semantic properties about the complete document, here: type, title and summary. Next a headline is rendered, followed by a list. The next paragraph in the example contains an *inline property* (via the syntax "%", explained in the next paragraph). The sample document ends with a table, which contains an embedded document (marked with "(((" and ")))") in cell 2.2.

Using the default formatter, the example in Fig. 38 results in the following HTML code:

```
<div class="content" id="content"><div class="doc">

<div class="property" url="rdf:type">
    <p><a href="toto:Document">toto:Document</a></p></div>

<div class="property" url="title"><p>Hello World</p></div>

<div class="property"
     url="summary"><p>This is a short description</p></div>

<div class="section section-level-1">
  <h1 class="section-title">Hello World</h1>
  <div class="section-content">
    <ul>
        <li>item one
            <ul><li>sub-item a</li>
                <li>sub-item b
                    <ol><li>ordered X</li>
                        <li>ordered Y</li>
                    </ol>
                </li>
                <li>sub-item c</li>
            </ul>
        </li>
        <li>item two</li>
    </ul>
```

Special Symbols

`text - text`	text - text
`text -- text`	text – text
`text --- text` or `text---text`	text — text
`<<text>>`	«text»
`text...`	text…
`(E)`	€
`(c)`	¢
`(Y)`	¥
`(L)`	£
`(P)`	¶
`(s)`	§
`(S)`	∑
`25(o)C`	25°C
`(8)/(8) => (8)`	∞/∞ ⇒ ∞
`CogniumSystems(C)`	CogniumSystems©
`TapTipTop(tm)` or `TapTipTop(TM)`	TapTipTop™
`A (*) B`	A • B
`A +/- B`	A ± B
`A != B`	A ≠ B
`A -> B` or `A --> B`	A → B
`A <- B` or `A <-- B`	A ← B
`A <-> B`	A ↔ B
`A => B` or `A ==> B`	A ⇒ B
`A <= B`	A ≤ B
`A <== B`	A ⇐ B
`A <=> B`	A ⇔ B
`A >= B`	A ≥ B
`A != B`	A ≠ B
`A ~= B`	A ≈ B

Figure 33: WikiModel Syntax for Special Symbols

Lists

Ordered lists

```
+ first ordered item
+ second ordered item
   (it has multiple lines)
```

1. first ordered item
2. second ordered item
   (it has multiple lines)

Unordered lists

```
- item one
- item two
  (it has multiple lines)
```

- item one
- item two
  (it has multiple lines)

or:

```
* item one
* item two
  (it has multiple lines)
```

- item one
- item two
  (it has multiple lines)

Definition Lists

```
? Term A
! Definition A
? Term B
! First Definition B
! Second Definition B
```

Term A
: Definition A

Term B
: First Definition B
: Second Definition B

```
? What do you think
   about this project?
! I think that it is a very
  interesting project!
```

What do you think about this project?
: I think that it is a very interesting project!

Mixed Lists

```
+ item one
+ item two
  - subitem A
  - subitem B
+ item three
  ? Term X
  ! The full term X description
  ? Term Y
  ! The full term Y description
+ item four
```

1. item one
2. item two
   - subitem A
   - subitem B
3. item three
   Term X
   : The full term X description
   Term Y
   : The full term Y description
4. item four

Figure 34: WikiModel Syntax for Lists

Tables

Simple Tables

```
!! Header 1.1 !! Header 1.2
:: Cell 2.1 :: Cell 2.2
```

Header 1.1	Header 1.2
Cell 2.1	Cell 2.2

Simple tables with parameters

```
!!{{colspan=2}} Table Header
:: Cell One :: Cell Two
```

Table Header	
Cell One	Cell Two

See also the next section for more information about formatting of tables.

Complex Tables formatting

The central cell the table has specific parameters:

```
:: C-1.1 :: C-1.2 :: C-1.3
:: C-2.1 ::{{bgcolor=#FAFAD2}} C-2.2 :: C-2.3
:: C-3.1 :: C-3.2 :: C-3.3
```

C-1.1	C-1.2	C-1.3
C-2.1	C-2.2	C-2.3
C-3.1	C-3.2	C-3.3

The second row of the table has parameters:

```
:: C-1.1 :: C-1.2 :: C-1.3
{{bgcolor=#FAFAD2}}:: C-2.1 :: C-2.2 :: C-2.3
:: C-3.1 :: C-3.2 :: C-3.3
```

C-1.1	C-1.2	C-1.3
C-2.1	C-2.2	C-2.3
C-3.1	C-3.2	C-3.3

A table with parameters:

```
{{bgcolor=#FAFAD2}}
:: C-1.1 :: C-1.2 :: C-1.3
:: C-2.1 :: C-2.2 :: C-2.3
:: C-3.1 :: C-3.2 :: C-3.3
```

C-1.1	C-1.2	C-1.3
C-2.1	C-2.2	C-2.3
C-3.1	C-3.2	C-3.3

All elements with parameters:

```
{{bgcolor=#FAFAD2}}
:: C-1.1 :: C-1.2 :: C-1.3
{{bgcolor="#7FFFD4"}}::{{colspan=3}} C-2.1
:: C-3.1 ::{{bgcolor="#EEEEEE"}} C-3.2 :: C-3.3
```

C-1.1	C-1.2	C-1.3
C-2.1		
C-3.1	C-3.2	C-3.3

Figure 35: WikiModel Syntax for Simple Table Formatting

Complex Table Formatting

Note that table can put any structrual elements in tables using the (((and))) symbols.

```
!! Head 1.1 !! Head 1.2 !! Head 1.3
:: Cell 2.1 :: (((
= Hello, world!
* one
* two
))) :: Cell 2.3
:: Cell 3.1 :: Cell 3.2 :: Cell 3.3
```

Head 1.1	Head 1.2	Head 1.3
Cell 2.1	**Hello, world!** ▪ one ▪ two	Cell 2.3
Cell 3.1	Cell 3.2	Cell 3.3

```
!! Head 1.1 !! Head 1.2 !! Head 1.3
:: Cell 2.1 :: (((
This is an internal "document" containing
additional block elements.
- item a
  + item one
  + item two
  + item three
- item b
!!{{colspan=2}} Header
:: X :: Y

A paragraph after the embedded table.
))) :: Cell 2.3
:: Cell 3.1 :: Cell 3.2 :: Cell 3.3
```

Head 1.1	Head 1.2	Head 1.3
Cell 2.1	This is an internal "document" containing additional block elements. ▪ item a 1. item one 2. item two 3. item three ▪ item b **Header** X | Y A paragraph after the embedded table.	Cell 2.3
Cell 3.1	Cell 3.2	Cell 3.3

Figure 36: WikiModel Syntax for Complex Table Formatting

Information Blocks

Information blocs can be used to select a block of information and to make it more visible on the page. There are the following types of blocks:

```
/*\ *Simple*
A normal text block
```

Simple
A normal text block

```
/!\ *Exclamation*
A simple text requiring
special attention to it.
```

Exclamation
A simple text requiring
special attention to it.

```
/i\ *Information*
Simple
information block
```

Information
Simple
information block

```
/w\ *Warning*
Warning message
```

Warning
Warning message

```
/x\ *Error*
This is forbidden!
```

Error
This is forbidden!

It is possible to include structured content in information blocks using embedded documents. Each embedded document can contain all markup elements available for the root document.
Each embedded document is delimited by the symbols (((and))).

```
/i\ *Any combinations are possible!* (((
This is an internal document
/!\ *Exclamation!*
+ item one
+ item two
  - a
  - b
+ item three
)))
```

Any combinations are possible!
This is an internal document

Exclamation!

1. item one
2. item two
   - a
   - b
3. item three

Figure 37: WikiModel Syntax for Information Blocks

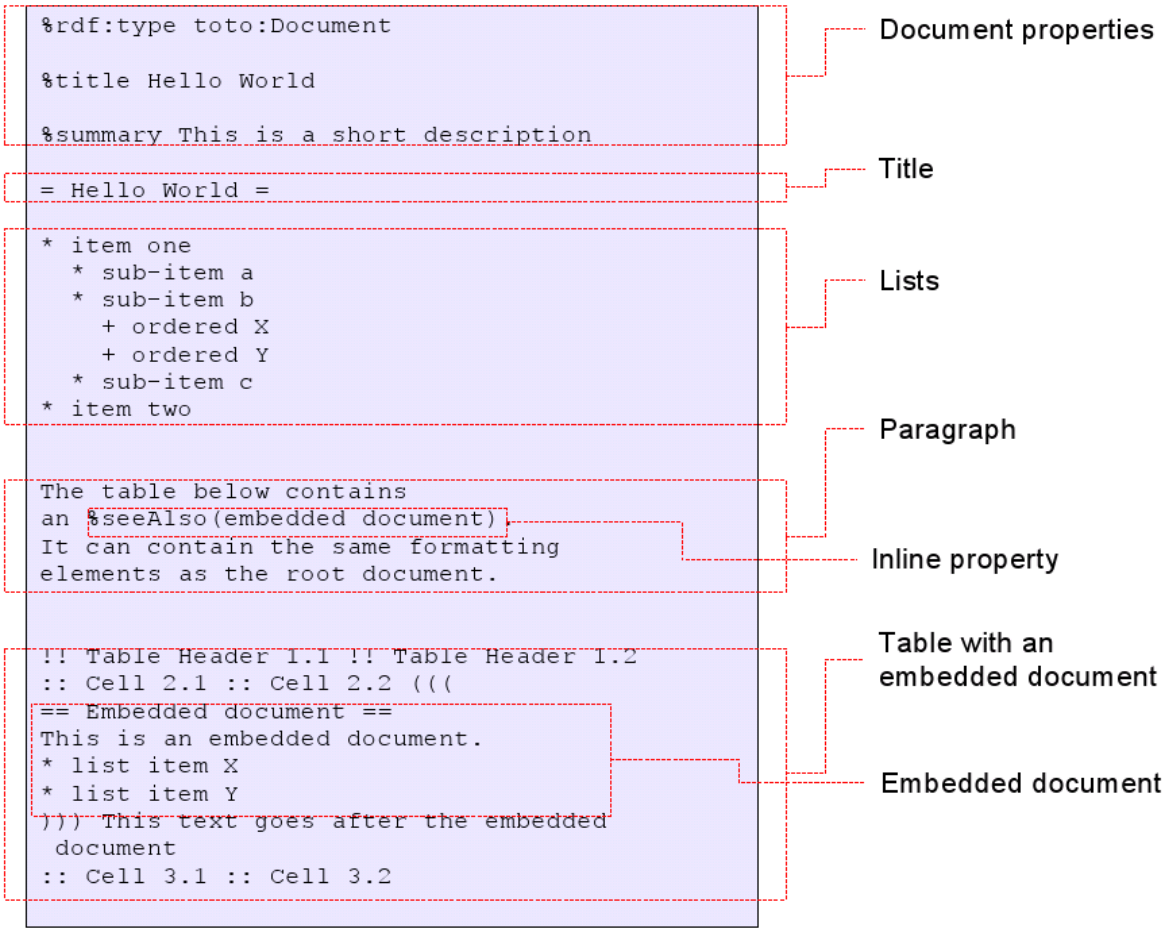

Figure 38: WikiModel Syntax Example 1

```
    <p>The table below contains<br>
       an <span class="property"
                url="seeAlso">embedded document</span>.<br>
       It can contain the same formatting<br>
             elements as the root document.
    </p>
    <table class="contentTable">
        <tbody>
            <tr><th>Table Header 1.1</th>
                <th>Table Header 1.2</th></tr>
            <tr><td>Cell 2.1</td>
                <td>Cell 2.2<div class="doc">
                        <div class="section section-level-2">
                            <h2 class="section-title">
                                Embedded document
                            </h2>
                            <div class="section-content">
                                <ul>
                                    <li>list item X</li>
                                    <li>list item Y</li>
                                </ul>
                            </div>
                        </div>
                    </div>
                    This text goes after the embedded<br>
                    document
                </td>
            </tr>
            <tr><td>Cell 3.1</td><td>Cell 3.2</td></tr>
        </tbody>
    </table>
    <p></p>
  </div>
</div>

</div></div>
```

Embedded documents The WikiModel introduces the notion of "embedded documents". The possible structure of such a document is exactly the same as the structure of the topmost one. It means that using embedded documents becomes possible to put block elements inside others blocks. This is a unique feature of the WikiModel, compared to other wiki syntaxes. Embedded documents are delimited by the following syntactical elements: "(((''...'')))". One embedded document can contain its own embedded documents and there are no limits in nesting depth. Syntax for embedded documents:

```
The top-level document content
(((
... the embedded document ...
)))
... the top-level document again
```

Semantic Properties WikiModel supports inline and block properties. The difference between these two is not important from the point of view of users. The names of properties can be chosen freely. All documents (top-level and embedded ones) can have their own properties.

Inline Property: The simplest property is an inline property, which can be used to annotate parts within a document, i. e. the value of this property may

only contain inline formatting. It is useful to annotate single word or short snippets *within a paragraph*.

Syntax example:

```
I am living in %city(Paris).
```

The syntax to create a semantic link similar to e. g. Semantic MediaWiki (Krötzsch, Vrandecic & Völkel 2006) to "Paris" is:

```
I am living in %city([Paris])
```

Block Property: Besides inline properties, WikiModel also allows *block properties*. Properties of this type takes the same place in the structure of the document as other block elements, like paragraphs, tables, list, or headers. The value of a block property is a block (hence the name), defined via:

```
%propertyName a property value
```

or even an embedded document, defined via this syntax:

```
%description (((
= Header =
This is a structured content of the property "description"
)))
```

Using semantic properties it is possible to create very complex structured documents containing at the same time semantic markup and visual formatting. A more complex example of properties follows:

```
%title A simple document

%summary A short description of this document.
It can contain *in-line formatting* and it can
span multiple lines forming one big paragraph.

%author (((
  %firstName Mikhail
  %lastName Kotelnikov
  %worksIn (((
    %type [Company]
    %name Cognium Systems
    %address (((
          ....
    )))
    %description *Cognium Systems* is a
    semantic web company...
  )))
)))

This is a simple content of the top-level document...
```

The *interpretation* of such properties depends completely on developers using the WikiModel. One possible application of this notion is the generation of XML structures inter-mixed with human-readable formatting. But the original goal of these structural elements is the introduction of a unified syntax for *semantic markups* in wiki documents. Each property can be used to declare a semantic statement about the document in which it was defined.

Note that there is no default mapping to RDF or any kind of interpretation from the semantic properties yet. This is up to the event listeners to decide.

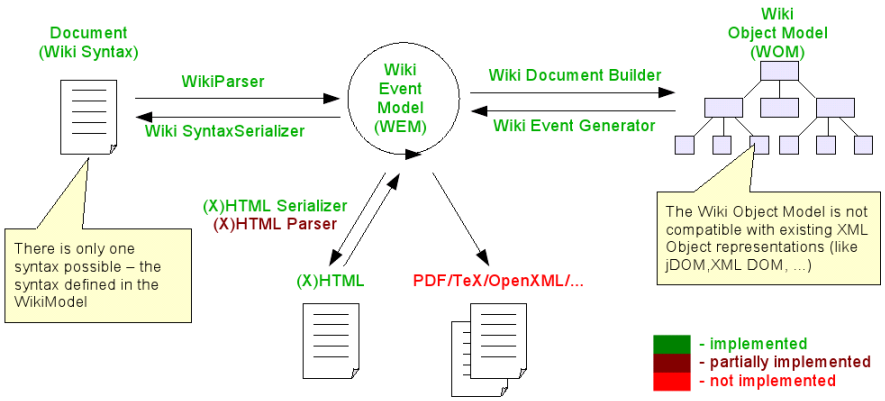

Figure 39: WikiModel Version 1

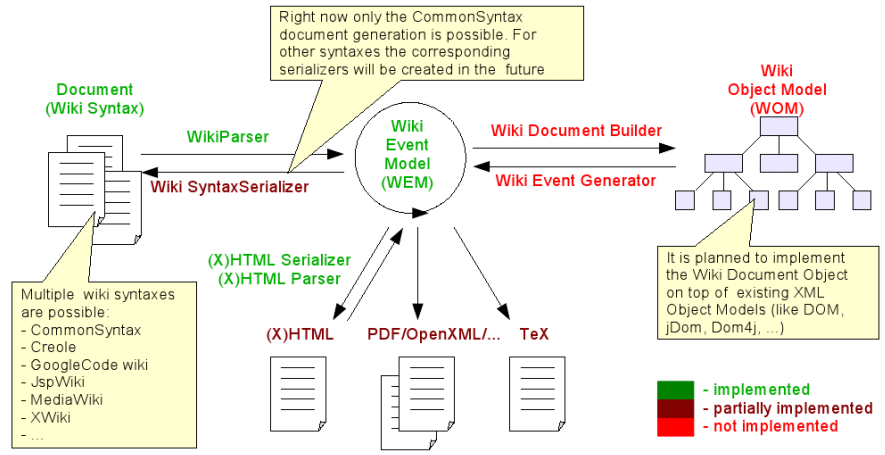

Figure 40: WikiModel Version 2

Macros Macros allow for syntax extensions, which are not handled by the parser. Similar to a verbatim block, the content is reported in its raw (unparsed) form. Different from a verbatim block, macro blocks can have names and parameters. The meaning of macro block is not defined by the WikiModel – it is up to the application developer using WikiModel to interpret the content of macro blocks and to do something specific. An example of macro syntax is:

```
{macroName param1=value1 param2='Value 2' param3="Value 3" ...}
 ... The content of the macro ...
{/macroName}
```

As an example, a user can embed HTML code within a wiki document:

```
{html a='b' c='d' comment='This is a block with HTML elements'}
<table><tr><td>
Hello, world!
</td></tr></table>
{/html}
```

Feature	WikiModel 1	WikiModel 2
Parser	JavaCC-grammar based	JavaCC-grammar based
Embedded documents	yes	yes
Semantic statements about documents	yes	yes
Syntax	only CommonSyntax	any; can be mixed
Performance	moderate	high

Figure 41: Evolution of WikiModel

6.1.3 Evolution of WikiModel

A brief overview on the evolution of the WikiModel component can be found in Fig. 41. Although more parts of WikiModel 1 (c. f. Fig. 39) are implemented, the version 2 (c. f. Fig. 40) has an improved design and other benefits. Other improvements of WikiModel 2 compared to WikiModel 1:

- Extended document schema for greater flexibility
- Extended CommonSyntax to allow expressing all WikiModel elements
- Possible to work with documents written with multiple syntaxes
- Parsers for multiple wiki syntaxes are available (JspWiki, XWiki, MediaWiki, Creole, GoogleCode wiki, ...)

6.1.4 Using WikiModel

The usage of WikiModel is exemplified in Fig. 42. First, a `WikiParser` is created. At this time, the developer decides which syntax to use. The text is read via a reader and events are routed to an event listener. Finally the parser is run over the input and delivers WEM events to the listener.

WikiModel will be used in the next iteration of HKW.

6.2 Structured Text Interchange Format (STIF)

The Structured Text Interchange Format (STIF) defines a small, manageable subset of XHTML (et al. 2002) focusing on *structural* elements.

Valid elements to be used in STIF are:

Headlines <h1>, <h2>, <h3>, <h4>, <h5>, <h6>

Block elements <p>, <pre>, and <hr>

Lists <dl>, <dd>, <dt>, <ol>, <ul>,

Tables <table>, <tr>, <th>, <td>

Inline elements , , <code> and

Images <img> with attribute `src`

Links <a> with attribute `href`

```java
package test;

import java.io.StringReader;

import org.wikimodel.wem.IWemListener;
import org.wikimodel.wem.IWikiParser;
import org.wikimodel.wem.IWikiPrinter;
import org.wikimodel.wem.PrintListener;
import org.wikimodel.wem.WikiParserException;
import org.wikimodel.wem.common.CommonWikiParser;

public class ParserTest {
    public static void main(String[] args) throws WikiParserException {
        IWikiParser parser = new CommonWikiParser();
        String text = "Hello [World]! This is a *text* to parse!";
        StringReader reader = new StringReader(text);
        IWikiPrinter printer = new IWikiPrinter() {
            public void print(String str) {
                System.out.print(str);
            }
            public void println(String str) {
                System.out.println(str);
            }
        };
        IWemListener listener = new PrintListener(printer);
        parser.parse(reader, listener);
    }
}
```

```
Available parsers:
- org.wikimodel.wem.common.CommonWikiParser
- org.wikimodel.wem.creole.CreoleWikiParser
- org.wikimodel.wem.gwiki.GWikiParser (GoogleCode wiki)
- org.wikimodel.wem.jspwiki.JspWikiParser
- org.wikimodel.wem.mediawiki.MediaWikiParser
- org.wikimodel.wem.xwiki.XWikiParser
```

Figure 42: WikiModel Syntax Example 2

All other elements and attributes may be ignored or may be processed by a STIF-conforming processor. The element nesting rules are the same as for XHTML.

A valid STIF document *may* contain an HTML header such as

```
<html>
  <head>
     <title>some title here</title>
     ... further header info here ...
  </head>
  <body>
     ...  main content goes here ...
  </body>
</html>
```

It is also valid to used STIF elements directly in plain text. A valid STIF document could look like

```
The brown fox <em>quickly</em> jumps over the fence.
```

Different kind of hyperlinks may be indicated in STIF via the use of CSS (kon Wium Lie & Bos 1999) classes.

Wiki links From a users point of view, it is relevant to distinguish wiki links which point to an entity within the same local address space and which will have a similar look and feel from external links which point to arbitrary web sites. STIF uses `class="external"` to mark external links and `class="internal"` to denote local links.

Auto-linking Some systems support automatic linking of text with pages, e. g. using NLP. From a users point of view, it is relevant to know if a link was stated explicitly by as human or has been created by heuristics. Explicit links are marked with `class="explicit"` and automatic links with `class="automatic"`. Note that CSS allows to have multiple classes on an element, hence a link can very well be e. g. `class="internal automatic"`, order of values does not matter.

Page creation A specialty of wiki links is their ability to point to non-existing pages. Often such links are rendered with a trailing question mark. If the user clicks on such a link the page is created and the user can fill it with content. STIF marks links to existing pages with the CSS class `existing` and links to non-existing pages with `nonexisting`. Note that usually only pages to other wiki pages can be non-existing.

STIF will be used to persist wiki content in the next versions of HKW.

6.3 The Wiki Syntax for HKW

HKW (c. f. Sec. 3) uses wiki syntax on two levels: structural and semantic. This section describes extensions to the WikiModel CommonSyntax. These extensions will be used in the next releases of HKW.

Structure For the structural part, which is parsed first, HKW uses the WikiModel CommonSyntax (c. f. Sec. 6.1). This allows rendering the content of *Items* in a more pleasant way. It turns hyperlinks into clickable items.

From a CDS point of view, all links are turned into items linked by *has target*.

Semantics On the semantic layer, the WikiModel syntax consists of two main ideas: Detecting NameItems and parsing structures into formal statements. Both is performed by the CDS syntax component [28].

Detecting NameItems Parsing *NameItems* is similar to named entity recognition in NLP, yet different. In NLP, there exist a number of approaches, of which most ultimately use a *gazetteer list*, i. e. a simple list of known names. The CDS syntax combines both ways to add elements to the list of known elements and detects usages of existing ones.

The following syntax snippet shows the use of link syntax to introduce new *NameItems*. Each *NameItem* is addressed via a URI. Right after their definition the parser knows it and correctly detects subsequent mentions of the same name.

```
My name is [Dirk] .
Dirk is a cool name.
```

After parsing:

```
My name is <a class="internal explicit" href="#uri">Dirk</a> .
<a class="internal automatic" href="#uri">Dirk</a> is a cool name.
```

In HKW, first the WikiModel parser turns `[Dirk]` into `<a class="internal explicit" href="Dirk">Dirk</a>` which is then further processed by an Auto-Linker component.

The Auto-Linker first tokenises the text to determine possible auto-links. This avoids linking to parts of words, e. g. linking "Jacobsson" as "[Jacob]sson". Tokenisation is performed at the characters *space*, *newline*, *tabulator*, quotation marks (`"`), apostrophe (`'`), full stop (`.`), comma (`,`), semicolon (`;`), exclamation mark (`!`), question mark (`?`), open brace (`(`), closing brace (`)`), open square bracket (`[`), and closing square bracket (`]`).

[28] `http://semweb4j.org/site/cds.syntax`

Parsing Statements After detecting all *NameItems*, the parser goes on and uses the structure and additional syntax patterns to derive semantic statements.

As syntax patterns, the parser can parse a subset of Turtle[29] syntax for triples as in the following example:

```
[Claudia] [knows] [Dirk] .
Dirk [knows] [Claudia] , [Martin].
```

The following statements are created then in the model:

```
stmt1: Claudia - knows          - Dirk
stmt2: stmt1   - has provenance - wikiPage
stmt3: Dirk    - knows          - Claudia
stmt4: stmt3   - has provenance - wikiPage
stmt5: Dirk    - knows          - Martin
stmt6: stmt5   - has provenance - wikiPage
```

The general syntax allows patterns of this form $spo.$ with ways to use multiple objects as in $spo_1, o_2, o_3.$ or multiple predicates as in $sp_1o_1; p_2o_2; p_3o_3.$. Of course, both shorthands can be combined as in $sp_1o_1, o_2, o_3; p_2o_5, o_6, \ldots p_no_m..$

Each triple is represented as a *Statement*. The parser also records provenance information so that one can keep track from which *Item* the *Statement* has been inferred. This makes correction of errors easier.

As a second option, the user can mix wiki syntax for structures and re-use these structures to yield statements. As an example, consider this wiki syntax snippet:

```
[Dirk]
* [knows]
** [Claudia]
** [Martin]
* [works at] [SAP]
* [age] : 26
```

Converted to HTML this might look like:

```
[Dirk]
<ul>
  <li>[knows]
    <ul>
      <li>[Claudia]</li>
      <li>[Martin]</li>
    </ul>
  </li>
  <li>[works at] [SAP]</li>
  <li>[age] : 26</li>
</ul>
```

As the parser operates on the generated XHTML, the structural wiki syntax to create lists is independent of the interpretation of lists.

The HTML list in our example yields the following triples in the model:

```
stmt1: Dirk  - knows          - Claudia
stmt2: stmt1 - has provenance - wikiPage
stmt3: Dirk  - knows          - Martin
stmt4: stmt3 - has provenance - wikiPage
stmt5: Dirk  - works at       - SAP
stmt6: stmt5 - has provenance - wikiPage
stmt7: Dirk  - age            - "26";
stmt8: stmt7 - has provenance - wikiPage
```

[29] http://www.ilrt.bris.ac.uk/discovery/2004/01/turtle/

Again, the provenance is recorded.

A *NameItem* used before the list is understood as a subject, list items are parsed as predicates. If a list item contains a colon (“:”), the remainder of the line is parsed as an object, more specifically as an *Item*. If the list item has no colon, but second-level list items underneath, those are parsed as objects instead.

Alternatively, if there is no *NameItem* above the list, the top-level list element itself is understood as a subject, subitems are parsed as predicates. If a second-level list item contains a colon (“:”), the remainder of the line is parsed as an object, more specifically as an *Item*. If the second-level list item has no colon, but third-level list items underneath, those are parsed as objects instead.

Fourth-level list elements are simply ignored. All other invalid structures also yield no statements.

6.4 BounceIt: Semantic Publishing

Based on the requirements identified in the Bioscience case study (workpackage 8) and confirmed in two other case studies, we have been developing a system that unifies the processes of messaging, blogging, and wiki while adding semantic aspects for better control, integration, and information retrieval. This system, called BounceIt, is based on the Semantic Pad wiki that was developed in the first year of Workpackage 1. Given that information on our desktops is rarely purely personal, BounceIt provides a much-needed smooth integration between personal and shared information. The goal is to integrate BounceIt with the CDS system in the final year of the workpackage.

BounceIt GUI is based on Web browsers. In particular, we used the *Google Widget Toolkit* (GWT) framework which is a stable and highly scalable platform for development of dynamic AJAX-based interfaces. Although initially BounceIt is designed to work off of a central server, we are also considering to use the Google Gear project to give users a possibility to work in a disconnected mode. This technology would increase the level of integration of BounceIt with the Personal Knowledge Workbench.

Functionalities implemented so far include:

- Read, write, and comment access rights via BounceIt that can be specified based semantic properties (e.g., an access can be specified as follows: ”allow person X to read and comment anything on a given topic”); open access to the general public.
- BounceIt allows users to create new accounts and groups without special administrative rights; users can be added to a group simply by giving them access to the group’s information.
- BounceIt allows wiki-based collaboration and conversations around the wiki pages, that can have independent access rights allowing various sub-groups to communicate within the same informational context.
- A Mozilla Firefox plug-in that allows to easily save an annotated hyperlink for later use or to send it to other people via BounceIt.
- Export from Wikipedia into BounceIt
- Inclusion of Google Maps into BounceIt, export and import of location information.

Below is a list of functionalities planned for the future:

- Re-integrate Text Analytics for related items detection
- Moderation of communication within a group; automatic extraction of social links from communications between users.

- BounceIt should allow users a simple way to specify the level of information urgency: notify by email (immediately, daily, weekly...), show in portal, remove; these operations apply to different information channels defined by a combination of various properties such as sender, addressee, topic, title, keyword, or type of modification (edit or comment).
- Versioning
- Integration with Nepomuk Middleware (both semantic and P2P parts) as well as with the CDS system.

Integration with CDS There are two ways how the Hypertext-based Knowledge Workbench (HKW) could be integrated with BounceIt. First, personal CDS knowledge models could be published as a network of bounces, as the items in BounceIt are called.

Second, a number of bounces could be imported to a personal knowledge model in CDS, or as a subcase of that, publicly accessible bounces could be annotated and interlinked in the personal model.

7 Summary and Outlook

In this deliverable we have introduced the concept and origins of CDS. We have presented the two parts, the SWCM data model and the CDS ontology. We described in depth the process and rationale for building the CDS core ontology of relations.

In Section 3 we presented the current prototype of our CDS editor, the *Hypertext Knowledge Workbench* (HKW), which is based on the CDS API. The usability assessment of the prototype will guide further refinement of the visual properties.

A second prototype, although in earlier stages, is the *Visual Knowledge Workbench* (VKW) which was presented in Sec. 4. We also introduced *QuiKey*, a semantic desktop visual command line.

Besides manual ways to create structures and formal statements, we also investigated less exact but more automated ways to extract structures and meaning from natural language text (Sec. 5). Such services will be integrated in CDS-based tools. We presented a text mining component from IBM, which is able to disambiguate terms or find related items, based on sophisticated analysis and NLP techniques. NLP support services include keyword extraction and speech act identification for a semantic email client. Authoring Tools included CLIE - Controlled Language for Information Extraction to create Ontologies using Controlled Natural Language and SALT, can be used to author text, semantics and structure at the same time and keep this information when producing a printable, shareable PDF document from it.

Finally we presented the ongoing use of wiki technology, most notably the WikiModel framework. We presented WikiModel in depth and also some extensions for integration in CDS tools.

7.1 Outlook

Several steps are planned for the remaining time of the project NEPOMUK. The CDS tools will be refined and integrated.

7.1.1 CDS

Currently, the RDF binding of SWCM uses its own ontology, decoupled from the NEPOMUK ontology framework. The following migration is planned in year 3 of the project:

NameItems: SWCM will use `nao:personalIdentifier`, from the NEPOMUK Annotation Ontology, to link URIs to unique strings.

Item To store the content, SWCM will use `nao:prefLabel`. Storing binary content, which is possible in SWCM, will be integrated into NEPOMUK. Most likely properties from NIE, the NEPOMUK Information Elements can be used.

We plan a tighter integration with existing desktop data sources and the CDS API. All CDS tools will profit from this integration. The following changes are planned for tighter integration with the NEPOMUK infrastructure:

- The BinStore component will be refactored into a NEPOMUK service and accessed by the NEPOMUK `swecr.core` adapter. This ensures that several instances of CDS-based tools as well as other services can all access the same set of binary content.
- The concept of data sources (depicted in Fig. 43) will allow to treat several kinds of models as dynamic (read-only) parts of the CDS data model. Via

this bridge the user will be able to browse – and annotate and link – her PIMO concepts, desktop files, and task items.

- The WikiModel component from Cognium Systems, plus our own extensions, will be integrated into the CDS API. The core data model behind WikiModel is a tree of documents (via the sub-document concept) which fits naturally into CDS notion of has detail/has context.
- NEPOMUKs NLP services will be called from the CDS implementation to help the user e g. to tag items and to find related concepts.
- To ease integration with PIMO the native RDF format of the CDS implementation will be adapted to use the NEPOMUK Annotation Ontology (NAO, [30]) and PIMO (e. g. all items will be linked to the users PIMO via

7.1.2 HKW

Based on the feedback of the expert evaluation, we started already to work on an improved HKW prototype. A design preview is depicted in Fig. 44.

To ease migration (and evaluation) of further HKW versions, we work on importing existing personal (semantic) wikis which are in use today among researchers (e. g. JSPWiki, Semantic MediaWiki) into a CDS model. The goal for year three is to turn semantic wiki users into HKW users.

7.1.3 Semantic Email and Blogging

Semantic Blogging Semantic Blogging means to attach formal metadata (e. g. RDF) to ordinary blog posts. Such metadata can be content-related - formal descriptions of what a blogger writes about - or structural - formal description about

[30] http://www.semanticdesktop.org/ontologies/nao/

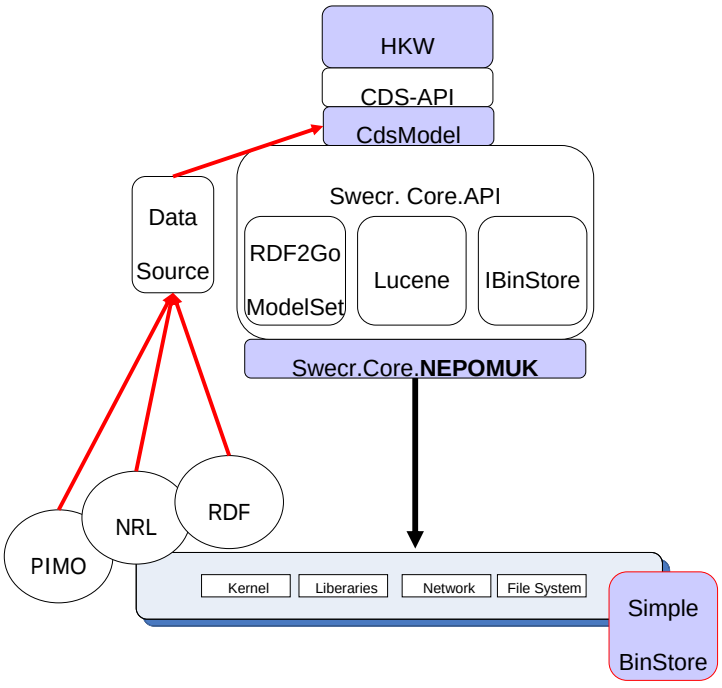

Figure 43: Planned Integration

how blogs and blog posts relate to each other. In the context of NEPOMUK, Semantic Blogging will allow users to open their personal semantic desktop environment to other, non-NEPOMUK-enabled users.

Semanta – Semantic Email Semantic Email will be able to link selective content of email messages to related data on the semantic desktop. Such data may consist of resources such as people (e.g. the communication partners in the message thread, or attendees involved in an event announcement), activities (tasks and events assigned through email) etc.

One of the largest flaws of the email communication genre is the lack of shared expectations about the form and content of the interaction. This can be attributed to the lack of explicit semantics covering the context and content of exchanged email messages. Earlier research has shown that email content can be captured by applying speech act theory. Refining and extending this research results in developing an email speech act ontology and outline a non-deterministic predictive model to support the user in deciding the best course of action upon sending or receiving an email. In the prototype, implemented as an Outlook extension, Text Analytics will help capturing content metadata whereas a thread-based approach will handle context metadata. Semantic Web technologies will invisibly embed both kinds of metadata in email, hence achieving semantically-enhanced email. The context and expectations of a message are made explicit before leaving the sender and reaching the recipient. As a result, when receiving or sending specific emails Outlook will know for example that a task or event has been created and will assist the user based on the knowledge. When a new message reaches the recipient, they will be prompted if they are expected to respond or perform some action, and so on, In this way, email communication becomes less ambiguous and more efficient. Semantic Email will be able to link selective content of email messages to related data on the semantic desktop. Such data may consist of resources such as people (e.g. the communication partners in the message thread, or attendees involved in an event announcement), activities (tasks and events assigned through email) etc. A prototype for the Semantic Email Client - *Semanta*. A beta release will occur in Month 25 in conjunction with first "Semanta" Prototype. We refer the reader to

7.1.4 Semantically Annotated LaTeX (SALT)

It should be possible to import the RDF extracted by SALT into a CDS model maybe – at higher implementation costs – also the reverse: To generate a SALT LaTeX file from a CDS model. We will explore both integration options in year 3 of NEPOMUK.

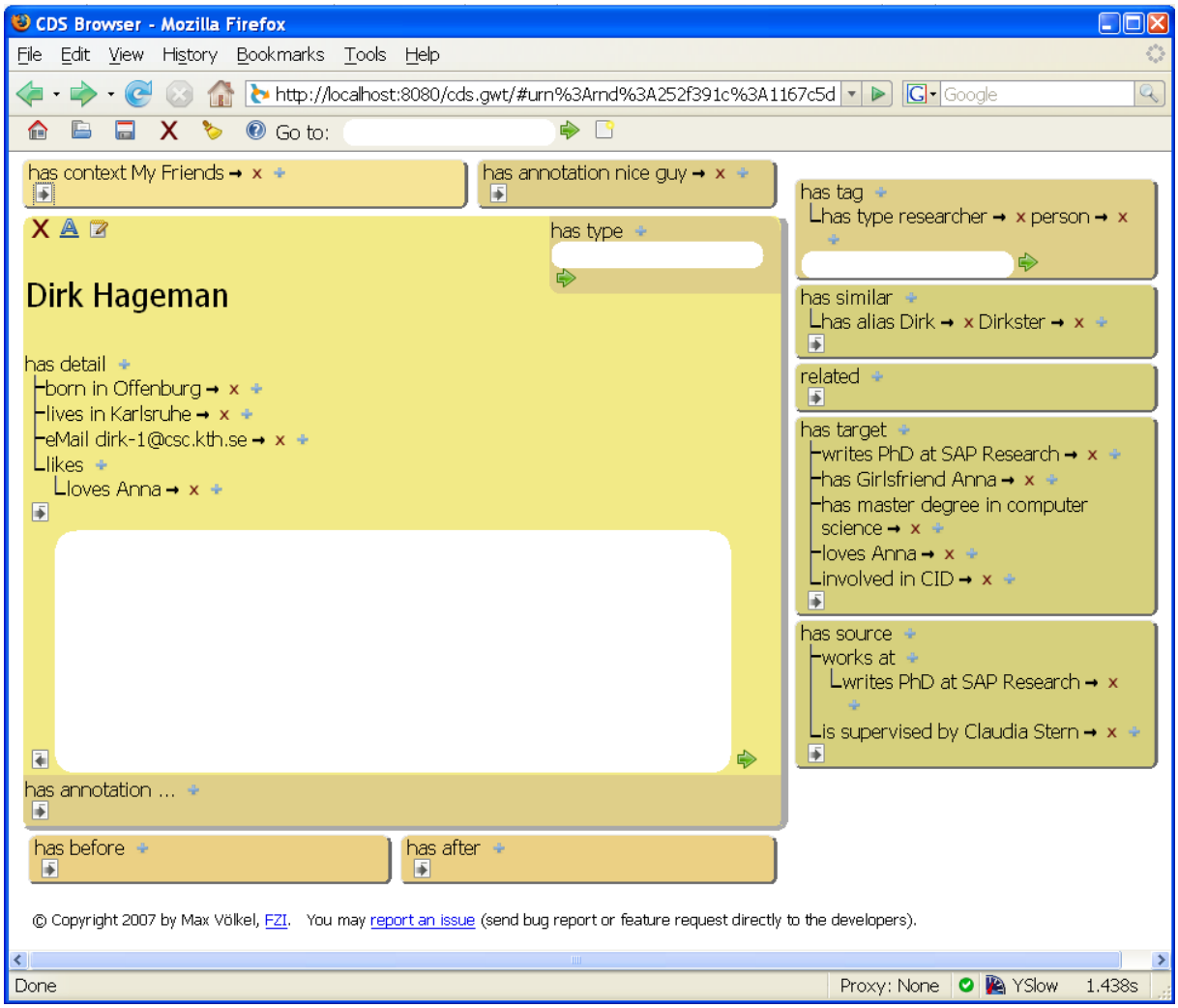

Figure 44: Design Preview of HKW

References

Adar, E., Karger, D. R. & Stein, L. A. (1999), Haystack: Per-user information environments, *in* 'CIKM', ACM, pp. 413–422.

Avery, S., Brooks, R., Brown, J., Dorsey, P. & O'Conner, M. (2001), Personal knowledge management: Framework for integration and partnerships, *in* 'Proc. of ASCUE Conf.'.

Boulos, M. N. K. (2003), 'The use of interactive graphical maps for browsing medical/health internet information resources', *International Journal of Health Geographics* **2**(1).
URL: *http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=149401*

Buzan, T. (1991), *Use Both Sides of Your Brain: New Mind-Mapping Techniques, Third Edition (Plume)*, Plume.
URL: *http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20\&path=ASIN/0452266033*

Chen, H.-H., Tsai, S.-C. & Tsai, J.-H. (2000), Mining tables from large scale html texts, *in* 'Proceedings of the 18th conference on Computational linguistics', Association for Computational Linguistics, Morristown, NJ, USA, pp. 166–172.

Cunningham, H. (1999), 'A definition and short history of language engineering', *Nat. Lang. Eng.* **5**(1), 1–16.

Cunningham, H., Maynard, D., Bontcheva, K. & Tablan, V. (2002), GATE: A framework and graphical development environment for robust NLP tools and applications, *in* 'Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics'.

Dienel, H.-L. (2006), *Technografie. Zur Mikrosoziologie der Technik*, Campus Verlag, Frankfurt/New York, chapter Schreiben, Zeichnen, Erinnern. Persönliches Wissensmanagement im Ingenieurberuf seit 1850, pp. 397–425.

Drucker, P. F. (1985), *Management: Tasks, responsibilities, practices (Harper & Row management library)*, Harper & Row.

Esselborn-Krumbiegel, H. (2002), *Von der Idee zum Text. Eine Anleitung zum wissenschaftlichen Schreiben.*, Utb. 2. Auflage.
URL: *http://www.amazon.de/gp/product/3825223345?ie=UTF8&tag=xamde01-21&linkCode=xm2&camp=1638&creativeASIN=3825223345*

et al., S. P. (2002), XHTML™1.0 the extensible hypertext markup language (second edition) – a reformulation of HTML 4 in XML 1.0, Technical report, W3C. W3C Recommendation 26 January 2000, revised 1 August 2002.
URL: *http://www.w3.org/TR/xhtml1/*

Frand, J. & Hixon, C. (1999), 'Personal knowledge management : Who, what, why, when, where, how?', Speech. working paper.
URL: *http://www.anderson.ucla.edu/faculty/jason.frand/researcher/speeches/PKM.htm*

Friedewald, M. (2000), *Der Computer als Werkzeug und Medium. Die geistigen und technischen Wurzeln des Personalcomputers*, taschenbuch edn, GNT-Verlag.

Funk, A., Tablan, V., Bontcheva, K., Cunningham, H., Davis, B. & Handschuh, S. (2007), Controlled-language information extraction for knowledge management, *in* 'Proceedings of the Sixth International Semantic Web Conference, (ISWC), Busan, Korea'.

Furnas, G. W. (1986), Generalized fisheye views., *in* 'Human Factors in Computing Systems CHI '86', pp. 16–23.
URL: *http://www.si.umich.edu/~furnas/Papers/FisheyeCHI86.pdf*

Gazdar, G. (1996), *Paradigm merger in natural language processing*, Cambridge University Press, pp. 88–109.
URL: *http://citeseer.ist.psu.edu/gazdar96paradigm.html*

Grebner, O., Ong, E., Riss, U., Brunzel, M., Bernardi, A. & Roth-Berghofer, T. (2006), Task management model, Technical Report 1.1, The NEPOMUK consortium.

Groza, T., Handschuh, S., Möller, K. & Decker, S. (2007), Salt - semantically annotated latex for scientific publications, *in* E. Franconi, M. Kifer & W. May, eds, 'ESWC', Vol. 4519 of *Lecture Notes in Computer Science*, Springer, pp. 518–532.

Groza, T., Möller, K., Handschuh, S., Trif, D. & Decker, S. (2007), Salt: Weaving the claim web, *in* K. Aberer, K.-S. Choi, N. Noy, D. Allemang, K.-I. Lee, L. J. B. Nixon, J. Golbeck, P. Mika, D. Maynard, G. Schreiber & P. CudrÃ©-Mauroux, eds, 'Proceedings of the 6th International Semantic Web Conference and 2nd Asian Semantic Web Conference (ISWC/ASWC2007), Busan, South Korea', Vol. 4825 of *LNCS*, Springer Verlag, Berlin, Heidelberg, pp. 197–210.
URL: *http://iswc2007.semanticweb.org/papers/197.pdf*

Haller, H. (2003), 'Mappingverfahren zur Wissensorganisation'. Knowledge Board Europe.
URL: *http://heikohaller.de/literatur/diplomarbeit/*

Haller, H. (2006), imapping - a graphical approach to semi-structured knowledge modelling, *in* L. Rutledge, ed., 'Proceedings of the The 3rd International Semantic Web User Interaction Workshop (SWUI2006)'. Poster and extended abstract presented at the The 3rd International Semantic Web User Interaction Workshop.
URL: *http://www.aifb.uni-karlsruhe.de/WBS/hha/papers/iMapping_SWUI2006_paper.pdf*

Hayes, P. (2004), RDF semantics, Recommendation, W3C.
URL: *http://www.w3.org/TR/rdf-mt/*

Hennum, E. (2006), Representing discourse models in rdf, *in* 'Extreme Markup Languages 2006 (R)', Montréal, Québec.

Hurst, M. (2000), The interpretation of tables in texts, PhD thesis, University of Edinburgh.

InfoVis Wiki (n.d.), availlable online.
URL: *http://www.infovis-wiki.net/index.php/Semantic_Zoom*

Jones, W. & Bruce, H. (2005), 'A report on the nsf-sponsored workshop on personal information management', report.

Jones, W., Bruce, H. & Dumais, S. (2001), Keeping found things found on the web, *in* 'CIKM '01: Proceedings of the tenth international conference on Information and knowledge management', ACM Press, New York, NY, USA, pp. 119–126.

Jones, W., Phuwanartnurak, A. J., Gill, R. & Bruce, H. (2005), Don't take my folders away!: organizing personal information to get ghings done, *in* G. C. van der Veer & C. Gale, eds, 'CHI Extended Abstracts', ACM, pp. 1505–1508.

Khosravi, H. & Wilks, Y. (1999), 'Routing email automatically by purpose not topic', *Nat. Lang. Eng.* **5**(3), 237–250.

Kinsella, S., Harth, A., Troussov, A., Sogrin, M., Judge, J., Hayes, C. & Breslin, J. G. (2007), Navigating and annotating semantically-enabled networks of people and associated objects, *in* T. Friemel, ed., 'Proceedings of Applications of Social Network Analysis'.
URL: *http://www.friemel.com/asna/*

kon Wium Lie, H. & Bos, B. (1999), Cascading style sheets, level 1, Technical Report REC-CSS1-19990111, W3C. W3C Recommendation 17 Dec 1996, revised 11 Jan 1999.

Kotelnikov, M., Polonsky, A., Kiesel, M., Völkel, M., Haller, H., Sogrin, M., Lanneröö, P. & Davis, B. (2006), Interactive semantic wikis, Technical Report 1.1, The NEPOMUK consortium.

Krötzsch, M., Vrandecic, D. & Völkel, M. (2006), Semantic mediawiki, *in* I. Cruz, S. Decker, D. Allemang, C. Preist, D. Schwabe, P. Mika, M. Uschold & L. Aroyo, eds, 'Proceedings of the 5th International Semantic Web Conference (ISWC06)', Vol. 4273 of *Lecture Notes in Computer Science*, Springer, Athens, GA, USA, pp. 935–942.

Kunz, W. & Rittel, H. W. J. (1970), Issues as elements of information systems, Technical report wp-131, University of California, Berkeley.

Marshall, C. C. (2007), *Keeping Found Things Found: The Study and Practice of Personal Information Management (Interactive Technologies) (Interactive Technologies)*, Morgan Kaufmann, chapter How People Manage Information over a Lifetime.
URL: *http://www.amazon.de/exec/obidos/ASIN/0123708664/xamde01-21*

McQuaid, H. L. & Bishop, D. (2001), An integrated method for evaluating interfaces, *in* 'CHI '01: CHI '01 extended abstracts on Human factors in computing systems', ACM, New York, NY, USA, pp. 287–288.

Mitchell, A. (2005), 'The rise of personal km', *Inside Knowledge* **9**(1).

Nelson, T. H. (1995), 'The heart of connection: hypermedia unified by transclusion', *Commun. ACM* **38**(8), 31–33.

Nielsen, J. (2005), 'Ten usability heuristics', online, retrieved 19 December 2007.
URL: *http://www.useit.com/papers/heuristic/heuristic_list.html*

North, K. (2007), Produktive wissensarbeit, *in* '5. Karlsruher Symposium für Wissensmanagement in Theorie und Prxais'. CD-ROM.

Oren, E., Völkel, M., Breslin, J. G. & Decker, S. (2006), Semantic wikis for personal knowledge management, *in* 'Database and Expert Systems Applications', Vol. 4080/2006, Springer Berlin / Heidelberg, pp. 509–518.

Pivk, A., Cimiano, P. & Sure, Y. (2005), 'From tables to frames', *Elsevier's Journal of Web Semantics: Science, Services and Agents on the World Wide Web* **3**(2-3), 132–146. Selected Papers from the International Semantic Web Conference (ISWC) 2004, Hiroshima, Japan, 07-11 November 2004.

Polanyi, M. (1958), *Personal Knowledge: Towards a Post-Critical Philosophy*, Routledge & Kegan Paul Ltd, London.

Polanyi, M. (1998), *Personal Knowledge*, Routledge.

Probst, G., Raub, S. & Romhardt, K. (2006), *Wissen Managen: Wie Unternehmen ihre wertvollste Ressource optimal nutzen*, 5 edn, Gabler Verlag.

Prud'Hommeaux, E. & Seaborne, A. (2007), 'Sparql', W3C Candidate Recommendation.
URL: *http://www.w3.org/TR/rdf-sparql-query/*

Quan, D., Huynh, D. & Karger, D. R. (2003), Haystack: A platform for authoring end user semantic web applications., *in* 'Int. Semantic Web Conf.', pp. 738–753.

Quillan, M. R. (1966), *Semantic Memory*, Bolt, Bernak, and Newman, Cambridge, MA.

Richter, J., Völkel, M. & Haller, H. (2005), Deepamehta - a semantic desktop, *in* S. Decker, J. Park, D. Quan & L. Sauermann, eds, 'Proceedings of the 1st Workshop on The Semantic Desktop. 4th International Semantic Web Conference (Galway, Ireland)', Vol. 175, CEUR-WS.
URL: *http://www.aifb.uni-karlsruhe.de/WBS/hha/papers/ISWC05-SemanticDesktopWorkshop_DeepaMehta_longpaper.pdf*

Scerri, S. (2007), Aiding the workflow of email conversations by enhancing email with semantics, *in* 'PhD Symposium, ESWC', Innsbruck, Austria.

Shneiderman, B. (1996), The eyes have it: A task by data type taxonomy for information visualizations, *in* 'VL '96: Proceedings of the 1996 IEEE Symposium on Visual Languages', IEEE Computer Society, Washington, DC, USA.
URL: *http://portal.acm.org/citation.cfm?id=832277.834354*

Shneiderman, B. & Plaisant, C. (2004), *Designing the User Interface: Strategies for Effective Human-Computer Interaction (4th Edition)*, Addison Wesley.

Tablan, V., Polajnar, T., Cunningham, H. & Bontcheva, K. (2006), User-friendly ontology authoring using a controlled language, *in* '5th Language Resources and Evaluation Conference'.

Taboada, M. & Mann, W. C. (2006), 'Rhetorical structure theory: Looking back and moving ahead', *Discourse Studies* pp. 423–459.

Völkel, M. (2007), A semantic web content model and repository, *in* 'Proceedings of the 3rd International Conference on Semantic Technologies'.
URL: *http://xam.de/2007/2007-05-voelkel-ISEMANTICS-swcm-CR.pdf*

Völkel, M. & Haller, H. (2006), Conceptual data structures (cds) – towards an ontology for semi-formal articulation of personal knowledge, *in* 'Proc. of the 14th International Conference on Conceptual Structures 2006', Aalborg University - Denmark.

Völkel, M., Haller, H. & Abecker, A. (2007), Modelling higher-level thought structures - method and tool, *in* 'Proceedings of Workshop on Foundations and Applications of the Social Semantic Desktop'.

A Appendix: CDS Ontology

```
@prefix :      <#> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .
@prefix swcm:  <http://purl.org/net/swecr#> .
@prefix foaf:  <http://xmlns.com/foaf/0.1/> .
@prefix cds:   <http://www.semanticdesktop.org/ontologies/2007/09/
@prefix xsd:   <http://www.w3.org/2001/XMLSchema#> .
@prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

cds:hasAnnotationMember
      cds:hasSuperRelation
              cds:hasTarget .

cds:isAliasOf cds:hasSubRelation cds:replaces .

cds:hasAlias cds:hasSuperRelation cds:hasSimilar .

cds:hasRelated
      cds:hasSubRelation cds:hasSimilar , cds:hasSource .

cds:hasAfter cds:hasSuperRelation cds:hasTarget .

cds:hasDetail
      cds:hasSubRelation cds:hasSubType , cds:hasSubRelation ;
      cds:hasSuperRelation
              cds:hasTarget .

cds:hasAnnotation cds:hasSubRelation cds:hasTag .

cds:hasSimilar
      cds:hasSubRelation cds:hasAlias , cds:sameAs ;
      cds:hasSuperRelation cds:hasRelated .

cds:hasTag
      cds:hasSubRelation cds:hasType ;
      cds:hasSuperRelation cds:hasAnnotation .

cds:hasTarget cds:hasSubRelation
       cds:hasAnnotationMember , cds:hasAfter , cds:hasDetail .

cds:hasSubType cds:hasSuperRelation cds:hasDetail .

cds:hasSubRelation cds:hasSuperRelation cds:hasDetail .

cds:hasSource cds:hasSuperRelation cds:hasRelated .

cds:hasType cds:hasSuperRelation cds:hasTag .

cds:sameAs cds:hasSuperRelation cds:hasSimilar .

cds:replaces cds:hasSuperRelation cds:isAliasOf .
```