
A Meta-Engineering Approach for Document-Centered Knowledge Acquisition

vorgelegt von

Jochen Reutelshöfer

Würzburg, 2014



Dissertation zur Erlangung des naturwissenschaftlichen Doktorgrades
der Bayerischen Julius-Maximilians-Universität Würzburg

**Eingereicht am 24.02.2014
bei der Fakultät für Mathematik und Informatik**

1. Gutachter: Prof. Dr. Frank Puppe

2. Gutachter: PD. Dr. Joachim Baumeister

Tag der mündlichen Prüfung: 11.07.2014

Abstract

Today knowledge base authoring for the engineering of intelligent systems is performed mainly by using tools with graphical user interfaces. An alternative human-computer interaction paradigm is the maintenance and manipulation of electronic documents, which provides several advantages with respect to the social aspects of knowledge acquisition. Until today it hardly has found any attention as a method for knowledge engineering.

This thesis provides a comprehensive discussion of document-centered knowledge acquisition with knowledge markup languages. There, electronic documents are edited by the knowledge authors and the executable knowledge base entities are captured by markup language expressions within the documents. The analysis of this approach reveals significant advantages as well as new challenges when compared to the use of traditional GUI-based tools.

Some advantages of the approach are the low barriers for domain expert participation, the simple integration of informal descriptions, and the possibility of incremental knowledge formalization. It therefore provides good conditions for building up a knowledge acquisition process based on the mixed-initiative strategy, being a flexible combination of direct and indirect knowledge acquisition. Further it turns out that document-centered knowledge acquisition with knowledge markup languages provides high potential for creating customized knowledge authoring environments, tailored to the needs of the current knowledge engineering project and its participants. The thesis derives a process model to optimally exploit this customization potential, evolving a project specific authoring environment by an agile process on the meta level. This meta-engineering process continuously refines the three aspects of the document space: The employed markup languages, the scope of the informal knowledge, and the structuring and organization of the documents. The evolution of the first aspect, the markup languages, plays a key role, implying the design of project specific markup languages that are easily understood by the knowledge authors and that are suitable to capture the required formal knowledge precisely. The goal of the meta-engineering process is to create a knowledge authoring environment, where structure and presentation of the domain knowledge comply well to the users' mental model of the domain. In that way, the approach can help to ease major issues of knowledge-based system development, such as high initial development costs and long-term maintenance problems.

In practice, the application of the meta-engineering approach for document-centered knowledge acquisition poses several technical challenges that need to be coped with by appropriate tool support. In this thesis KnowWE, an extensible document-centered knowledge acquisition environment is presented. The system is designed to support the technical tasks implied by the meta-engineering approach, as for instance design and implementation of new markup languages, content refactoring, and authoring support. It is used to evaluate the approach in several real-world case-studies from different domains, such as medicine or engineering for instance.

We end the thesis by a summary and point out further interesting research questions considering the document-centered knowledge acquisition approach.

Acknowledgements

It has been more than six years now that I started working on a rather vague topic that evolved towards what is written down in this thesis. I was very lucky to have many nice people around me during that time, who supported me in various ways.

First of all I want to thank my supervisors. Frank Puppe had an open door and an open mind for all my ideas and needs at any time. Special thanks go to Joachim Baumeister for the close mentoring before, during, and after my time as a scientific staff member at the Department of Applied Informatics and Artificial Intelligence. He introduced me to all aspects of practical research in the field of knowledge engineering. His friendly but firm critics on my daily work as well as his inspiring input were like a constant boost for me and my studies on the topic. Thanks also go to the other doctoral students within the knowledge engineering group, Reinhard Hatko and Martina Freiberg, as well as to Peter Klügl for the great time in our office room that we shared for more than four years. I would also like to thank Florian Lemmerich, Marianus Iland, Alexander Hörnlein, and all the other members of the department for the fruitful discussions and the great working atmosphere.

Further, I need to thank my new employer denkbare GmbH. Not only did I receive a warm welcome in a great team and helpful comments on my work. I was given also the extraordinary opportunity to finish my work smoothly, by providing me extra time to complete this thesis.

Last but not least I have to thank my lovely wife Stephanie. She always encouraged me during the hard times and showed understanding during the busy times.

Without the support of all those persons, this work would not have come to this point.

Foreword

Knowledge Acquisition is still the bottleneck of developing knowledge-based systems. While machine learning approaches require large amounts of well documented cases, which are often not available, the key to success is to reduce the effort as far as possible. A long term goal of our research is therefore enabling domain experts to develop knowledge bases largely by themselves without much help of a knowledge engineer. This approach requires adequate tools being tailored to the task, easy to learn and to use for developing and testing knowledge bases. The dissertation of Jochen Reutelshöfer presents a new solution based on a "meta engineering approach", which allows continuous adaptations of the knowledge acquisition tool used by the domain experts. The key idea is to support the following process: Domain experts develop knowledge bases in a distributed manner editing just documents with a mixture of informal and formal knowledge, e.g. semantic wikis. The semantic is defined by special markups within the documents. The markups are parsed in the background and allow immediate testing of the knowledge base, either interactively or automatically with test cases defined in another markup. If a domain expert feels, that a different formalization of existing markups or new markups would speed the development process, the meta engineering approach allows to add new markups easily in a systematic manner. In that way, experts get tailored tools adapted to the domain and also to their individual styles and preferences. Since the wiki approach is based on text documents, it supports a continuous knowledge formalization process, where the domain expert is able to copy or enter informal and partly formalized knowledge and transform it later - or to add special knowledge transformation markups for knowledge formalization of partly structured data. Several case studies in research and industrial knowledge engineering projects demonstrate the usefulness of the approach. By increasing the flexibility of the tool support for domain experts formalizing knowledge, the work of Jochen Reutelshöfer is an important contribution to widen the knowledge acquisition bottleneck.

Prof. Dr. Frank Puppe

Contents

Abstract	5
Acknowledgements	7
Foreword	9
1 Introduction	17
1.1 Research in Knowledge Acquisition	18
1.1.1 An Overview of Knowledge Acquisition History	19
1.1.2 Progress in Knowledge Acquisition Research	21
1.2 Knowledge in a Bottle	23
1.2.1 The Knowledge Acquisition Bottleneck	23
1.2.2 The Competency Dilemma in Knowledge Engineering	24
1.2.2.1 Direct vs. Indirect Knowledge Acquisition	25
1.2.2.2 A Mixed-Initiative Approach by Active Participation	26
1.2.3 Knowledge Acquisition in a Cognitive Environment for Social Processes	28
1.3 About Knowledge	31
1.3.1 Systems Levels	31
1.3.2 The Knowledge Level	32
1.3.3 Knowledge Acquisition in the Knowledge Level Perspective	34
1.4 Contribution of this Work	35
1.4.1 Goals	35
1.4.1.1 The Goals of the Social Process	36
1.4.1.2 The Knowledge Acquisition Process	36
1.4.1.3 Purpose of this Work	37
1.4.2 Scope	38
1.5 Structure of this Work	39
2 Approaches for Knowledge Base Authoring	41
2.1 Form-based Authoring	41
2.2 Graph-based Authoring	43
2.3 Table-based Authoring	45
2.4 Authoring by Domain Specific Languages	46
2.5 Document-based Authoring	48
2.6 Approach of this Work	49

3	Document-Centered Knowledge Acquisition	51
3.1	DCKA in a Nutshell	51
3.1.1	Documents for Knowledge Base Development	52
3.1.2	Knowledge Markup Languages	54
3.1.3	Multimodal Knowledge	55
3.1.4	The Document Space	58
3.1.4.1	The Document Space as a Graph	58
3.1.4.2	The Document Space and Human Mental Models	58
3.1.5	The Collaborative Social Process of DCKA	62
3.1.5.1	Preconditions	62
3.1.5.2	Mixed-Initiative KA by Incremental Formalization	64
3.1.5.3	Mutual Exchange of Expertise	65
3.1.6	Authoring of Multimodal Knowledge	68
3.1.6.1	A Human-Computer Interaction Model of Interactive Alignment	69
3.1.6.2	The Levels of Knowledge Communication	69
3.1.6.3	Problems in Markup-based Knowledge Authoring	70
3.1.6.4	Support for Knowledge Authoring	71
3.2	The Advantages and Challenges of DCKA	73
3.2.1	Advantages	73
3.2.1.1	Low Barriers for Basic Contributions	73
3.2.1.2	Incremental Formalization	73
3.2.1.3	Freedom of Structuring	74
3.2.1.4	Example-based Authoring	74
3.2.1.5	Quality Management	75
3.2.2	Challenges	75
3.2.2.1	Authoring Assistance	75
3.2.2.2	Content Refactoring	76
3.2.2.3	Navigation and Search	77
3.2.2.4	Redundancy Detection	78
3.2.2.5	Debugging	78
3.2.3	Requirements for a Document-Centered Authoring Environment	79
3.3	Semantic Wikis and DCKA	80
3.3.1	Semantic Wikis	80
3.3.2	Wikis and DCKA	80
3.3.3	Semantic Wikis and Knowledge Engineering	81
3.4	Markup-based Knowledge Acquisition Tools	82
3.5	Different Application Scenarios for DCKA	83
3.5.1	Ontologies in RDFS/OWL	83
3.5.1.1	Markup for RDFS:	84
3.5.1.2	Markup for OWL	86
3.5.2	Diagnostic Problem-Solving Knowledge with d3web	86
3.5.2.1	Markups	88
3.5.2.2	Knowledge Organization	89
3.5.3	Knowledge for Exploratory Data Analysis	90

3.5.4	Training Cases for e-Learning with CaseTrain	92
3.6	CommonKADS	93
3.6.1	A Brief Overview of CommonKADS	93
3.6.1.1	The CommonKADS Model Suite	93
3.6.1.2	The CommonKADS Process Model	94
3.6.1.3	The CommonKADS Role Model	94
3.6.2	CommonKADS and DCKA	95
3.6.2.1	Comparison	95
3.6.2.2	Combining DCKA and CommonKADS	95
4	A Meta-Engineering Approach for DCKA	97
4.1	An Overview of Knowledge Acquisition Tool Customization	98
4.2	Design Time and Use Time: The Systems Design Dilemma	99
4.2.1	Agile Software Development	100
4.2.2	Meta-Design	100
4.3	Flexibility and Coordination in DCKA	101
4.3.1	Flexibility in Document-centered Knowledge Acquisition	101
4.3.2	The Document-Centered Knowledge Acquisition Architecture	102
4.4	The Meta-Engineering Process for DCKA	103
4.4.1	Exploration	104
4.4.2	Design	105
4.4.2.1	Markup Design Principles	106
4.4.2.2	Markup Aspects	111
4.4.3	Implementation	114
4.4.3.1	System Level	114
4.4.3.2	Content Level	115
4.4.4	Knowledge Acquisition	115
4.4.5	Conclusion	116
4.4.5.1	Language Complexity vs. Model Complexity	117
4.4.5.2	Meta-Design: GUI-based Tools vs. DCKA	118
4.5	Extending Semantic Wikis	118
4.5.1	Dimensions of Semantic Wiki Extensions	119
4.5.2	Decorating Semantic Wikis	121
4.5.3	Challenges towards an Extensible Semantic Wiki	122
5	Techniques for the Implementation of DCKA	123
5.1	Comparison with Software Engineering	124
5.2	Overview: Techniques Presented	126
5.3	Parsing of Multimodal Knowledge	126
5.3.1	The KDOM Data-Structure	126
5.3.2	A Top-Down Parsing Algorithm	127
5.3.2.1	Description	128
5.3.2.2	Performance	129
5.3.2.3	Example	130

Contents

- 5.3.2.4 Discussion 133
- 5.3.3 Extension 1: Incremental Top-Down Parsing 134
 - 5.3.3.1 Description 135
 - 5.3.3.2 Performance 136
 - 5.3.3.3 Discussion 136
- 5.3.4 Extension 2: Cardinality Constraints 137
 - 5.3.4.1 Description 137
 - 5.3.4.2 Performance 138
 - 5.3.4.3 Discussion 138
- 5.3.5 Extension 3: Backtracking for Top-Down Parsing 139
 - 5.3.5.1 Description 139
 - 5.3.5.2 Performance 141
 - 5.3.5.3 Discussion 141
- 5.3.6 Implementation Architecture 141
- 5.3.7 Tutorial: Implementing Markups as KDOM Schemas 143
 - 5.3.7.1 Introduction by Examples 143
 - 5.3.7.2 Recursive KDOM Schemas 147
 - 5.3.7.3 Limits of KDOM Schema parsing 150
- 5.4 Terminology Resolution and Knowledge Generation 153
 - 5.4.1 Reference Resolution in Closed-World Authoring 154
 - 5.4.2 An Abstract Model for Knowledge Bases 155
 - 5.4.3 A Formal Model for Knowledge Authoring 156
 - 5.4.4 Closed-World Authoring Reconsidered 157
 - 5.4.4.1 Strict Object Definition 157
 - 5.4.4.2 Complex Object Definitions 158
 - 5.4.5 The Knowledge Compilation Task Summarized 158
 - 5.4.6 The Resource Delta 159
 - 5.4.7 The Genericity of the Incremental Knowledge Base Update Task 159
 - 5.4.8 An Incremental Knowledge Base Update Algorithm 161
 - 5.4.8.1 Description 161
 - 5.4.8.2 Termination 161
 - 5.4.8.3 Efficiency 163
 - 5.4.8.4 Proof of correctness 163
 - 5.4.9 Discussion 165
- 5.5 A Meta-Model for the Declarative Implementation of Markups 166
 - 5.5.1 The Levels of Multimodal Knowledge Compilation Revisited 166
 - 5.5.2 The Knowledge Markup Description Language 169
 - 5.5.3 Semantics 169
 - 5.5.4 Example 171
 - 5.5.5 Meta-Level Authoring Support 173
 - 5.5.6 Discussion 175

6	An Authoring Environment for DCKA	177
6.1	KnowWE: An Overview	177
6.1.1	History	177
6.1.2	Architecture	177
6.2	Knowledge Acquisition with KnowWE	178
6.2.1	Manual Knowledge Testing and Use	178
6.2.2	Automated Testing by Continuous Integration	181
6.2.3	Debugging	183
6.2.3.1	The Rule Debugger	183
6.2.3.2	The Rule Markup Rendering Component	184
6.2.4	Refactoring	185
6.2.5	Authoring Support	186
6.2.5.1	Instant Editing	186
6.2.5.2	Table Editing	186
6.2.5.3	Code Completion	187
6.2.5.4	Drag & Drop	187
6.2.5.5	Term Overview	189
7	Case Studies	191
7.1	ESAT: Assisting Technologies for Handicapped Persons	191
7.1.1	Introduction	191
7.1.2	Application Scenario	191
7.1.3	Knowledge Base Structure	192
7.1.4	The Meta-Engineering Process	192
7.1.4.1	Knowledge Acquisition Architecture	192
7.1.4.2	Markups	193
7.1.5	Discussion	198
7.1.6	System Use	198
7.2	WISSASS: Medical Knowledge about Cataract Surgery	200
7.2.1	Introduction	200
7.2.2	Application Scenario	200
7.2.3	Knowledge Base Structure	201
7.2.4	Knowledge Acquisition Process	201
7.2.4.1	Seeding of an Initial Knowledge Base	201
7.2.4.2	The Meta-Engineering Process	203
7.2.4.3	Outlook	207
7.2.5	System Use	208
7.3	Managing Chemical Safety with KnowSEC	209
7.3.1	Introduction	209
7.3.2	Application Scenario	209
7.3.3	Knowledge Base Structure	209
7.3.4	The Meta-Engineering Process	210
7.3.5	System Use	212

Contents

- 7.4 Maintenance Knowledge for Special Purpose Machines 213
 - 7.4.1 Introduction 213
 - 7.4.2 Application Scenario 213
 - 7.4.3 Knowledge Base Structure 213
 - 7.4.4 Knowledge Acquisition 213
 - 7.4.5 System Use 215
- 7.5 HermesWiki: E-Learning in Ancient Greek History 216
 - 7.5.1 Introduction 216
 - 7.5.2 Knowledge Acquisition Architecture 216
 - 7.5.3 The Meta-Engineering Process 218
 - 7.5.4 System Use 219
 - 7.5.4.1 Generated Geographic CV 219
 - 7.5.4.2 Automated Quiz Sessions 219
- 8 Conclusion 221**
 - 8.1 Summary 221
 - 8.1.1 Introduction 221
 - 8.1.2 Approaches for Knowledge Base Authoring 222
 - 8.1.3 Document-Centered Knowledge Acquisition 222
 - 8.1.4 A Meta-Engineering Approach for DCKA 223
 - 8.1.5 Techniques for the Implementation of DCKA 224
 - 8.1.6 KnowWE - An Authoring Environment for DCKA 224
 - 8.1.7 Case Studies 225
 - 8.2 Outlook 225
 - 8.2.1 A Catalogue of Markup Design Guidelines 225
 - 8.2.2 Combination with Heavy-weight Knowledge Acquisition Approaches 226
 - 8.2.3 Formal Definition of the KAA 226
 - 8.2.4 User Interface and Deployment 227
 - 8.2.5 Learning Material 227
 - 8.3 Discussion 228
- Abbreviations 229**
- Bibliography 231**

1 Introduction

The idea of human built more or less intelligent behaving beings has been fascinating people for ages and is subject of myths and tales of all cultures. For example, ancient Greek mythology tells that Hephaestus was creating bronze robots to help him in his workshop or to defend the island. In this kind of ancient myths the life spirit of the artificial creatures was rooted in god powers or magic. It was only a few centuries ago, that stories began describing the creation of autonomous artificial beings also as a result of scientific practice. As the first real scientific attempts in modern times, related to artificial intelligence, one can consider the development of formal logics and conceptualizations by mathematicians and philosophers beginning in the late 19th century, as for instance Gottlob Frege's *Begriffsschrift* [Fre79]. In those ideas one of the most important aspects of artificial intelligence is rooted, which we today call *knowledge representation*. The actual term *artificial intelligence* was not coined until the time when the first digital computers had come up, providing concrete means for intelligent systems to calculate decisions. After artificial intelligence was established as a research discipline in 1956¹, its pathway was characterized by both, magnificent achievements and severe setbacks. Also in the sub-domain of knowledge engineering early successes have been celebrated. Hayes-Roth predicted a high impact of this technology in the early eighties.

"Over time, the knowledge engineering field will have an impact on all areas of human activity where knowledge provides the power for solving important problems. We can foresee two beneficial effects. The first and most obvious will be the development of knowledge systems that replicate and autonomously apply human expertise. For these systems, knowledge engineering will provide the technology for converting human knowledge into industrial power. The second benefit may be less obvious. As an inevitable side effect, knowledge engineering will catalyze a global effort to collect, codify, exchange and exploit applicable forms of human knowledge. In this way, knowledge engineering will accelerate the development, clarification, and expansion of human knowledge."

Hayes-Roth et al., 1983, [HRWL83]

Today, about 30 years later, knowledge engineering indeed is applied in industrial context to create intelligent systems, corresponding to the first effect predicted by Hayes-Roth. Considering the second category, collection and exchange of human knowledge has been brought to a new level due to emergence of the world wide web. However, codification and exploitation of applicable knowledge at global scope, which is also the aim of the so-called *semantic web* [BLHL01], is in a state of current research. Hence, it is probably exaggerated to state

¹Darhmouth Conference, summer 1956 at Darhmouth College

1 Introduction

that knowledge engineering by today has an impact on all areas of human activity in a sense as intended by Hayes-Roth. Indeed, he was wise enough not to bind this forecast to a concrete point in time, when this vision will have become true. Consciously or unconsciously, he avoided that fault famously made by Allen Newell in 1957 saying: "*Within ten years a digital computer will be the world's chess champion.*" The first defeat of a world chess champion by a computer was in 1997, achieved by Deep Blue [CJhH02]. It took until about 2006 until it was commonly agreed, that the computer chess programs clearly out-range even the strongest human players. Hence, since very recent years we can say that Newells statement has been fulfilled, while having taken more than 40 years instead of ten. This and other examples from the history of artificial intelligence teach us that the way from initial (very) promising experiments and results to the actual break-through in practice of an AI technology often holds much more difficulties than envisioned even by the most experienced researchers of the field. This progress in most cases is characterized by minor gradual improvements over time rather than particular breakthrough events. That pattern of progress also applies to many notable achievements in AI of recent years, such as the development of autonomously driving cars in the DARPA autonomous vehicle challenge [TMD⁺06] or the Jeopardy solver Watson [FBCC⁺10] by IBM. These kinds of achievements indicate that it is very often not the one key idea or invention leading to success, but continuous improvements according to experiences made in practical application. Considering the discipline of knowledge-based system development, which focuses on the creation of intelligent systems for particular problem domains, it appears that this pattern also holds. Despite all the progress made in that field within the last decades, the challenge of knowledge acquisition, which aims to accumulate the knowledge of the problem domain in a computer-interpretable form, is not yet entirely solved. The data-driven approaches for populating knowledge bases automatically, using machine learning and/or natural language processing technologies, made significant progress during the last years. However the natural language processing problem is still hard and due to a lack of quality or quantity of the required data, many knowledge acquisition endeavours still need to be addressed by classical knowledge engineering techniques.

The pathway, that still lies in front of us before reaching the state envisioned by Hayes-Roth, comprises several challenges, such as the reduction of development costs, the cognitive and social demands of development, the risks, and the long term maintenance problem. More progress on these aspects is required to make knowledge-based systems the first choice solution of all reoccurring knowledge intense tasks of the modern world. Proceeding a few more steps on this long way is the intention of this work. Before going into detail, a retrospective summary of the part of the way, that already has been accomplished, is presented in the following section.

1.1 Research in Knowledge Acquisition

In this section, a short overview of the last decades of knowledge acquisition research, including some aspects of the related domain of software engineering, is provided.

1.1.1 An Overview of Knowledge Acquisition History

The early days of artificial intelligence in the sixties and early seventies of the last century were strongly focused on the development of general problem-solving algorithms. After first advances on exemplary problems, the application in real-world problems uncovered the fact that the encoding of the necessary domain knowledge forms an essential task itself. While inference algorithms in general can be reused in various applications, the process of knowledge formalization showed to be necessary and tedious for each other subject domain anew. This issue, showing to be demanding and strongly recurring, evolved into a problem of critical importance. It cannot be considered by solely employing methods of computation and the rules of logics, but inherently involve human factors, as the most important sources of knowledge are humans. After some successful expert systems applications had come up in the seventies and early eighties, most prominently MYCIN [BS84], showing the potential of this technology to be promising, high expectations arised. These early expert systems, called *first generation expert systems*, were created in an ad-hoc manner. The way those systems were created even had been formulated to be a kind of art [LAY87]. While a provisional collection of methods were available, the choice and application of those was more or less up to the intuition of the developer, while being strongly determinant for the success. Hence, the research goal of the community stated then was to develop a well-founded engineering discipline providing clear process models and decision guidelines to rule out risks and make the development costs predictable. In this context the term *knowledge engineering* was coined. One turned away from the prototyping approaches [BKMZ84, BKKZ92] of the first generation expert systems that had been predominant for the development of knowledge systems until that time. Progress was anticipated by the introduction of development methodologies with well-structured processes, guiding the activities including for instance verbose project planning and quality management efforts.

In the eighties knowledge acquisition itself developed into a grown-up research discipline within artificial intelligence. A large research community has formed, organizing workshops and conferences on the topic regularly all over the world, most noticeable the KAW² workshop/conference series, which started in Banff, Alberta 1986. Already then, the metaphorical term of the *knowledge acquisition bottleneck* had been coined by Feigenbaum [Fei77] and others. It postulates that the difficult and tedious acquisition of the required knowledge basically prevents the technology of intelligent systems to be extensively employed in a wide range of knowledge intensive domains (e.g, industrial manufacturing, medical treatment).

Today, after about 35 years of research about widening the knowledge acquisition bottleneck many valuable insights have been gained. Intelligent systems have been built and applied successfully in numerous practical applications [HBM⁺04, MNTMQ96]. Nevertheless, the decades of ongoing research efforts also indicate the wickedness of the problem, as the costs of the required knowledge acquisition in most cases stayed considerable high with project risks still being prevalent. This fact questions the economic profitability of the application of an intelligent system for many potential use cases. Thus, while general feasibility of intelligent system solutions has been proven on numerous case studies in various domains, one has to admit that the knowledge acquisition bottleneck still has not been solved in principle, when considering the economic point of view. In the meantime, methods for automated reasoning however have

²<http://www.k-cap.org/k-cap/about-k-cap.html>

1 Introduction

strongly evolved, becoming standard techniques with the computational power still having increased by orders of magnitude. Considering this development, the relevance of a knowledge acquisition bottleneck may be considered more obvious than ever.

For the establishment of a full-fledged engineering discipline, researchers were able to find inspiration by the related discipline of general software engineering. There, the situation was similar in the time of the early attempts to create software artefacts. Already in the sixties computer scientists have become aware that software development in many cases needed much more resources (budget and time) than expected and had much lower quality than intended. This phenomenon also is known under the term *software crisis* enduring for multiple decades. The term *software engineering* has been coined in context of the efforts to face the software crisis by introducing improved process models and quality management mechanisms. Considering these parallels, knowledge acquisition researchers could examine and adapt the experiences made in software engineering on introduction of structured development processes. A whole family of software development process models arose from the idea of the waterfall model. It was first mentioned by the United States Navy Mathematical Computing Advisory Panel [PoNR56] in 1956, being republished by Herbert Benington [Ben83] in 1983, and proposes a process model running through several different phases. Each of these phases produces a specification of the entire project captured by a set of specification documents, which have to comply to well-defined standards. These specifications start from a very abstract level in the first phase and stepwise leads to concise and formal specifications, with the last specification being transformed to programming code in a (more or less) straight forward way. Important knowledge engineering methodologies developed in the nineties, e.g. *KADS* [SWB93] and the refined version *CommonKADS* [SAA⁺01], have some similarities when compared to the predominant software engineering methodologies of that time. CommonKADS aims to formalize the knowledge as different kinds of models in distinct subsequent phases, each delivering a complete specification at some level of abstraction resulting at a computer-interpretable knowledge base after the implementation phase, conducted last. These 'heavy-weight' methodologies, driving the entire project specification through multiple phases, have been discussed controversially, especially in the software engineering community [Hig01], for many years. One of the major pro arguments is, that problems discovered at an early phase are much easier to solve than when being discovered at a late phase, motivating the careful production of comprehensive specification documents and their validation at any phase. Important counter arguments are the lack of flexibility to change significant parts at a late phase. This often becomes necessary due to either changes in the requirements specification or design errors that have still not been discovered by then. One of the earliest proclamations of this problem is given by McCracken et al. in 1982 [MJ82]:

"System requirements cannot ever be stated fully in advance, not even in principle, because the user doesn't know them in advance—not even in principle. To assert otherwise is to ignore the fact that the development process itself changes the user's perceptions of what is possible, increases his or her insights into the applications environment, and indeed often changes that environment itself. We suggest an analogy with the Heisenberg Uncertainty Principle: any system development activity inevitable changes the environment out of which the need for the system arose."

McCracken and Jackson [MJ82]

These lines basically claim that it is rather impossible that a deep requirements specification at project beginning can lead to a product that then satisfies the customer as he is significantly influenced by the development process.

This problem is addressed by another family of software engineering methodologies, the so-called *agile* methodologies which emerged at the beginning of the new millennium mainly out of open source software development practices. The first representative of this family of 'light-weight' approaches was *extreme programming* proposed by Kent Beck [Bec00], being followed by others as for example *Scrum* [SB01], which is very popular today also in commercial software development. The general idea is to hunt for quick wins in a feature-by-feature approach, being suitable especially for small and medium sized projects. Therefore, the entire project specification is usually formulated as a set of small features. These features are put into a priority order and then implemented accordingly. The sustainability of the implemented features is asserted by ongoing extensive automated testing. These process models are strongly inspired by the way that early open source software was developed in the late nineties, showing a workflow completely contrary when compared to the waterfall model but still in many cases produced high quality software. As the idea of agile development became more and more successful and popular in software development, it is no surprise that it was adopted by knowledge engineering researchers. In the early years of the millennium several methodologies for agile development of knowledge-based systems have been developed, e.g., by Baumeister [Bau04] and Knublauch [Knu02]. While not providing an entire new methodology, the approach presented in this work is strongly related to the principles of agile knowledge engineering.

1.1.2 Progress in Knowledge Acquisition Research

"Most computer scientists, however, mistakenly view the engineering of intelligent systems as a solved problem."

Mark Musen [Mus13]

In this section we provide a brief discussion of the progresses made in knowledge acquisition research during the past 20 years. Shatbolt stated the most pressing issues in 1991 as follows:

1. *Knowledge acquisition is difficult.*
2. *We have only the beginnings of a KA methodology.*
3. *There is little reusability of our knowledge in expert systems.*
4. *Few integrated knowledge acquisition environments exist.*
5. *There is little synergy between acquisition techniques.*
6. *It is difficult to translate between the results of different KA tools.*
7. *It is hard to integrate the results of different acquisition sessions.*
8. *It is difficult to verify and validate the results of acquisition.*
9. *The acquisition techniques are sometimes inappropriately applied.*
10. *Experts and their cognitive processes are poorly understood.*

Shatbolt [Sha91]

1 Introduction

Two decades have passed since these propositions have been stated. They provide a suitable basis to discuss the progress within the field of knowledge acquisition. Shatbold described the KADS methodology [SWB93] as the most comprehensive methodology existing. It has been consequently been evolved and completed forming the CommonKADS methodology available as comprehensive handbook [SAA⁺01]. Another methodology focused on engineering knowledge is MOKA [SC01]. More recently, agile methodologies have emerged [Bau04, Knu02], also focusing the distributed development [VPST05]. Therefore considering the second proposition, substantial progress has been made. The question of the a priori decision for an agile or a traditional methodology however has not been discussed as much as in software engineering [Hig01]. The advances in software engineering practices also had influences on the challenges of knowledge acquisition. The invention of better tools (e.g., IDEs) and methods (e.g., object-oriented programming) made the creation of knowledge acquisition tools less costly and risky. Technologies like XML³ (including for example XSLT⁴) made translation between different KA tools (6) easier, at least at a syntactical level. As for all computer science areas, the advent of the web had a major impact also on knowledge acquisition research and practice. Component-based software engineering, using and sharing open-source libraries on the web, facilitated tool development in particular in the academic context. With collaborative and distributed knowledge acquisition over the web [SG96] flexibility significantly improved. The research efforts in the context of the semantic web [BLHL01], envisioning an all-encompassing knowledge system on the web, while not yet showing satisfying success, brought many important achievements to knowledge acquisition. The standardization of particular knowledge representation languages considering syntax and semantics might be considered as one of the greatest achievements of the semantic web research efforts (c.f., RDF⁵, RDFS⁶ and OWL⁷). Tools and libraries for the management of data in these standardized knowledge representation languages followed, forming a notable progress on knowledge reusability (3), synergy (5), translation between tools (6) and knowledge integration (7). While this is surely an achievement, one has to admit that these languages are only suitable for a narrow range of knowledge-based tasks. Hence, the standardization of further knowledge representation languages is strongly desired.

The verification and validation of knowledge bases (8) was subject of intense research since that time [VC99, BS06, NL05, Bau11, BR11]. For most common knowledge representation languages verification methods can be looked up from the literature and applied in a straight forward way. That might even be considered as the aspect where the most clear advances have been made.

The problem of understanding the cognitive processes of the experts (10) still have only received comparatively little attention [FMPS10, ESA05, WSG92].

Proposition (1) is obviously an issue that cannot be changed. Possibly the intention of the author was to propose that this fact needs to be accepted. That even more indicates the necessity to work on the other issues to empower people to cope with that difficult task.

Today, knowledge-based systems are applied successfully in many domains. But despite all

³<http://www.w3.org/XML/>

⁴<http://www.w3.org/TR/xslt>

⁵<http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/>

⁶<http://www.w3.org/TR/rdf-schema/>

⁷<http://www.w3.org/TR/owl2-overview/>

the progress made, it has not yet become a real standard technology. Beside all the success stories, that can be found in literature, many other projects fell victim to the knowledge acquisition bottleneck or the maintenance trap (while hardly being published in that case). Therefore, the technology is often still perceived as immature and risky [SS01]. The methods being invented and experiences being made within the last decades in knowledge acquisition research are available in papers and books. Nevertheless, the knowledge even more needs to become integral part of the common education of computer scientists and IT business engineers. Another reason for many potential use cases for knowledge-based systems not being addressed is, that they are not recognized as such. The border between knowledge-intense and traditional information processing tasks is fluent. Therefore, the distinction and corresponding decision making is not easy and requires notable experiences with knowledge-intense information systems. Furthermore, socio-technical problems often not being clearly obvious in (exemplary) research projects but emerging in large real-world applications hinder a more wide-spread employment. This is where today's research on knowledge-based systems needs to set in.

1.2 Knowledge in a Bottle

"A little semantics goes a long way."

James Hendler

1.2.1 The Knowledge Acquisition Bottleneck

For about twenty-five years, researchers of knowledge acquisition have been confronted the knowledge acquisition bottleneck. No matter what methods or knowledge sources are employed, the knowledge acquisition bottleneck always seems to emerge at some point on the pathway between the knowledge sources and the intelligent system. It leads to the following general problems for the wide-spread practical application of knowledge-based system solutions:

- **High General Development Costs:** In practice, a cost-benefit analysis is the basis for deciding for or against a knowledge-based system solution for a given problem. Today, the development costs are still quite high, leading to a negative result of the cost-benefit analysis in many potential application scenarios. Therefore, reduction of the average development costs will increase wide-spread application of knowledge-based systems technologies.
- **Long Term Maintenance Problems:** Keeping a productive knowledge-based system alive demands maintenance activities as usually the requirements and the domain change over time. As the need for these maintenance updates typically occurs after longer time periods, the developers have to familiarize with the knowledge base again. Even worse, often the initial developers and experts are not available any more. For this reason, successfully introduced systems can fail after years of productive use due to strong dependency on particular persons.

- **Risks and Uncertainty:** Accurate estimation of development costs a priori is hard. This problem has also been experienced in software engineering. Estimates always have uncertainties causing unexpected deviations of the development costs. In rare cases unforeseen problems even cause the project to be rather unfeasible.

Automated knowledge acquisition, for instance from books or other documents, can be considered the holy grail of knowledge acquisition. Much research efforts have been put into natural language processing within the last years (e.g., [MS99, Mit03]). However, automated extraction of expressive semantic relations from natural language text is still hard and not (always) suitable for knowledge acquisition for knowledge-based systems development. Therefore, manual acquisition and modeling of knowledge is still common practice in many application scenarios. In that case, persons being experienced in the application domain, usually called domain specialists or experts, play a fundamental role as knowledge source. In practice, the development of knowledge-based systems usually requires collaboration of a heterogeneous group, involving (at least) domain specialists and knowledge engineers. It is commonly agreed that the knowledge acquisition bottleneck more often is rooted in the implied social process than in the technology [DSW⁺00]. Therefore, in this work, we also focus on the social aspects and cognitive challenges of this kind of collaboration in a knowledge intensive domain.

1.2.2 The Competency Dilemma in Knowledge Engineering

The development of knowledge systems by the help of domain specialists inevitably requires collaboration of domain specialists and the knowledge-based system developers, i.e., knowledge engineers. This usually brings together two groups of persons with very different backgrounds. While the experts are experienced with the application domain, the knowledge engineers are computer scientists, being skilled at the methods of knowledge-based system development. The two areas of competency can be considered as orthogonal dimensions, as depicted in Figure 1.1. The *competency dilemma* of knowledge engineering emerges from the ignorance of both groups in the other dimension respectively, typically being prevalent at the beginning of a knowledge engineering project. It is one major root of the knowledge engineering bottleneck.

The first barrier caused by this dilemma is the problem of miscommunication [Mus89a]. This problem, however, can be overcome by some efforts, for example by the formation of a *systematic domain* [WF86]. A systematic domain is an explicitly defined terminology of unequivocal, agreed-on terms of the domain. The establishment of this kind of shared vocabulary however, can be considered as a first knowledge acquisition step itself.

The next barrier being raised by the competency dilemma emerges at the task of knowledge modeling, i.e., the knowledge being formalized in a structure, which is suitable to support the process of automated inference. While knowledge engineers are very competent at this task, they lack the necessary domain knowledge to be modeled. While the domain specialists have the domain competency, they however are usually rather clueless in knowledge modeling—naturally. The question, who should how perform the knowledge modeling task, was and is the research question, which needs to be solved to ease the knowledge acquisition bottleneck for classical knowledge system development. Many different approaches addressing that question

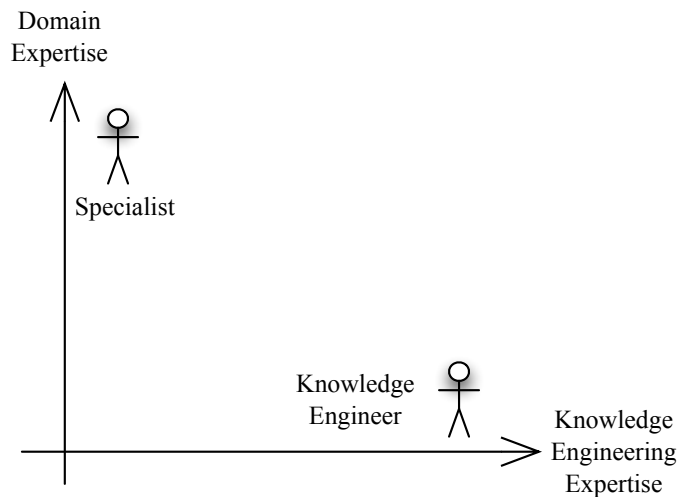


Figure 1.1: The two orthogonal dimensions of expertise in knowledge engineering.

have been proposed and explored until today. They can be categorized in two major strategies, *direct* knowledge acquisition and *indirect* knowledge acquisition.

1.2.2.1 Direct vs. Indirect Knowledge Acquisition

The two strategies that are distinguished in knowledge engineering—direct knowledge acquisition and indirect knowledge acquisition—are discussed in detail in the following sections.

Indirect Knowledge Acquisition In *indirect knowledge acquisition* (indirect KA), knowledge engineers obtain the domain knowledge from the domain specialists and then implement it in a computer-interpretable form into the knowledge base [SAA⁺01, BS84]. In that way, the domain specialists are not too strongly involved in the 'unfamiliar' knowledge modeling task. This method, in general leading to high quality knowledge modeling, however implies for every piece of knowledge a two-step process. That is, for each modification or extension at least two persons have to be available, raising considerable high development costs and organizational problems. This also imposes large challenges for the long term maintenance of the system. While this method is on the one hand comfortable for the domain specialists, not being involved at the technical level, this also can cause problems at a later stage of the system life cycle, as they are not familiar with the knowledge base structure and its possibilities for adaptation or extension. In indirect knowledge acquisition the competency dilemma basically is bridged by the knowledge engineers more or less becoming (low level) experts of the domain, as they have to collect and transform all the required domain knowledge.

Direct Knowledge Acquisition The idea of *direct knowledge acquisition* (direct KA) aims for active involvement of domain specialists within the knowledge formalization task [KBD⁺89,

1 Introduction

Mul90, Bau04, Mus89a, PG92]. In direct KA the domain experts are creating (at least significant parts of) the knowledge base autonomously. This method proposes high flexibility and cost efficiency. In particular, a domain specialist in principle is able to extend a knowledge base (in the ideal case) on his own. This is especially valuable at a later stage of the product life cycle, when requirements for minor adaptations emerge. However, it demands from the domain specialists to perform the task of knowledge formalization, which usually turns out to be difficult and unfamiliar for them. Therefore, in direct knowledge acquisition the design of appropriate knowledge acquisition tools is important [Mus89a], to ease this task for the domain experts. In contrast to the indirect knowledge acquisition approach, here the competency dilemma is bridged to some extent by domain specialists acquiring knowledge engineering skills.

Both methods have been discussed and explored extensively and showing significant strengths and weaknesses respectively. In the following, possibilities for mixing up the two strategies are discussed.

1.2.2.2 A Mixed-Initiative Approach by Active Participation

We claim that one conclusion that can be drawn from the experiences in direct and indirect knowledge acquisition is, that in most cases neither the one nor the other way in its pure form is optimal. Direct and indirect knowledge acquisition can be considered as the extremes of a scale, providing a large space for intermediate solutions. One cannot expect that after the decades of research and experiences in knowledge acquisition, the knowledge acquisition bottleneck can be resolved abruptly. However, an intermediate solution, also taking advantage of the socio-technical conditions given today, can provide support for the social processes of knowledge engineering to significantly ease the knowledge acquisition bottleneck.

Today's Socio-Technical Conditions The possibilities for flexible collaborative computer-supported work are better than ever before as domain specialists usually are acquainted to the use of computers and the internet. While about 15 years ago, the web was used by a narrow group of technology affine people, the situation significantly has changed. Also people without professional technical background are used to work with digital devices, ubiquitous interconnectivity on the web provided. Collaboration of distributed groups over the web, using synchronous or asynchronous communication, is a common practice. The emergence of the Web 2.0 technologies probably had a stronger influence on our everyday life than any other technology within the last 15 years. The use of computers and the web for social interaction concerns all aspects of our lives, leading to a notable prevalence of basic expertise in human-computer interaction. This provides large potential for direct participation of a wide range of persons in a collaborative workflow of knowledge acquisition. It also facilitates long term maintenance of systems, as a person can easily and quickly connect and login, also if his working environment has changed over time. Building up on existing technologies, special conditions, that are suitable for supporting a collaborative knowledge acquisition workflow, can be established easily.

Active Participation A discussion of *active participation* of domain specialists in knowledge engineering is given by Schilstra and Spronck [SS01]. Active participation is related to di-

rect knowledge acquisition. While direct knowledge acquisition however focuses on the actual knowledge formalization task, being carried out by experts autonomously, the focus of active participation is differently. The primary goal there is to involve the experts into a collaborative social process of development. Beside of the formalization of knowledge, this comprises a wide range of different activities. While autonomous knowledge formalization should be enabled technically, this is not required nor intended initially. For being empowered for active participation a person needs to be enabled to:

- Comfortably access, browse, and test the knowledge base autonomously
- Trace the changes being made by others
- Comment on the content in a simple way
- Communicate to other participants
- Modify the content ('enabled' in a technical sense)

Typically, the development of a knowledge base comprises various kinds of tasks, ranging from simple to highly complex ones. However, telling another participant when having recognized a (potential) flaw is considered as a (minimal demanding) act of active participation, which nevertheless can be very valuable. Beside solely notifying, inquiring more experienced participants about the detected issue to extend the own competency in a social process of learning is possible. Hence, the lower the socio-technical barriers for the prerequisites discussed above are set, the easier it is to empower more people for active participation. Informally spoken, the strategy of active participation is to 'bring on board' as many of the project participants as possible, in particular also those with non-knowledge engineering backgrounds. This is very much in contrast to indirect knowledge acquisition, where the experts play a rather passive role, usually being interviewed by the knowledge engineers [WR95]. Unlike in direct knowledge acquisition, active participation does not determine who performs the knowledge formalization tasks, but rather describes a socio-technical state of empowerment. It does however not presume knowledge formalization skills, nor any kind of predefined task assignments.

Mixed-Initiative Knowledge Acquisition We introduce the concept of *mixed-initiative knowledge acquisition* (MIKA), being an intermediate solution between direct and indirect knowledge acquisition. There, the formalization process is driven in a collaborative incremental process involving alternating initiative of various participants with different expertise. The development of a knowledge base is not a monotonic process where knowledge base entities are created one by one until the knowledge base is completed. In addition to the task of creating new knowledge base content, this process implies a wide range of other tasks, which are essential for a successful result. This includes for example documentation, refactoring, testing/creation of test cases, and proof reading. These tasks again require different kinds of expertise considering both dimensions of competency. Some tasks like documentation or proof reading require less knowledge engineering expertise than others, e.g. refactoring. Hence, the assignment of these different tasks to the participants needs to consider these demands in expertise. The MIKA approach proposes that:

1 Introduction

- Multiple participants with different expertise are empowered for active participation on the knowledge acquisition process.
- Any task is performed by a participant with suitable expertise considering the task.
- If no single participant is capable of the task, then a suitable collaboration of multiple participants is initiated dynamically.

Hence, mixed initiative knowledge acquisition not only proposes dynamic assignment of different tasks to participants with different backgrounds. Ideally, it also should support the decomposition of a task into subtasks, being addressed by different participants alternately in cooperation.

1.2.3 Knowledge Acquisition in a Cognitive Environment for Social Processes

Learning plays fundamental role in knowledge acquisition [SS01], as a means for bridging the competency dilemma. As we discussed in Section 1.2.2.1, no matter whether direct or indirect knowledge acquisition is approached, competency development in the one or other dimension is necessary. The compromise approach of mixed-initiative knowledge acquisition does not change that issue in general. However, it can provide some flexibility to resolve the competency dilemma, exploiting existing capabilities of the participants as good as possible. Nevertheless, the conditions for learning in a socio-technical context are essentially important for successful knowledge acquisition. Provide the conditions to allow the exchange of expertise as shown in Figure 1.2 has to be considered as one important goal itself. To understand the required processes of learning, understanding, and communicating, Gaines proposes to refer to the corresponding expertise in the domain of cognitive psychology [WSG92]. This process is conducted within a cognitive environment as a platform. Gaines claims that the understanding of this cognitive environment will provide a basis for the development of knowledge engineering tools and methods that support the cognitive processes. One also needs to incorporate in the project setting that even if a suitable cognitive environment is given, a process of learning requires a considerable amount of time and resources. Hence, a knowledge engineering project cannot barely be considered as the creation of a digital artifact, but to a significant extent is a social process. Forming the conditions for this social process is an important aspect of the knowledge acquisition approach discussed in the following.

In the previous sections, mixed-initiative knowledge acquisition and many of its facets have been introduced. Now, we present a coherent view on all these facets and discuss their causal relations in context. It serves as a guideline to derive a concrete approach supporting mixed-initiative knowledge acquisition. Figure 1.3 outlines the structure of the mixed-initiative knowledge acquisition approach. Each level is based on the underlying one in a sense as being supported or enabled by it.

As this flexible approach of mixed-initiative knowledge acquisition allows every participant to contribute according to his expertise, the overhead of communication and learning is limited. Therefore, it has high potential for reducing overall development costs. Domain specialists are empowered by active participation and trained by the social process and therefore able to some extent to contribute directly to the knowledge base, which simplifies long term maintenance.

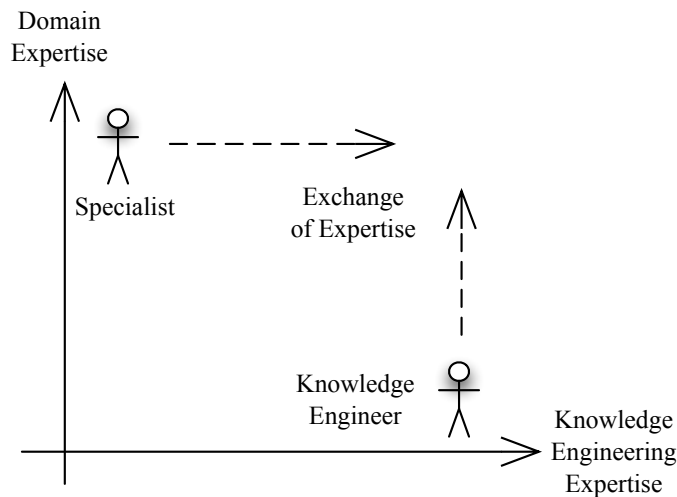


Figure 1.2: Intended exchange of expertise along the two dimensions of expertise.

The mixed-initiative knowledge acquisition approach heavily relies on specific socio-technical conditions, which are essential for the effective cooperation of knowledge engineers and domain specialists. For the task of knowledge formalization a knowledge acquisition tool, custom-tailored to the needs of the domain experts, should be used. Closely related to the tool is the structure of the knowledge base content. A content structure, which is intuitively understandable by the domain specialists. i.e., allows for simple alignment to their mental models, needs to be established. A well tailored tool and an intuitive content structure strongly eases the knowledge formalization task for the domain specialists [Mus89a]. On the one hand, the design and creation of such kind of tools needs to be performed by the knowledge engineers, requiring deep understanding of the domain context. On the other hand, the domain specialists need to obtain *basic* knowledge engineering skills to contribute actively to the knowledge acquisition process, even with a custom-tailored tool. At this point, the competency dilemma discussed in Section 1.2.2 needs to be addressed. To make the domain specialists familiar with basic knowledge engineering aspects on the one hand and on the other to provide the knowledge engineers with the required information about the domain, a social process of learning is conducted. It aims to exchange competencies along the dimensions domain expertise and knowledge engineering skills.

This social process relies on multiple preconditions. First of all, a cognitive environment for learning is required [WSG92], providing an interaction space for communication, studying, and resource sharing. Further, it is important to lower the technical barriers as far as possible for the participation of the domain specialists in the social process and the overall knowledge acquisition process. The knowledge engineers need to familiarize themselves with the domain, which usually happens in cooperate sessions with the experts. For that purpose, additional domain descriptions, e.g., illustrating documents, should be provided as lookup resources.

The stack depicted in Figure 1.3 shows the causal dependencies from bottom up, not rep-

1 Introduction

representing any temporal dimension. However, two levels are representing processes, the social process and the knowledge acquisition process. The knowledge acquisition process relies on the socio-technical conditions, which can be considered as the output of the social process. The better the socio-technical conditions become, the more efficient the knowledge acquisition process will be. Hence, the two processes are running in parallel, the outcome of the social process improving the knowledge acquisition. Naturally, at the beginning of a project the focus is on the social process, while shifting to the knowledge acquisition process over time. The social process can be considered as a process of continuous improvement of the knowledge acquisition process, never to be stopped entirely.

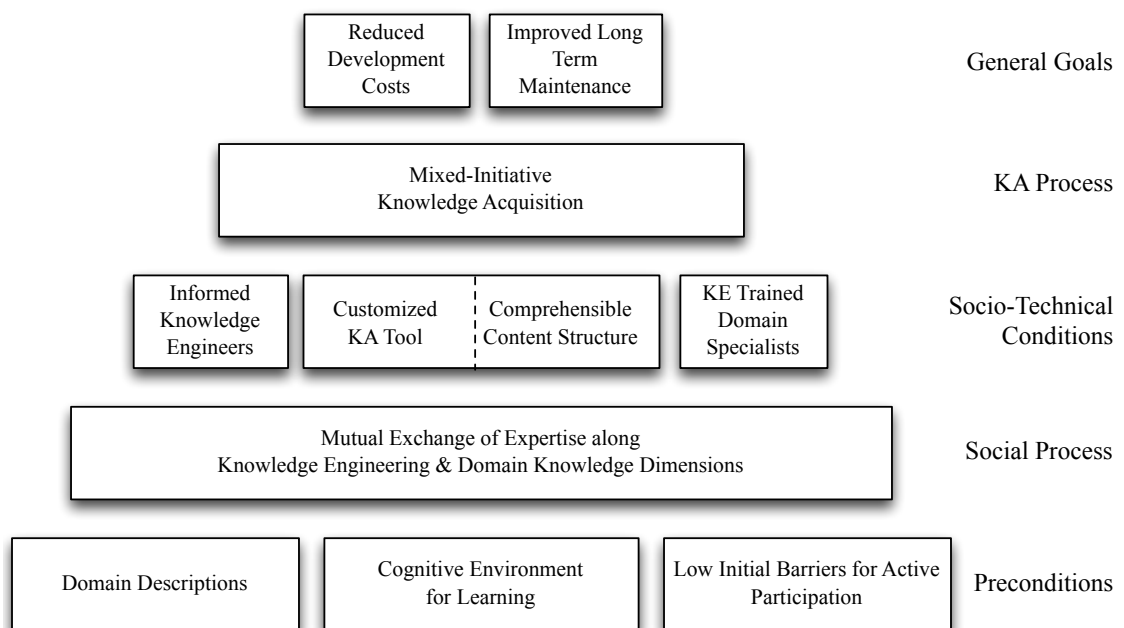


Figure 1.3: The structure of the mixed-initiative knowledge acquisition approach.

The next question is, how this stack can be supported in practice, in particular considering the preconditions at the bottom. We have developed the *document-centered knowledge acquisition* approach, which is well suited to conduct a knowledge engineering project according to this schema. It is the major topic of this work.

Before discussing the contribution of this work in more detail, the meaning of term 'knowledge' for this work is defined, to obtain a more clear notion of what needs to be 'squeezed' through the bottleneck.

1.3 About Knowledge

The notion of 'knowledge', while being rather important—not only in knowledge engineering but in AI in general—is often used in an informal way. For clarity, in this work the term is used in a sense complying to Newell's [New82] definition of the nature of 'knowledge'. He was one of the first to postulate a comprehensive and consistent definition of the term, which also has been adopted by many AI researchers. It will be used throughout this work as it is—despite of its radical abstractness—very suitable to demonstrate the purpose and scope of this approach. In the following, Newell's notion of knowledge is briefly explained. Elegance and justification of his definition of the term come from its derivation from the abstraction layer stack of (common) computer systems, being explained in the following.

1.3.1 Systems Levels

The systems levels stack describes the different technological layers of abstraction of a digital system. Depicted in Figure 1.4, it starts at the bottom with the device level ranging up until the program level. Each level can be characterized by a medium, components, composition laws,

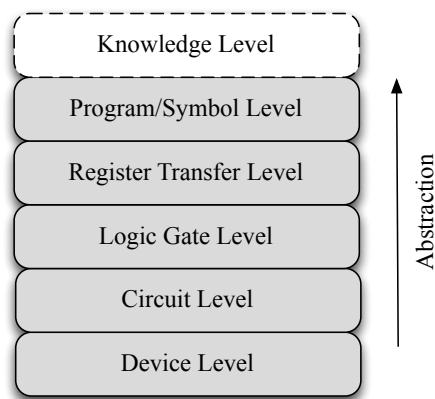


Figure 1.4: The layer stack of abstraction levels of computer systems with the knowledge level.

behavior laws, and a system. The components of the *device level* at the bottom for example are atoms of different matter typically involving conducting, semi-conducting, and non-conducting material. Further, the medium of this level is electrons, the behavioral laws are given by the laws of electrodynamics. The composition laws correspond to the capabilities of technical manufacturing and the system might be described as solid body. All these characteristics can be defined for each level and provide a perspective on the overall apparatus at the respective level of abstraction. Climbing up the levels of abstractions of the stack, the medium evolves to voltage at the *circuit level*, to bits at the *logic gate level*, and to bit arrays at the *register transfer level*. Above that level the *program level* resides, dealing with variables and data types. There, the components might be subroutines, which can be composed. For the case of AI systems Newell

proposed the notion of *symbol level* instead of *program level* as these systems usually are built on physical symbol manipulation mechanisms. In a physical symbol system the knowledge is represented in some kind of data structures containing physical symbols provided with a computation mechanism drawing the inference by manipulating these symbols [New80]. These data structure might be theories of formal logics, production (or any other sort of) rules, frames, or any other kinds of knowledge representation each including an inference mechanism responsible for the symbol manipulation. Hence, the symbol level describes a special family of architectures of the *program level* that is typically present in intelligent systems. All these levels are pretty obvious, well-known, and understood. Each level can be constructed from the level below. Further, it is notable—and extremely fortunate for systems design—that one can successfully design a system at a certain level despite of a remarkable lack of understanding of the levels below.

1.3.2 The Knowledge Level

Newell [New82] proposes the existence of another level on top of the symbol level—the *knowledge level* (c.f. Figure 1.4). The characteristics of the knowledge level are as follows: The system at the knowledge level is an agent. As commonly agreed, the characteristics of an agent are that it has a body that is situated in an environment, which he can perceive and manipulate by actions to achieve his goals as shown in Figure 1.5. Bodies, goals, and actions are forming

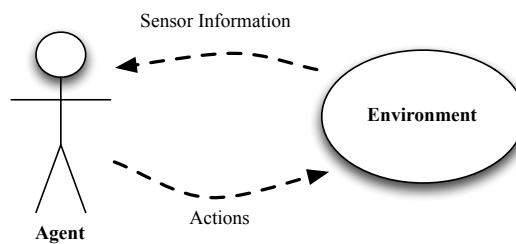


Figure 1.5: An agent perceives sensor information from the environment and can manipulate it by actions (according to Russel et. al. [RN03]).

the components of this level. The behavioral laws are defined by the principle of rationality as we expect an agent to act, i.e., selects actions, rational according to his goal preferences. From this characterization of the knowledge level as an (intelligent) system level, the definition of the notion 'knowledge' emerges: *Knowledge* is the medium at the knowledge level that connects goals and actions to produce a rational behavior of the agent. This definition of knowledge however does not specify any structure or organization of these connections between the goals and actions.

"Knowledge will stay forever abstract and will never be actually at hand."

Allen Newell [New82]

In fact, this abstractness is one of the key points of Newell's definition. The actual structure and inference is only determined at symbol level. Hence, we can not actually build a system

by only specifying it on the knowledge level. This radical incompleteness characterizes the knowledge level [New82]. However, the sense and value of this definition of knowledge, which might appear somehow shallow at first glance, becomes more obvious when considering an agent from an external perspective. Observing an agent in his environment, the symbol level is hidden. The agent can still be analyzed at the knowledge level. If the observer knows about the goal and knowledge of the agent, he can predict the actions of the agent according to the principle of rational behavior, even though it is completely hidden which kind of symbol level representation or inference mechanism the agent is using. The knowledge level abstracts from computer systems to agents. Due to the fact that on each level the details of the underlying level can be ignored, on the knowledge level also other kinds of agents, i.e. non-computer systems, as for example humans can be discussed.

The knowledge level perspective can be illustrated by a simple example of daily life. One (human) agent *A* drops his apartment keys while walking through the city. Another (observer) agent *B* perceives the human agent *A* having lost his keys. The event of the loss of the keys is a piece of knowledge. Those agents, who are perceiving (and understanding) the loss, possess this knowledge. Depending on whether agent *A* realized the loss himself, agent *B* can predict his action according to the principle of rationality. That is, if agent *A* realizes, he will pick up the keys, probably instantly. This prediction, however, requires for Agent *B* to have some more background knowledge about the goals of *A*, which probably are in conflict with being locked out from home, and about whether *A* has the knowledge to infer this, but works in principle. Despite not knowing what kind of symbol level representation the fact of the loss is transformed to during perception—and in fact we do not know exactly how a human symbol level representation works (if any)—, we can discuss and even to some extent predict the effect of the piece of knowledge about the loss. This is true even if agent *B* is non-human having a completely different symbol level representation and inference mechanism compared to agent *A*. This indicates that the knowledge has been translated to two different symbol representations during perception but leading to the same conclusions using different inference mechanisms. The example illustrates, that knowledge can be discussed at a level abstracting from the symbol level while still allowing predictability of the system behavior.

Knowledge obviously is something abstract that can in some way be perceived from the environment. Either symbol expression inside an agent (only) *represents* a piece of knowledge. Hence, the relation of the notions of *knowledge* and *knowledge representation* is described [New82] by this informal equation:

$$\textit{Representation} = \textit{Knowledge} + \textit{Access} \quad (1.1)$$

This equation indicates that agents need *access* to (abstract) knowledge to convert it into a specific (symbol level) representation for making use of it. This access in most cases is a rather difficult task. While perception is a prerequisite there is still a complex process from sensor data to a reasonable representation of the knowledge, often requiring considerable background knowledge already existing. In a computer science perspective, after perception of sensor data a syntactical analysis is done, followed by a semantic interpretation. Performing actions, that purposely are modifying the environment in a way enabling other agents to access specific knowledge, can be considered as an act of agent communication.

The knowledge level and its distinction from the symbol level is helpful for the discussion of the knowledge acquisition approach presented in this work.

1.3.3 Knowledge Acquisition in the Knowledge Level Perspective

Newell proposed the existence of a new systems level—the knowledge level—providing a new perspective on the notion of knowledge and the analysis of intelligent systems [New82]. Being focused on how the knowledge level allows to predict agents’ actions according to the principle of rationality considering the prevalent knowledge and goals, little is said about the transfer of knowledge towards an agent. However, a consistent perspective of knowledge acquisition and transfer can be derived easily from what Newell declared about the knowledge level. By doing so, the meaning and interrelations of the terms knowledge and (symbol level) representation, including the transition from one to the other, will be set clear for the remainder of this work.

For the derivation of a knowledge level-oriented perspective on knowledge acquisition, the key concepts knowledge, knowledge representation, and situated agents need to be aligned to the traditional view of knowledge-based systems development. There a knowledge base is created by humans manually by using some kind of knowledge acquisition tool. Applying the knowledge level perspective, this involves two (categories of) agents, the human and the knowledge base. The human user of a knowledge acquisition tool can be considered as an agent. The knowledge base itself in the first place is bare of sensors and actuators. However, assuming the knowledge acquisition tool forming the shared environment of the human and the knowledge-based system agent, sensors and actuators can be defined for knowledge exchange. The agent representing the knowledge base can perceive knowledge entered into the knowledge acquisition tool by a human. Further, it can tell its knowledge or derivations when demanded by the user. This agent-based view on the use of knowledge acquisition tools is depicted in Figure 1.6. Each of the two agents, according to Newell, has its specific internal (symbol level) knowledge

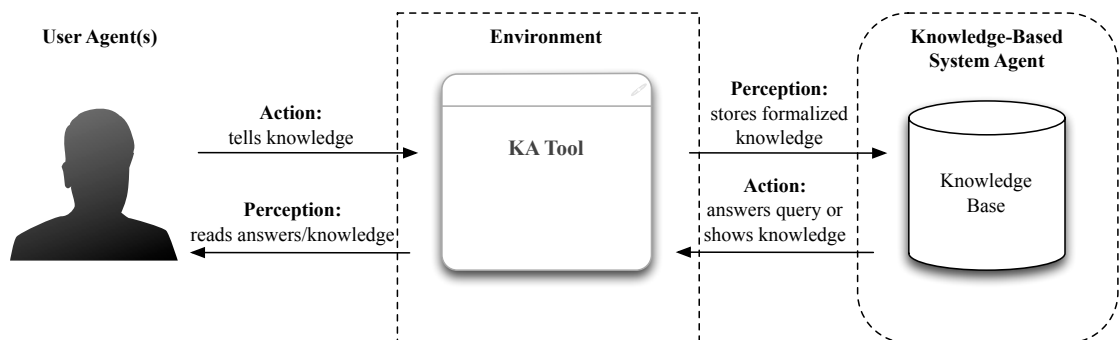


Figure 1.6: An agent-based view on knowledge acquisition for knowledge-based systems.

representation. Although these knowledge representations in this case are substantially different from each other, transfer of knowledge from the human to the knowledge base (and also vice versa) is required. In Section 1.3.2 it was already discussed that knowledge can be perceived

from the environment. The role of the environment for knowledge transfer becomes clear when considering once more equation 1.1, denoting that (abstract) knowledge can be converted into the agent's internal representation if access is possible. That is, agents can emit knowledge into the environment in some way. That knowledge can then be consumed and converted into the internal representation by another agent *iff* a suitable access function is available. Hence, that environment can be used for knowledge communication, but strongly depends on access capabilities. When considering knowledge acquisition in the knowledge level perspective, this kind of access functions have a role of particular importance. When considering human and knowledge-based system agents, the symbol level representation as well as the access functions strongly differ in general. They have to be realized by the knowledge acquisition tool, which forms the environment of knowledge transfer in this context.

Consequently in this perspective, KA-tool design means to design and establish appropriate means of access to knowledge—for the knowledge-based system agent *and* the human user. The agent-based perspective for knowledge communication allows to analyze how a knowledge acquisition tool supposed to be designed to make the perception and the emission of knowledge as efficient as possible for the involved agents.

While in Section 1.2.1 the need for a cognitive environment for a social process for exchange of expertise among participants was explained, here the structure of an (agent) environment for knowledge transfer towards the knowledge base (and vice versa) is discussed. Both aspects taken together provide a basis for the development of an effective knowledge acquisition environment for mixed-initiative knowledge acquisition.

1.4 Contribution of this Work

"..KA for systems development is fundamentally an anthropological activity."

Regoczei and Plantinga, 1987, [RP87]

The significance of this statement has not decreased in the last 25 years. On the contrary, with all technological advances considering computational power, knowledge engineering practices, and technical infrastructure, it feels more truthful than ever. The knowledge acquisition bottleneck, mainly being rooted in social aspects, as for instance the competency dilemma, still exists. Therefore, this work discusses a knowledge engineering approach reflecting this statement made by Regoczei and Plantinga. Besides of the actual knowledge acquisition process it also focuses on the support of a social process to enable efficient and flexible collaborative knowledge acquisition.

1.4.1 Goals

In Section 1.2.3 two processes, the social process and the knowledge acquisition process (c.f. Figure 1.3), have been introduced and discussed. In the following, the goals of these two processes, as being important topics of this work, are summarized.

1 Introduction

1.4.1.1 The Goals of the Social Process

The major goals of the social process are as follows:

1. The knowledge engineers become familiar with the application domain and the socio-technical conditions of the participating experts. This is not only important for designing the knowledge base and coordinating the knowledge formalization, but also to be able to specify suitable project specific knowledge acquisition tools.
2. An important goal is the design of a knowledge acquisition tool, which is custom-tailored to the conditions and needs of the domain specialists. Due to the competency dilemma, close and ongoing cooperation of knowledge engineers and domain experts is a prerequisite for the specification of an appropriate tool.
3. Domain experts are made familiar with the general principles of knowledge engineering, such as the need for knowledge formalization. Further, they have to assess and get used to the customized knowledge acquisition tool.

The precondition is first of all an appropriate cognitive environment for learning and communication. It is important to enable active participation of the domain specialists within the process at low socio-technical barriers. The knowledge engineers should be provided with illustrative information about the application domain as learning material, supplementing the cooperative sessions.

1.4.1.2 The Knowledge Acquisition Process

Simple things should be simple, complex things should be possible. Alan Kay

In this work, we propose the application of the mixed-initiative knowledge acquisition approach, which is an intermediate solution between direct and indirect knowledge acquisition. In knowledge engineering a wide range of different tasks, requiring different competencies, are required. Mixed-initiative knowledge acquisition proposes, that every participant is socio-technically enabled to take action. Especially for domain specialists, the following two challenges of human-computer interaction need to be considered:

1. **Reading and Comprehending a Knowledge Base:** The passive task of perceiving and comprehending an existing knowledge base by the use of a knowledge acquisition tool already is a challenge of human-computer interaction by itself. The perception (access) of a knowledge base and its alignment to the knowledge in the persons head is a non-trivial process. It implies presentation of knowledge base entities but also means for selection and navigation of the knowledge base content. There is only few work published addressing this issue [BF10].
2. **Modifying a Knowledge Base:** Perceiving and understanding (a fragment of) the knowledge base (1) is the precondition for making modifications. An authoring tool needs to be designed according to the principles of usability, carefully considering the user profile.

The human-computer interaction perspective of tool-supported knowledge base authoring has hardly been addressed [Jon88, Hen88]. In the context, of the semantic web a helpful general model of human-computer interaction for tool supported interaction with formal knowledge has been proposed [HSE11]. It can serve as a general guideline for the design of a knowledge authoring tool.

The second aspect, the actual editing of the knowledge base, is certainly the more demanding task. However, the first point is more decisive for the mixed-initiative approach, forming the prerequisite for taking any kind of action in a sense of active participation. Additionally, we claim that once this first stage being achieved, the second stage is not too hard to reach, if a reasonable editing paradigm and user interface is provided. Therefore, the organization and presentation of formalized knowledge for the alignment to a person's mental model is a key challenge for effective mixed-initiative knowledge acquisition. It should be the major design goal for a knowledge authoring environment.

The two processes cannot be considered to be separated, but are running in parallel being tightly interwoven. The social process can be considered as a process of continuous improvement for the actual knowledge acquisition process.

1.4.1.3 Purpose of this Work

"From these examples it is clear that it [active participation] is possible, but there is a considerable need for support from the tools used to create these systems."

Schilstra and Spronck [SS01]

The purpose of this work is to provide a comprehensive overview of the *document-centered knowledge acquisition* approach and its potentials to support the social process. The idea of a collaborative knowledge acquisition process [GS97, HJ02], also by active participation of domain specialists [SS01], is not new. The document-centered approach however introduces a new category of tools, which is well-suited as an environment to support the social process in collaborative knowledge acquisition with active participation. In particular it allows for:

- Active participation of specialists at very low socio-technical barriers.
- Smooth and precise tool customization.
- Flexible content structuring according the specialists' mental model.
- Seamless ongoing improvement of the tool customization and content structuring.
- Mixed-initiative knowledge acquisition including decomposition of modeling tasks.
- Agile & incremental development including instant testing.
- Exchange of expertise in a social process.
- Convenient inclusion of illustrative content describing the domain.

1 Introduction

We provide a complete discussion of the general principles of document-centered knowledge acquisition including its advantages and limitations. Then, guidelines for its project-tailored application in practice are introduced. Technical principles for implementation of a document-centered knowledge acquisition environment are presented. Additionally, a mechanism enabling the cost-efficient implementation of project specific customizations for such kind of tools is introduced. The application of document-centered knowledge acquisition is outlined by reporting multiple real-world knowledge acquisition projects.

1.4.2 Scope

*“Contributions to AI may be either flavor, i.e.,
either to the knowledge level or to the symbol level.”*

Allen Newell [New82]

Knowledge-based systems development fundamentally is concerned with the construction of knowledge bases at the symbol level. In fact, the selection or design of a knowledge representation at the symbol level is a prerequisite for the implementation of a knowledge base. Nevertheless, in knowledge engineering there are certain problems and phenomena, for instance considering the challenges in context of the social process, being independent of what kind of representation is used at the symbol level. According to the knowledge level perspective of knowledge acquisition, as discussed in Section 1.3.3, this work is concerned with the transfer of knowledge through the (knowledge acquisition) environment and the analysis and design of the corresponding access functions. Hence, the contribution of this work has to be attributed to the knowledge level. The symbol level is only discussed in a general fashion. The document-centered knowledge acquisition approach can be applied using (almost) any symbol level knowledge representation. In particular, the selection or design of a suitable knowledge representation at the symbol level for a given application task is not subject of this work. For this task fundamental literature is available [SAA⁺01, vHLP07].

Also the approach is not intended to provide a full-fledged knowledge engineering methodology. Organizational aspects, as for instance the detection of demands for knowledge-based solutions, are not discussed. For the application of the presented knowledge acquisition approach, we assume that the project context already is known. That context includes a precise requirement specification of the desired knowledge system, the knowledge sources, and the participants. Further, we assume that at least one experienced knowledge engineer is available to play the leading role in the knowledge engineering process.

This work describes a novel and generic approach for manual knowledge acquisition for knowledge-based systems by the usage of a specific category of knowledge acquisition tools, the document-centered knowledge acquisition tools. It can be combined with various knowledge engineering methodologies and is well-suited for rapid prototyping. Due to its agile and light-weight nature it is reasonable to choose an agile knowledge engineering methodology for that purpose. Nevertheless, it turns out that CommonKADS [SAA⁺01], although being a comprehensive heavy-weight methodology, can be combined with document-centered knowledge acquisition if a more structured approach is desired. More details about this combination will be discussed in Section 3.6.

1.5 Structure of this Work

This work is organized in eight chapters. In this first chapter, concluding with this outline, the required preconditions for the approach have been discussed. The remainder of the this work is organized as follows:

- **Chapter 2** gives an overview of the commonly used state-of-the-art knowledge authoring techniques. Five categories of techniques are distinguished and for each category examples from the literature are discussed.
- **Chapter 3** introduces a general method of knowledge formalization for knowledge-based systems development, the document-centered knowledge acquisition. The characteristics and potentials of documents with multimodal knowledge as a means for knowledge capture are presented. Its advantages and challenges and the requirements for a corresponding tool are discussed. To illustrate the approach, we sketch how it can be employed for a selection of exemplary application scenarios.
- **Chapter 4** introduces a process how knowledge engineers can systematically improve a document-centered knowledge acquisition environment during the progress of a project. This improvement, which is part of the social process, is achieved by meta-level design of the environment in an evolutionary process, running in parallel to the actual knowledge acquisition activities.
- **Chapter 5** discusses a number of technical challenges implied by the application of the document-centered knowledge acquisition approach. These technical tasks, together with solution methods, are discussed with the aim to support the implementation of the basic components of a generic document-centered knowledge acquisition system.
- **Chapter 6** presents an implementation of a document-centered knowledge acquisition environment. The system *KnowWE* is based on a (semantic) wiki engine and is designed according to the requirements as discussed in Chapter 3.
- **Chapter 7** provides an overview of several case studies of document-centered knowledge acquisition using the KnowWE system. The experiences made in these projects, dealing with different subject domains and different symbol level knowledge representations, are reported.
- **Chapter 8** provides a summary of the present work. Additionally, an outlook is given and further research questions with respect to document-centered knowledge acquisition and the meta level process are discussed.

2 Approaches for Knowledge Base Authoring

Before the document-centered knowledge acquisition approach is discussed, we provide a brief overview of the knowledge authoring approaches most commonly used.

For the very first attempts of building expert systems, the knowledge base has been created directly in the target knowledge representation, as for example LISP [McC78] code. This workflow has proven to be inconvenient and basically prevents direct participation of domain experts within the development process. In the following, we discuss the different approaches that can be used for knowledge base editing in a more comfortable way. We outline the approaches form-based authoring, tabular knowledge acquisition, graphical knowledge representation languages, domain specific languages, and electronic (text) documents. For each category, representative tools are briefly discussed. The different approaches are not exclusive, i.e., in practice can be combined. While some tools apply to one category in a strong way, other tools mix up two or more of the discussed aspects. The three techniques form-based authoring, tabular knowledge acquisition, graphical knowledge representation languages can be summarized to the use of graphical user interfaces (GUI). One example for combining these different kinds of GUI-based authoring techniques is the tool CLASSIKA presented by Gappa et al. [GPS93]. It uses forms, as well as tabular editors and graph-based representations for hierarchies.

At the end of this chapter, existing work on knowledge authoring with electronic documents is discussed, leading over to the document-centered approach being the main subject of this work.

2.1 Form-based Authoring

Form-based authoring is a method very commonly used in GUI-based editing. Graphical user interfaces are forming the human-computer interaction paradigm, which is probably most widely used in general, not only considering knowledge acquisition. It is usually characterized by the use of windows, menus, tool bars, tabs, and panels, which are used to organize different kinds of input forms. Since the eighties, this paradigm has been widely used to create knowledge acquisition tools (e.g., [Jon88, SEA⁺02, KV03, Bau04, NL05, DSW⁺00]). Figure 2.1 shows the OWL editor of Protégé [KFNM04], which is a GUI-based knowledge acquisition tool for authoring ontologies in OWL. Entities of the ontology, e.g., classes, properties and individuals, are managed in hierarchies, which are organized in tabs. They can be created and modified using forms. Another, example for a GUI-based knowledge acquisition tool is KnowME [Bau04], which allows to create diagnostic knowledge bases in d3web¹ as shown in Figure 2.2. It shows a similar structure managing the inputs and the solutions in hierarchies. Further, form-based editors for rules and other kinds of knowledge representations are provided.

2 Approaches for Knowledge Base Authoring

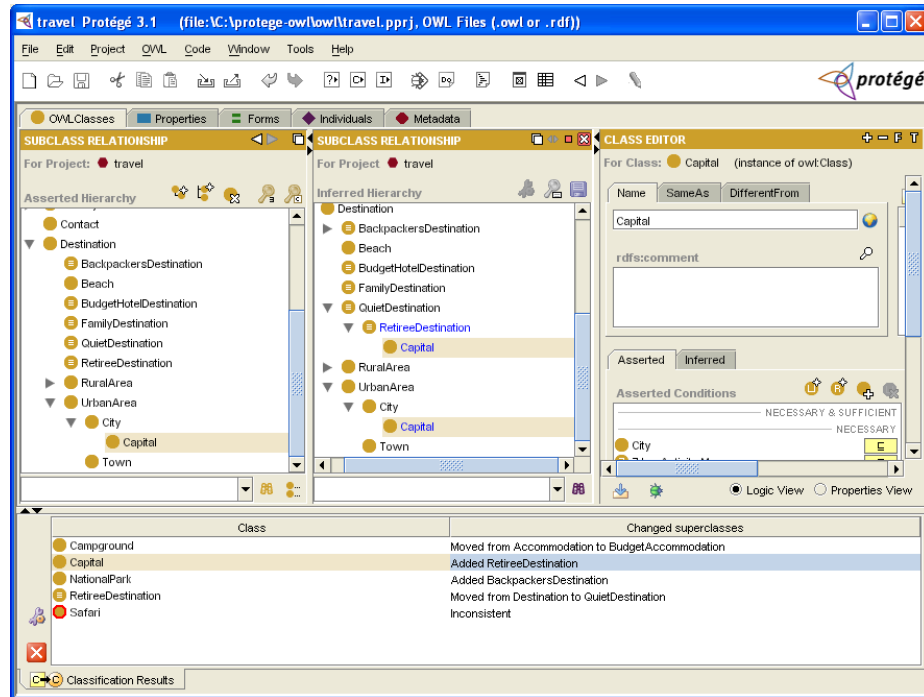


Figure 2.1: Protégé: A GUI-based knowledge acquisition tool for ontologies [KFN04].

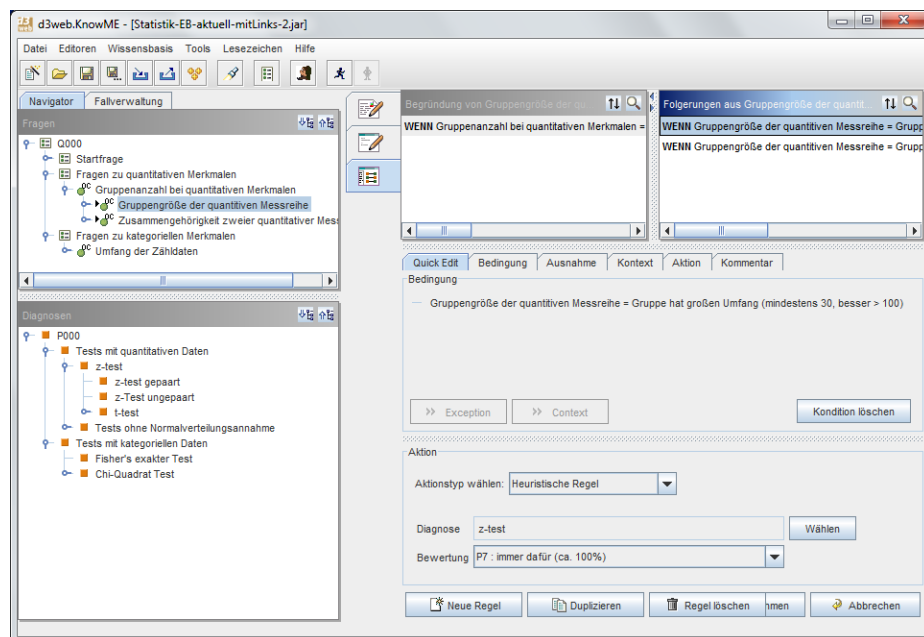


Figure 2.2: KnowME: A GUI-based tool for diagnostic knowledge bases [Bau04].

Form-based knowledge authoring tools for knowledge representations of high expressiveness can become complex, providing a high number of different interaction elements. Especially for domain specialists without knowledge engineering background, the use of these kind of tools is highly challenging. For this reason, often custom-tailored knowledge acquisition tools for a specific domain have been build [Mus88, Mus89b]. Those tools have reduced complexity and represent the knowledge in a way, which is intuitive to the domain specialists.

In general, form-based interfaces commonly are easy to use, as it is a wide-spread and well-used interaction paradigm. Further, it prevents the user to a far extent from making errors on the syntactical level, and visually shows him the possible actions that can be taken in the respective context. However, it often does not provide means for flexible structuring of the content. In Section 3.2 a more detailed comparison between GUI-based authoring and the approach proposed in this work.

2.2 Graph-based Authoring

Often knowledge can conveniently be represented by graphs. There is a wide range of graph-based knowledge representation languages known today [CM08]. Semantic networks or ontology schemas often are edited as graphs. In particular, for procedural knowledge such as, workflow, guidelines, or (business) process models, graph-based representations are used for editing. An example for modeling executable medical guidelines is provided by the DiaFlux language with by Hatko et al. [HBBP12], shown in Figure 2.3. An interactive graphical editor

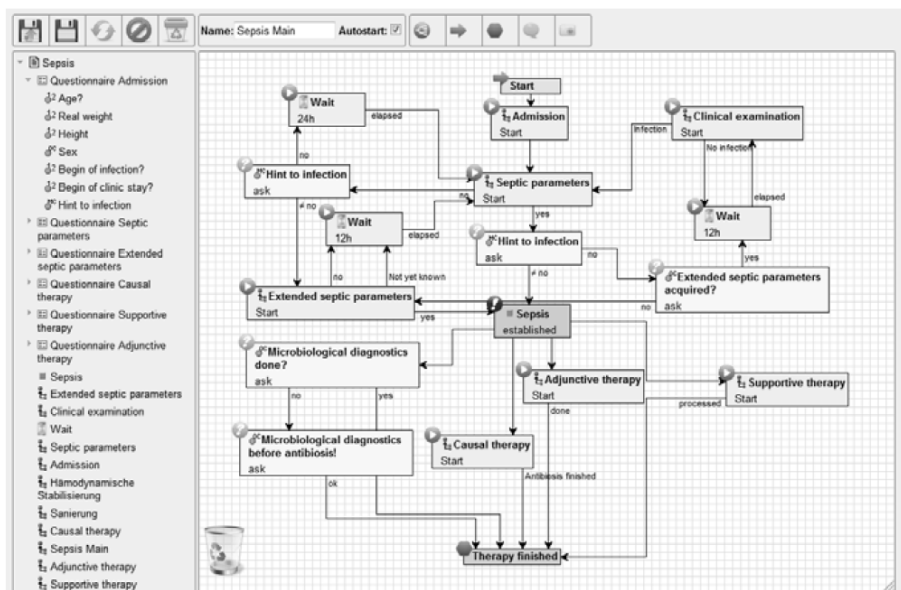


Figure 2.3: The DiaFlux editor for executable guidelines: A treatment plan for sepsis [HBBP12].

¹<http://www.d3web.de>

2 Approaches for Knowledge Base Authoring

allows to create and modify nodes and edges. The node positions can freely be defined by the user to allow for a comprehensible layout and presentation of the knowledge. The flowcharts edited by the user are transformed into an executable representation, which is ready for testing instantly. Further, the tool provides a debugging view, allowing visual tracing of the flowchart execution.

Another example is provided by Ferstl et al. [FSA⁺94], presenting a tool for modeling business processes in the SOM (Semantic Object Model) formalism. SOM defines a graphical language where objects are defined as nodes that can be interconnected by transactions. Objects can represent a customer or a department for example, while transactions transfer messages or results between objects. Further, the SOM meta model allows for the representation of events and tasks. An example is depicted in Figure 2.4.

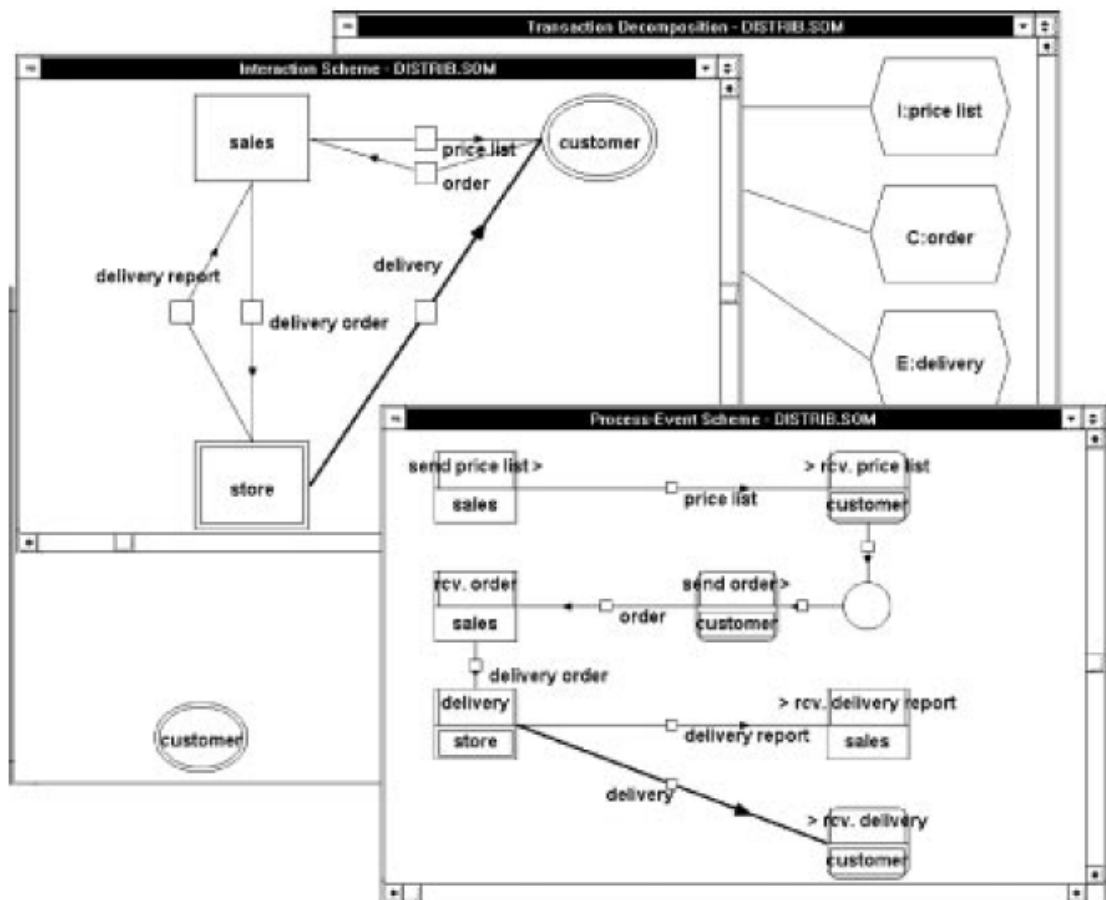


Figure 2.4: A tool for modeling business processes in SOM by Ferstl et al. [FSA⁺94].

Other examples for knowledge modeling by the use of graphical languages are *More* by Kahn et al. [KNM85] and *MOLE* by Eshelman [EEMT87].

2.3 Table-based Authoring

Another method for representing problem-solving knowledge is the use of tables. Tabular knowledge formalization patterns [Pup00], such as decision tables or diagnosis score tables [Pup90], are widely used in knowledge acquisition. Figure 2.5 for example shows a diagnosis score table of the tool CLASSIKA [GPS93]. It connects symptom values in the column header with the diagnosis denoted in the header row by predefined score values. The table can be used for directly editing the knowledge base.

Diagnosis Table for "Psoriatic-arthritis", ...			
Conditions	Diagnoses	Psoriatic-arthr...	Spondylarthrop...
Predisposition			
A priori Frequency		fairly of...	fairly se...
Basic Valuation			
Spondylarthropathies		+ P5	+ P4
X-ray-joint-finding			
- psoriasis-typical changes		+ P4	+ N3
X-ray-spine-finding			
- syndesmophyts, parasyndesmophyts etc.			+ P5
kind-gastro-intestinal-troubles			
- increased bowel movements - less than 3 times a..			* P5
SEX			
= male			+ P4
sum-swollen-large-joints			
> 0			+ P4
formation-of-dandruff			
- at the hairy region of the head		+ P5	
- a psoriasis is known		+ P5	N5
result-of-skin-examination			
- psoriasis vulgaris		+ P5	
sum-arthritis-synovitis			
> 0		+ P4	
swollen-joints			
- sternoclav right			necessary
former-spine-problems			
Hla-typing			
- HLA-B27 negative			always pro = 100%
- HLA-B27 positive			P6 = almost always pro = 95%
tendons,-sheaths,-insertions			P5 = mostly pro = 80%
- inflammation of the achilles-tendon insertion			✓ P4 = usually pro 60
kind-of-joint-changes			P3 = often pro = 40%
- dactylitis		+ P4	P2 = sometimes pro = 20%
location-spine-troubles			P1 = seldom pro = 10%
- small of the back-loins			
diff.counter.degen.-infl.spine			
< 0			N1 = seldom contra = 10%
> 1			N2 = sometimes contra = 20%
tendon-insertion-bone-pain			N3 = often contra = 40%
- pain at the Achilles tendon-heel		* P4	N4 = usually contra = 60%
dactylitis			N5 = mostly contra = 80%
- single fingers		+ P4	N6 = almost always contra = 95%
spinal-column-finding			never = always contra = 100%
- positive menell-sign			
family history			
- a psoriasis		* P5	
- bowel diseases or diseases with diarrhea			
known-eye-diseases			
			-> Rule Form
			New rule
			Delete Rule

Figure 2.5: Diagnosis score table of the tool CLASSIKA [GPS93].

2 Approaches for Knowledge Base Authoring

Another method for table-based representation of knowledge is the *Extensible Tabular Trees* (XTT) formalism. It is a combination of table-based and graphical knowledge acquisition [NL05], where the nodes in a hierarchical graph are given by tables. Figure 2.6 shows the tool HQEd, which supports the definition of knowledge bases in XTTs [NLK11].

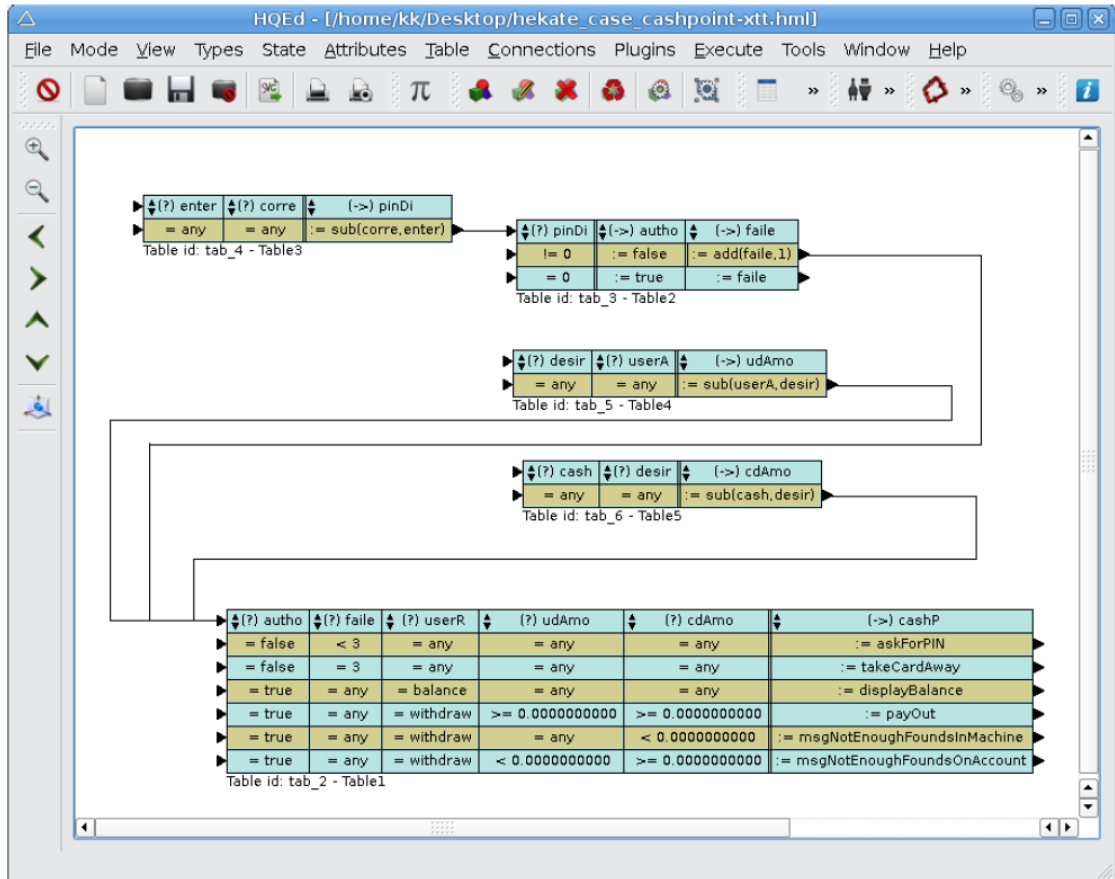


Figure 2.6: The HQEd tool for the definition of Extensible Tabular Trees [NLK11].

2.4 Authoring by Domain Specific Languages

The concept of a *Domain Specific Language* (DSL) received increased attention in software engineering within recent years. One can distinguish two major categories of DSLs, being graphical and textual languages. Graphical languages have already been discussed in Section 2.2. Hence for this work, we always refer to *textual* languages when using the term *DSL*.

Fowler [Fow10] defines a domain specific language as a language that meets the following three criteria:

- **Computer Language:** A DSL is a computer language and thereby has a formal character. It is or at least could be processed by a computer in some way.
- **Domain Focus:** A DSL is used in one particular application domain of narrow scope. This is in contrast to the so-called *general purpose* programming languages.
- **Limited Expressiveness:** A DSL has (very) limited expressiveness, that intently allows not for more complex tasks as its intentional use of the domain focus.

One example for this category, while not having a direct research background, is *Drools expert*² by JBoss, which is a free business rules engine [Bro09]. Figure 3.9 shows a screenshot of the authoring tool. It allows to define the rules by a domain specific language within different

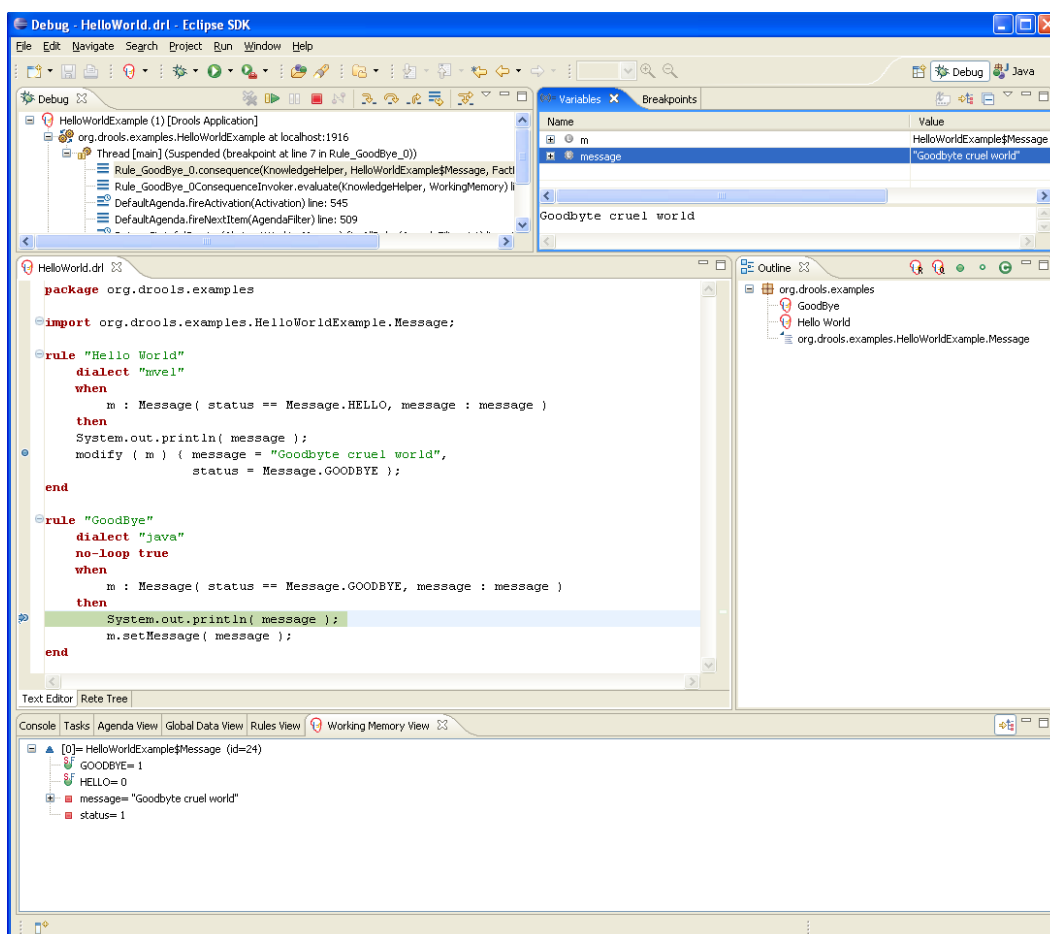


Figure 2.7: The knowledge authoring environment of drools expert [Bro09].

²<http://www.jboss.org/drools/drools-expert>

2 Approaches for Knowledge Base Authoring

source files. Being integrated within the Java programming language, it however does not really comply to the criteria of limited expressiveness stated by Fowler.

Similar editors are existing for the rule engine Jess³ [Hil03] and the expert system shell CLIPS⁴ [Wyg89]. Just as the Drools Expert example, those tools more resemble programming environments for software engineers than knowledge acquisition tools. For Jess also a graphical tool was created [JGD04]. However, using forms and other GUI elements, that tool then belongs to the category of graphical/form-based interfaces.

Decker [Dec98] was one of the first researches to propose the use of domain specific languages for knowledge acquisition. However, it has not yet been widely explored in practice by the research community.

2.5 Document-based Authoring

The use of electronic documents for direct knowledge base modeling has hardly been addressed by researchers yet. This excludes (semi-) automated generation of knowledge bases by using natural languages processing methods on preexisting documents. Also, 'common' specification documents, describing the knowledge base content and inference mechanisms as proposed for example by CommonKADS [SAA⁺01], are excluded from this consideration. Those only serve as a specification for the actual implementation, being performed by the system developers later in a distinct step. Hence, no direct knowledge base modifications are possible. Much research can be found for the concept of *active documents* [GLR09, HM00, WKJ⁺01]. When studying this literature, it appears that there is no unique and consistent notion of the term 'active document', ranging from question answering applications over user adaptive documents to agent-based architectures of documents. Due to the imprecision of the term and the low relevance of the corresponding research for this topic, it will not further be used for this work.

Martin [Mar95] proposed the interweaving of text document elements with a knowledge base consisting of a conceptual graph. The main purpose of this approach was to create an intelligent retrieval system for the document corpus, which exploits typed relations between document elements and concepts of the knowledge base. The knowledge base however is developed in parallel to the documents. Hence, the approach is not very much related to the perspective of this work, where knowledge bases for problem-solving are (exclusively) created by modifying electronic documents.

The first approach in this direction was proposed by Gaines and Shaw [GS99]. They present a document authoring system, that in addition to common content such as text and figures, allows for embedding of executable knowledge models by the use of a graphical language, similarly as discussed in Section 2.2. For the graphical language elements an additional graphical editing component is provided. As an example, a semantic network representing the credit control policy of a bank is created within a set of documents. Gaines and Shaw were the first to claim that it is attractive to represent a knowledge base as a set of documents. They outline that it is advantageous to combine the formal model with content elements intuitively comprehensible

³<http://www.jessrules.com>

⁴<http://clipsrules.sourceforge.net/>

for people such as text, images, and movies. In that way, documents are created, which are understandable by both, humans and computers.

Another approach for creating knowledge models by editing electronic documents is reported by Molina and Blasco [MB03]. Also they emphasize understandability of the documents by humans as well as by the computer. Further, they distinguish the document in *static*, *automatic*, and *users* content elements. For the formation of the knowledge models, they propose a pre-defined structure of the documents, alternating the content categories mentioned. In that way, documentation, formal knowledge parts, and parts with information generated by the system alternate in a particular order. The approach also focuses on intelligent user support by the use of meta knowledge, which allows the system to inform the user about incompleteness or inconsistency of the edited model. Instant processing of the defined knowledge establishes an *intelligent document processor*. For the actual formalization task of the knowledge model, for example deductive tables, mathematical functions, or probabilistic causal relations are proposed.

2.6 Approach of this Work

In this work, we propose a knowledge acquisition approach based on editing electronic documents, similar to the ones presented by Gaines et al. [GS99] or Molina et al. that just have been discussed [MB03]. One major difference between the GUI-based knowledge acquisition interface and the document-based knowledge acquisition lies in the persistence structure. While some GUI-based tools also allow for the insertion of informal knowledge such as comments and images, the user has no control of the 'spatial' structure of all these content elements such as order or distance. In document-based case, the user intuitively creates a reasonable and memorable structure within a document, as well as relations between documents. GUI-based tools in contrast generate these spatial characteristics 'on the fly' when showing the knowledge base content being read from a data base. Hence, the spatial structure of the displayed content element there is a characteristic of the tool and not of the (user created) content.

The focus of this work is to also support the entire social process of knowledge acquisition as discussed in Section 1.4. This includes a focus on tool customization according to the project's and users' characteristics, similar as addressed by Musen [Mus88, Mus89b] in the context of GUI-based tools (c.f. Section 2.1).

For the actual formalization of knowledge embedded within the documents, we focus on the use of (textual) domain specific languages as discussed in Section 2.4. The use of DSLs for this purpose provides two major advantages: It allows smooth and precise tool customization, while the actual knowledge base is formed. Further, it is very suitable to support a workflow of mixed-initiative knowledge acquisition, where single formalization tasks are decomposed and performed by subsequent action of different participants. In the following chapter, the *document-centered knowledge acquisition* approach is introduced in detail.

3 Document-Centered Knowledge Acquisition

Within the last couple of years the work with electronic documents found its way into the daily lives of most people, ranging from the management of a digital photo library to text-documents for official correspondence or excel charts for managing housekeeping money. The computer-based interaction with documents has become a banality such as using a car or a phone. Based on this human-computer interaction paradigm a knowledge acquisition method for intelligent system engineering can be derived. It has the potential to support well the requirements discussed in Section 1.2.1. In this chapter the so called *Document-centered Knowledge Acquisition* method (in short DCKA) is introduced. The advantages and challenges of this approach are discussed, as well as the requirements for a corresponding knowledge acquisition tool are being formulated. Further, scenarios for different exemplary use cases, sketching the document-centered knowledge acquisition approach for a wide range of knowledge formalisms, are presented.

3.1 DCKA in a Nutshell

Graphical user interfaces are forming the most popular paradigm of human-computer interaction. The most obvious advantage, for example compared to command-line based interaction, is that all actions available in the current context are visualized to the user and can simply be selected by a pointing device reducing the risk of syntactical errors. Research and practice in the domain of knowledge acquisition tools is focused on this human-computer interaction paradigm since the eighties (e.g., [Jon88, KFN04, SEA⁺02, KV03, Bau04, NL05, DSW⁺00]). While providing some obvious advantages, GUI-based tools have their drawbacks, for instance requiring complex cognition and strongly limit the kind of information that can be entered [VKBKs07]. Therefore, the GUI-based approach is not necessarily the best approach for every knowledge engineering scenario.

Foundational literature of human-computer interaction describes various kinds of different user interface paradigms [SP09]. A classical paradigm being used since the early days of digital computers, is editing and management of digital documents by interactive editors. There are a number of characteristics making the exploration of its potentials for knowledge engineering promising. Having evolved within the decades, e.g., WYSIWYG¹ editors as well as web-based document management system being introduced, the interaction paradigm is still playing a very important role in information management today. One major reason for its broad acceptance in our daily lives is its clear metaphor, imitating the management of real (paper-) files and folders. This simple and well-known metaphor helps people with the task of structuring the content in a memorable way, providing quick and simple access to all content elements.

¹<http://en.wikipedia.org/wiki/WYSIWYG>

3 Document-Centered Knowledge Acquisition

However, it has hardly [GS99, MB03] been explored yet as a means for knowledge acquisition for knowledge-based systems. A comprehensive discussion of using electronic documents combined with knowledge markup languages for knowledge engineering is the purpose of this chapter. At first however, we briefly summarize the major characteristics of electronic document authoring in general, that provide the well-known and successful user experience:

- **Full Content Control:** The user has exclusive control of the document content. She or he is free to make arbitrary changes. In particular, the content elements can be inserted in the document in arbitrary order. User modifications are always accepted and stored persistently. Further, the system does not perform modifications of the documents without the user being involved.
- **New Documents:** The user can freely decide to create a new document assigning a name of his choice.
- **State by Content:** Only the current content is important to represent the state of the document corpus. Usually, many different sequences of editing actions can lead to a specific document content. However, the order of the actions should not influence the behavior of the authoring environment or the results in any way. Hence, this can be considered as commutativity of editing actions.
- **Content Presentation:** The presentation of the content is strongly determined by how the content has been formed by the user. Especially for non-WYSIWYG editors it is important that the compliance of the presentation and editing view of a content element is obvious and intuitively recognizable by the user.

These are the most fundamental characteristics inherent to most digital document management systems valid for any kinds of document types or editors. The unconstrained contribution of content makes life easy for the user but on the other hand leads to a document corpus, being non-recognizable for machines. This kind of content is meaningful to humans only and therefore in the first place not practicable for knowledge engineering as natural language processing methods are not (yet) reliable enough. For this reason, we need to extend the approach to allow for the definition of structured knowledge with computer interpretable semantics.

3.1.1 Documents for Knowledge Base Development

As discussed, until now most knowledge engineering tools carry out the human-computer interaction task of knowledge formalization by employing graphical user interfaces. The document-centered authoring approach, in contrast, aims to extend the paradigm of editing digital documents in a seamless way to allow for authoring of computer-interpretable knowledge. The major goal is to make the knowledge, contained in a set of documents, in the end understandable by both—humans and machines. This refers to the notion of *access* to knowledge according to Newell as discussed in Section 1.3. This extension is a non-trivial step, where the general conditions and human aspects of knowledge engineering, including the cognitive challenges, need to be incorporated. The goal is to establish a knowledge authoring environment that is suitable to

provide a cognitive environment as discussed in Section 1.2.1 and support the social processes of knowledge engineering.

The document-centered knowledge authoring environment provides access to a set of documents, that can be modified and extended by employing some basic text-editing interface, retaining all the characteristics of document authoring discussed above. The formalization process is performed by the use of formal syntax, which is (in the first place) predefined by the authoring environment. The basic workflow of creating a knowledge base by document authoring, is shown in Figure 3.1. After each document modification, statements complying to the syntax automatically are translated to the knowledge repository, being stored in the computer-interpretable format according to the employed symbol level knowledge representation formalism. Further, feedback is given to the user accordingly. In that way, the knowledge repository always contains the most recent version of the knowledge base corresponding to the current document contents. Additionally, the knowledge base is instantly ready for testing. The content of the knowledge repository further will be called *compiled knowledge base* or *symbol level knowledge base* in contrast to the document base, containing the human readable content.

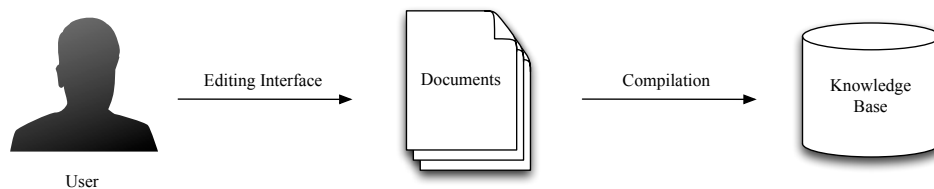


Figure 3.1: Structure of the document-centered knowledge authoring paradigm.

The documents form a human-oriented layer in between the contributors and the symbol level knowledge base. It is dedicated to provide human-centered access to and organization of the knowledge. The use of documents as a basis for knowledge transfer has two important implications:

- In contrast to the use of graphical user interfaces, an interface layer of documents has a persistent structure, that can be arranged conveniently in a memorable way. The decoupling of the human author and the symbol level knowledge base by this additional layer provides a large degree of freedom for structuring the content according to the users' needs.
- As the knowledge engineers are able to control the compilation process on the technical level, the system capabilities can be extended or modified, without the look and feel being changed for the user (as the documents do not change). In that way, the customization towards a project-specific tool can be driven in a smooth incremental process.

These two implications can be exploited to enable an effective mixed-initiative knowledge acquisition process. Their nature, potentials, and risks are the main subject of this and the next chapter.

The Document-centered Knowledge Acquisition Environment The knowledge acquisition scenario envisioned in Section 1.2.1 considers a community of multiple participants being involved within the knowledge engineering process. As mixed-initiative knowledge acquisition proposes direct manipulation of the knowledge, the access to the documents needs to be coordinated. All participants should be able to access and edit the document easily. A schematic view of this collaborative working process with documents is shown in Figure 3.2. To beware of editing conflicts however, one should prevent the case that multiple persons are editing the same document at the same time. Hence, when one document is edited it should be locked for others, as known from common collaborative document management systems. Additionally, for the safe development and progress monitoring, a versioning mechanism, showing all the prior states of the documents, should be provided. Beside these functions, a trigger mechanism for the automated compilation of the knowledge base from the documents after changes is required, completing the extension from a set of accessible documents to a document-centered knowledge acquisition environment. The design of this kind of authoring tool should stick as close as possible to the original document editing paradigm to retain the well-known user experience as far as possible.

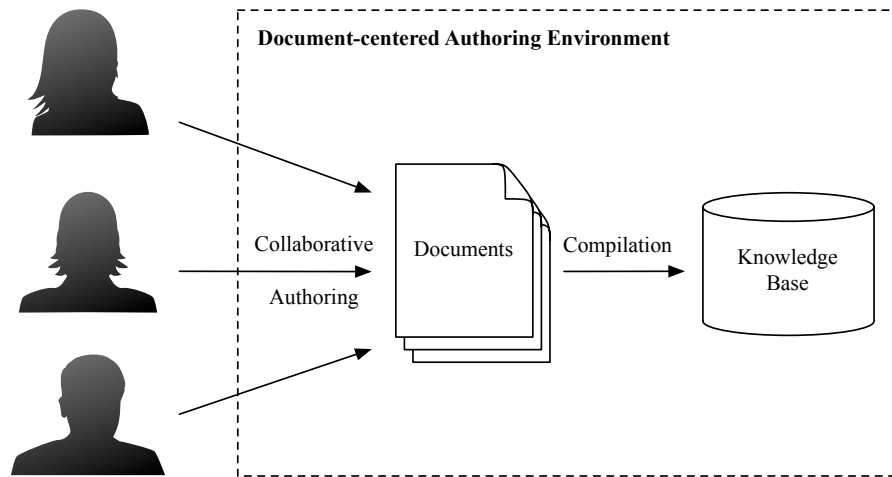


Figure 3.2: Schematic view of the collaborative workflow in DCKA.

3.1.2 Knowledge Markup Languages

Content management as electronic documents is a common approach. Creating an executable knowledge base using some formal syntax implies some additional technical challenges. The formal syntax and the way it is translated to the symbol level representation plays a fundamental role in this document-centered knowledge acquisition approach. This formalization method can be applied to most common symbol level knowledge representation languages, as for example rule-based or logics-based formalisms. In the following, knowledge formalization with knowledge markup languages is defined in a more technical and formal way.

In formal terms a knowledge base of the knowledge representation language \mathcal{K} is a set of words of \mathcal{K} . The set of all possible knowledge bases of \mathcal{K} is then given as: $2^{\mathcal{K}}$. To form a knowledge base using an input syntax \mathcal{L} , we define a mapping $\tau: \mathcal{L} \mapsto \mathcal{K}$.

Now we can define a *knowledge markup language* (or short *markup*), forming the formalization method of the DCKA approach, as a triple comprising in input syntax, a target knowledge representation, and a corresponding mapping:

$$\mathcal{M} = \{\tau, \mathcal{L}, \mathcal{K}\}$$

We call a markup language \mathcal{M} *complete* according to \mathcal{K} if τ is surjective, as then any knowledge base of $2^{\mathcal{K}}$ can be created by \mathcal{L} . We call a markup language \mathcal{M} *injective* if τ is injective, that is every knowledge base can be created only by one particular set of words of \mathcal{L} .

For some concrete examples of markup languages and their use within documents, the interested reader may refer to the example scenarios in Section 3.5 or to Chapter 7, where real-world case studies are described.

Knowledge Markups as Domain Specific Languages: Knowledge markups are related to the concept of domain specific languages (DSL). Fowler [Fow10] distinguishes internal and external DSLs. Internal DSLs are restrictions of existing (general purpose) programming languages while external DSLs are defined independently from scratch. External DSLs are usually either used to generate code or to populate a semantic model. In that sense, a knowledge markup language is an external DSL that is used for fragments of documents to populate a semantic model that is established within the knowledge repository.

The term DSL does not have a unique and precise definition and is not always used in a consistent way. In literature for example the term *domain* in domain specific language is interpreted in different ways. Sometimes it is used to refer to the target model that is populated by the language (e.g., production rules, regular expressions, dependency networks). Often however, the term describes the application domain (e.g., sales management, cardiography, aeronautical engineering). For clarity, we will primarily use the term (knowledge) markup instead of DSL, indicating that the target model is a fixed characteristic of the language, determined by the knowledge representation. A domain specific knowledge markup in consequence denotes a markup that is specifically tailored towards some application domain.

3.1.3 Multimodal Knowledge

In this section, we discuss the nature of the document content, when markup languages are used to form a knowledge base. Therefore, the concept of *multimodal knowledge*, including distinct content categories, is introduced.

In Section 1.3 the knowledge level view on the knowledge acquisition task has been presented. There, knowledge acquisition is considered as communication of agents on the knowledge level via the environment. In the following, we apply this perspective on the document-centered knowledge acquisition approach. As discussed in Section 1.3.3, we consider the knowledge base as a (knowledge-based system) agent. Further, there are multiple human agents that aim to create an (executable) knowledge base. All these agents share an environment, the knowledge

3 Document-Centered Knowledge Acquisition

acquisition environment. In the document-centered case the environment—apparently— is consisting of the documents, completed by a testing interface for the knowledge base. Every agent can perceive the document contents. The knowledge-based system agent only has read access to the documents, while the human agent(s) can freely modify the environment. They accumulate the domain knowledge relevant for the executable knowledge base in the documents. Considering the access functions of agents (c.f. Section 1.3.2) for perceiving the knowledge from the documents, human agents fundamentally differ from the knowledge-based system agent. The access function τ of the latter only is able to access the knowledge that is defined using the markup language, as this access function is implemented as characteristic feature of the DCKA environment. For human agents, especially domain specialists, access to markup language is non-trivial. In contrast, access to knowledge defined in natural language or figures is easy if the content is well illustrated.

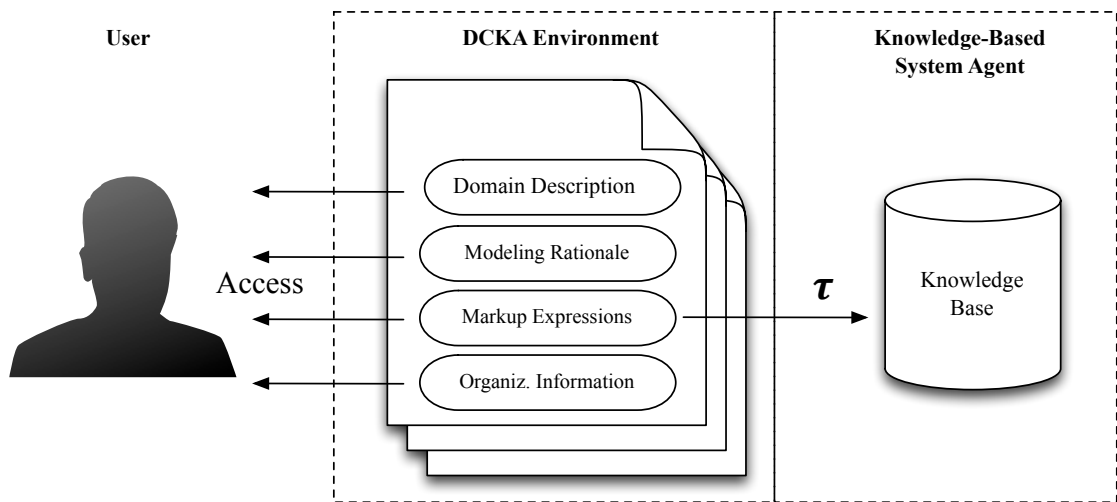


Figure 3.3: The four different content categories of multimodal knowledge.

We distinguish four different categories of content that are employed for the description of some topic of the domain during the knowledge acquisition process.

- **Domain Description:** This informal description of the respective topic of the domain should provide a comprehensive view on the treated topic at an adequate level of detail. It is necessary that a participant can understand the subject topic described and its role for the knowledge-based system. Therefore, a textbook style including illustrative figures, charts, or photos is appropriate. Indeed, textbook chapters can be used to create this kind of content if available in appropriate quality and level of detail. Hyper-links to topically related content elements (in other documents) should be provided to simplify research. Aspects considering the formalization of this knowledge into the knowledge base are completely omitted for this category.

- **Modeling Rationale:** The description of the domain in many cases can not be converted to formal knowledge in a straight forward way. The description is dedicated to explain general domain issues in a human perspective, while the formalized knowledge always implies aspects of automated problem-solving strategies. This content category therefore describes in an informal but precise way, how the part of the domain represented in this document is or should be modeled as formal knowledge using the markup language. This includes justification of what parts are formalized, as well as why and how this way of modeling will achieve the desired behavior of the knowledge base.
- **Markup Expressions:** This category contains markup expressions that create the required parts of the executable knowledge base. Reasonably, it should be organized next to the domain description discussing the corresponding subject topic and the corresponding modeling rationale. It is the only content category actually contributing to the compiled knowledge base.
- **Organizational Information:** This category contains information related to the development process itself such as:
 - Status of this subtopic (e.g., draft, prototype, tested, reviewed)
 - Things that still need to be done (e.g., todo lists)
 - Names (and contact) of participants that are working on or are experts on that topic
 - Discussions about the appropriateness of domain description or modeling
 - Information explaining the organization of the document base (e.g., navigation help)

Reconsidering the agent-based view, where the documents are forming the environment perceived by the user and knowledge-based system agent, the roles of these content categories can be defined more precisely.

Domain description and markup expression both contain domain knowledge about one and the same topic. However, the former cannot be accessed by the knowledge-based system agent, but easily by a human agent. The latter, can be accessed by the knowledge-based system agent, but access often is challenging for human agents. The purpose of the modeling rationale is to help human users to access the markup expressions. This meta-knowledge describes how the markup expression content can be accessed by explaining the symbol level semantics and relating it to the domain description.

The forth category comprising organizational information aims to support the development process and as such is not directly related to the domain. Therefore, it can be considered as optional (e.g., for single user projects) but is often helpful for the coordination of a collaborative development process.

The content elements belonging to these four categories are strongly differing with respect to multiple aspects, such as the syntactical shape (natural language, figures, tables, markup language) and the subject (domain knowledge, meta-knowledge, organizational information). Therefore, we call this inhomogeneous content, being tightly interwoven with each other and embedded into a document-space, *multimodal knowledge*.

3.1.4 The Document Space

In the context of multimodal knowledge, the different categories of content have been discussed. Now, the possibilities for structuring these content elements within documents are explained, also considering the potential for simplifying knowledge understanding and authoring.

3.1.4.1 The Document Space as a Graph

There are many possibilities of structuring content elements in documents. A document base is a hierarchical structure, as documents can be associated as groups and subgroups. Also on single document level, the content is organized hierarchically using chapters, sections, subsections, paragraphs, and so on. All these content elements, being aggregated according to the composite pattern, can be related to each other in different ways. Hierarchical inclusion is only one kind of relation between content elements. Naturally, a document has a sequential structure relating each content element logically with its predecessor and successor. Another category of associating content elements is cross-referencing. It is a valuable method to relate content elements, which are not associated hierarchically or sequentially. While the hierarchical and sequential relations are emerging naturally from the document structure, cross-references have to be included explicitly by the author by placing reference targets. A reference target can be a document, but also a specific content element within a document. An abstract example with four documents, illustrating the different content element relation types, is depicted in Figure 3.4. The content elements are forming a graph structure made up of the content element relations hierarchical inclusion, (in-document) successor, and cross reference. The graph emerging from the documents of example Figure 3.4 is shown explicitly in Figure 3.5. Incorporating the possibility of explicit cross referencing, any graph structure can be created within the document space. We call the generic graph structure, which emerges from the content elements and their relation types, the *document space*. It can freely be organized and modified by project participants to organize the knowledge conveniently. The edges of this graph give the user the opportunity of topic-related navigation, be it by following hyper-links or simple scrolling within a document.

3.1.4.2 The Document Space and Human Mental Models

As discussed in Section 1.4.1, the reading and comprehending of knowledge, which is a prerequisite for editing, requires an alignment towards the person's internal mental model of the corresponding knowledge. To make this alignment as easy as possible, one needs to consider the nature of the mental model of the participants. For insights about how knowledge is structured in people's mind, one should refer the corresponding research from psychology.

A Model of Semantic Memory The organization of knowledge within the human mind is subject of interest of cognitive psychologists since generations. A theory, that gained wide attention and acceptance, was first proposed by Quillian [Qui68] proposed a theory about *semantic memory*. It was since then adopted, extended, and verified by many other researchers of the field [Tul72, CL75, CQ95, Mcr04, MHM12]. It was also being referred to by Gaines [WSG92]

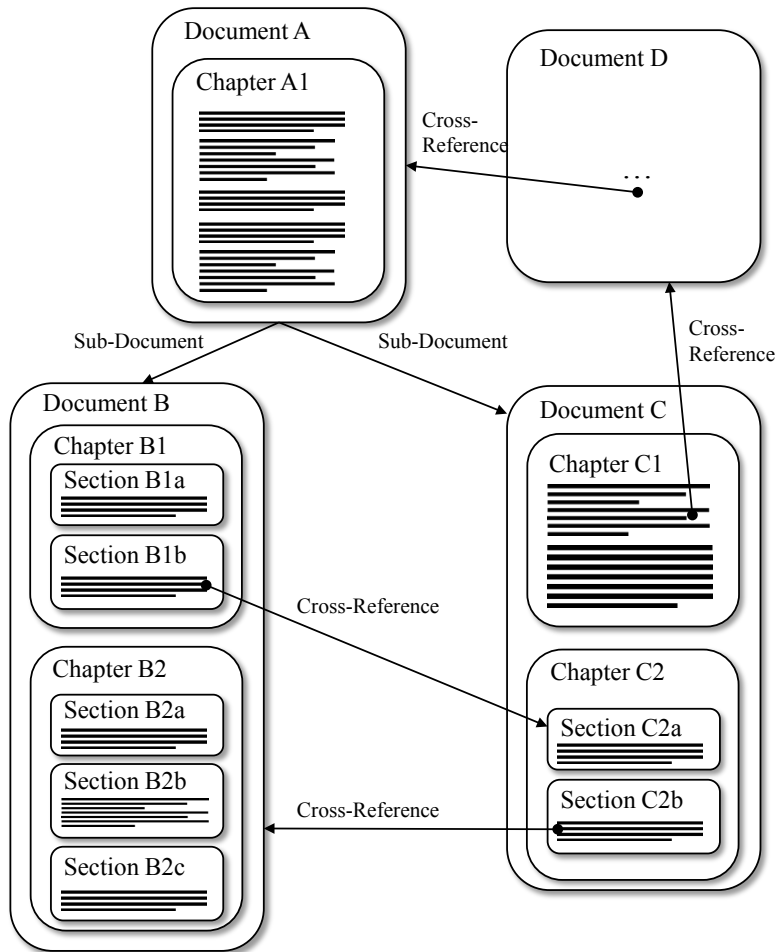


Figure 3.4: The hierarchical document space.

to analyze the cognitive challenges of knowledge acquisition. According to Quillian's theory of *semantic memory* the human knowledge is organized as concepts, where each concept corresponds to particular senses of words or phrases. The concepts are forming nodes in a graph, where properties of a concept are represented as labeled relational links to other concept nodes.

These directed links in addition to the relation label also contain a weight value, which is indicating how essential the corresponding link is for the meaning of the source concept. The full meaning of a concept is the whole network as entered from the concept node (considering multiple levels in depth). Figure 3.6 shows an exemplary fragment of a part of a human's memory modeled according to the graph structure proposed by Collins [CL75]. It shows nodes for common sense concepts, as for instance colors, vehicles, flowers, and fruits. Concepts which are assumed to be related are connected by edges. The edge weights are omitted from this illustration as the actual weight value is not relevant in this context.

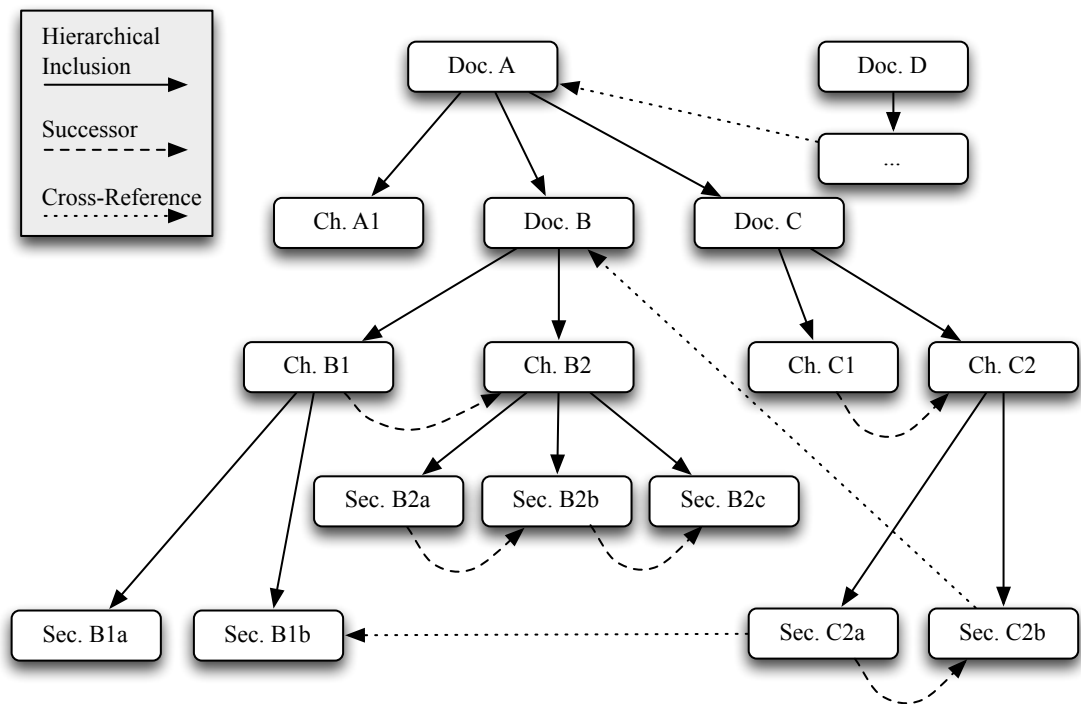


Figure 3.5: The abstract document space graph according to the example shown in Figure 3.4.

Today it is commonly agreed that the semantic memory together with the episodic memory are forming the declarative memory [Tul72], which contains the information that can be accessed in a conscious way. The episodic memory contains the persons autobiographical events, while the semantic memory stores factual and general knowledge about the world. Also, in recent years semantic memory networks have been the bases for the majority of theorizing and empirical investigation [Mcr04, MHM12]. The knowledge, which is relevant in the context of knowledge engineering, can be assumed to be located in the semantic memory, containing the general knowledge about the world.

Modeling the Semantic Memory in the Document Space To support the intuitive understanding of the knowledge, the alignment of the content towards the user’s mental model should be made as easy as possible. Therefore, one should aim to resemble the structure of the semantic memory within the structure of the document space. Then, each content element relates to one or a group of concepts. The document space provides different types of associations to relate content elements to each other, as discussed in Section 3.1.4. By use of these relations the connections between the concepts can be modeled. The relations between content elements in the document space however do not allow for relation labels or weights. This information can easily be incorporated within the text of the content elements if necessary. Modeling the semantic memory fragment shown in Figure 3.6 could for instance be performed as follows:

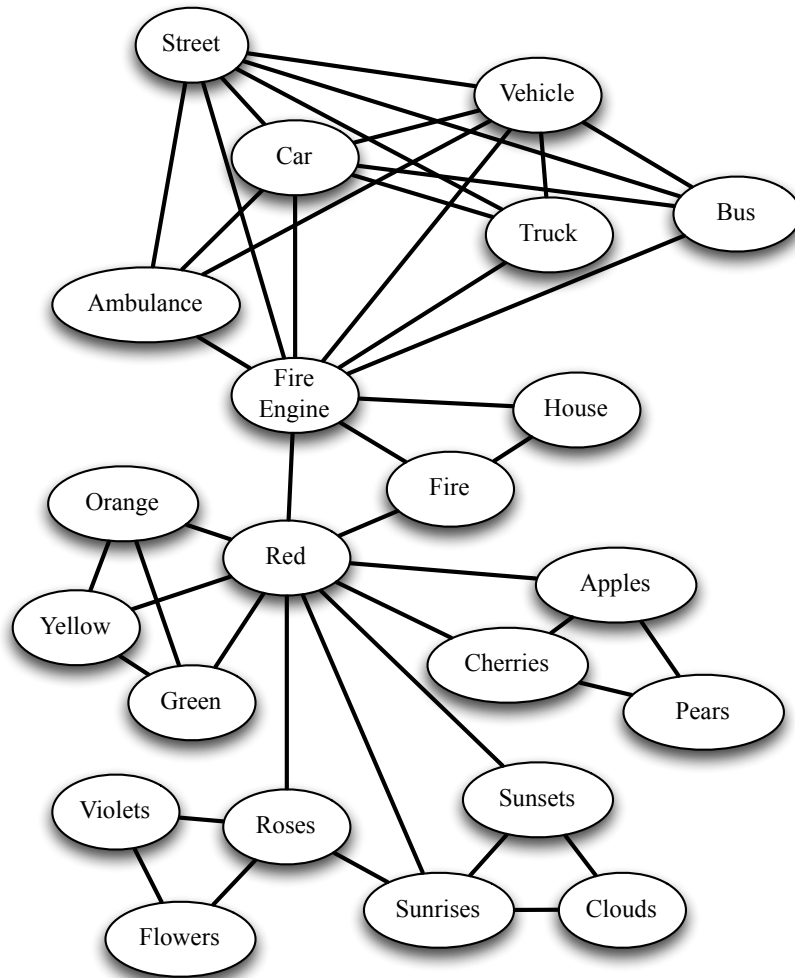


Figure 3.6: An example fragment of the human memory mind according to Collins et al. [CL75].

3 Document-Centered Knowledge Acquisition

There might be a document 'Vehicles' describing in chapters the concepts *Vehicles*, *Bus*, *Truck*, *Car*, *Ambulance*, and *Fire Engine*. Further, there could be one document 'Colors' listing the colors *Orange*, *Red*, *Green*, and *Yellow* as chapters. The chapter about *Red* could have links to the chapter about *Fire Engine* in the document 'Vehicles', and a link to the document 'Fire', and so on and so forth. The document space graph for such document base (c.f. Figure 3.5) then to some extent will replicate the semantic memory graph (c.f. Figure 3.6).

Hence, it should in principle be possible to recreate an (excerpt of a) person's semantic memory graph by a document space graph. In contrast to GUI-based tools, a document-centered authoring environment enables the flexible formation of a structure, which can be adapted to a semantic memory model of a domain.

While the design of this structure is a notable and non-trivial task itself, we claim that its careful creation and ongoing refinement will pay off on the long term, significantly improving understandability of the content. Elicitation of the exact structure of a persons semantic memory is difficult. For this purpose, Gaines [SG93] proposes the use of a method from construct psychology, which is called *repertory grid*, first proposed by Kelly [Kel55]. However, one practical heuristic for approximating the structure is to create documents for the most important concepts of the domain and continuing by intuitive top-down refinement.

An explicit process for evolving the document space towards a comprehensible structure is introduced in Chapter 4 in context of the meta-engineering approach.

3.1.5 The Collaborative Social Process of DCKA

In Section 1.4 mixed-initiative knowledge acquisition has been introduced as a promising strategy for effective collaborative knowledge acquisition. Now, we describe how document-centered knowledge acquisition can support this strategy and the social process it relies on. Figure 3.7 shows the stack already discussed in Section 1.4, now being based on the document-centered knowledge acquisition environment, which is suitable to provide the required preconditions.

3.1.5.1 Preconditions

At first the way, how the document-centered knowledge acquisition environment supports the preconditions shown in Figure 3.7, is discussed.

Low Barriers for Active Participation The level of the technical skills of the participating users typically is rather diverse in knowledge engineering projects, as discussed in the context of the competency dilemma (c.f., Section 1.2.2). To enable many participants for active participation (c.f. 1.2.2.2) it is very important to provide low barriers for contributions. Perception, i.e., reading and browsing of existing content, is posing the first barrier, also constituting an important prerequisite for any other knowledge engineering task. Working through some documents is not very demanding and most people are already used to read and maintain content as electronic documents. Hence, the barriers for active participation are rather low for the document-centered knowledge acquisition approach. If the document space is well-structured and provided with comprehensible domain descriptions, also unexperienced users can obtain an idea about the

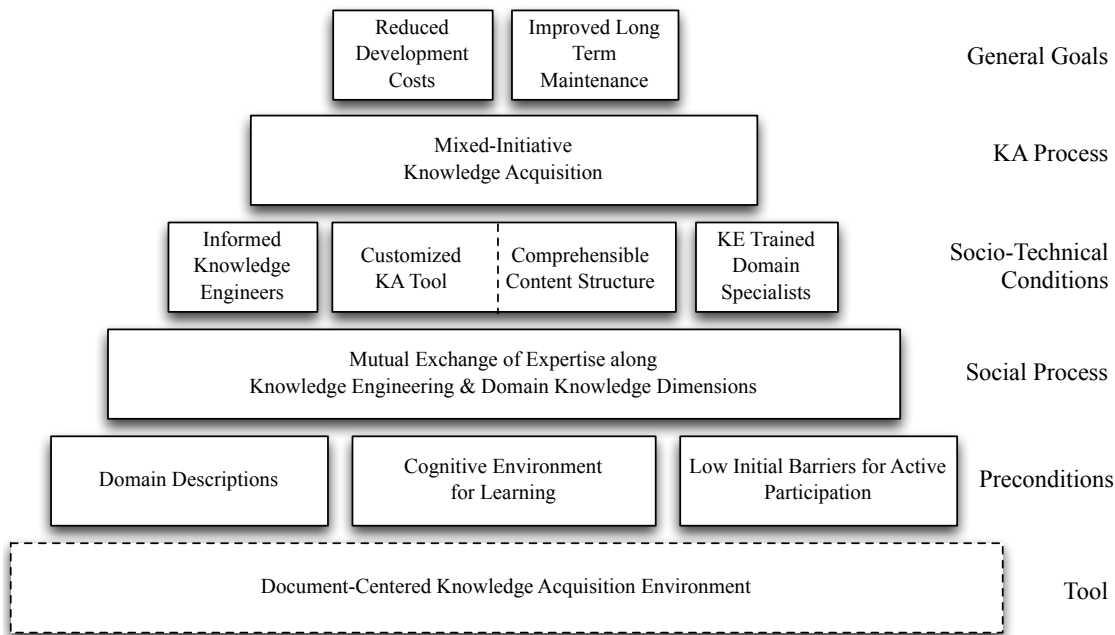


Figure 3.7: A DCKA environment as a basis for mixed-initiative KA and the social process.

contained topics of the domain. For the next step, to actually contribute content, editing text documents is a rather simple authoring paradigm. It allows for simple contributions (e.g., on domain description or organizational information) basically without any training of a new tool or further expertise.

Domain Descriptions The domain description forms the basis for a common understanding of the subject domain. A centralized description for example forces the alignment process of multiple domain specialists, possibly having different opinions about some aspects of the domain. Further, it serves as an important source of information for getting familiar with the problem domain. This is important for the knowledge engineers or any persons coming into the project at a later point. Often previously existing documents and illustrations can be used to create the domain description. The document-centered approach allows to add or create the domain description as documents easily.

Cognitive Environment for Learning For effective collaborative knowledge acquisition a cognitive environment for learning is required, which has to provide an interaction space for communication, studying, and resource sharing [WSG92]. The documents can form the center of such a cognitive environment. The domain descriptions serve as a resource for learning about the subject domain. Formalized parts of the knowledge can serve as knowledge modeling tutorials. Therefore, some modeling examples, albeit toy examples, should be included already at an early stage. Further, the documents can be used for indirect communication, for example

3 Document-Centered Knowledge Acquisition

considering the content category organizational information (c.f. Section 3.1.3). However, for a collaborative cognitive environment of learning with distributed participants additional means for communication are necessary to allow for discussions and coordinate the development process. For this purpose, common communication methods can be used such as email, forums, messenger, chat room, or voice/video chats. Those communication channels in principle are independent of the documents and the authoring environment. Nevertheless, often specific document content elements are serving as reference point for the domain aspect being discussed.

3.1.5.2 Mixed-Initiative Knowledge Acquisition by Incremental Formalization

The purpose of mixed-initiative knowledge acquisition is to allow every participant to contribute according to his specific expertise. *Incremental formalization* is one attempt to overcome the competency dilemma in knowledge engineering and is well-suited to support mixed-initiative knowledge acquisition in a DCKA environment.

The process of incremental formalization is based on the domain description (c.f. Section 3.1.3), illustrating the domain knowledge in an informal way using text and figures. From the informal description the formalization task is driven by multiple small steps. The first step is the identification of the knowledge fragments within the description, that need to be formalized to form a part of the intended executable knowledge base. These fragments then can be reformulated, also describing their intended role for the reasoning within the knowledge base. This knowledge base oriented description still can be created in natural language. After that, a first try of transforming this knowledge base oriented description into the knowledge markup language can be made. This initial, potentially erroneous or incomplete modeling can then be refined gradually in multiple additional steps.

The advantage of incremental formalization is the decomposition of the overall knowledge formalization task into distinct sub-steps. Each sub-step shows to be significantly easier than performing the overall task in one step. Especially, considering the heterogeneous distribution of competencies (c.f., competency dilemma 1.2.2) the step-wise transformation from informal to formal knowledge provides large advantages. First, the smaller steps are easier to comprehend one by one. Further, the distinct steps can involve different persons according to the required expertise. For example, one person, e.g., a domain expert, can start with the initial steps. If the person notices to have problems with further steps, e.g., with using the markup language, the current intermediate state is persisted. The person can then inform another participant with more modeling experience to join the formalization process at this point. The modeling expert continues the formalization task in coordination with the former person. This mixed-initiative workflow can either be performed in real-time interaction or in an asynchronous way by making small modifications and leaving comments at the corresponding content elements.

We claim, that the weakness of many knowledge acquisition tools is, that any knowledge formalization task has to be performed entirely in one single working step. Those tools then are very hard to use by single persons, as a formalization task always requires the full range of expertise on both dimensions, domain knowledge and knowledge engineering/modeling. Hence, the ability for doing incremental formalization, decomposing the formalization task into distinct steps, is one of the most important advantages of document-centered knowledge acquisition with markup languages.

Several socio-technical conditions can support and improve this knowledge acquisition process (c.f. Figure 3.7). The overall knowledge acquisition approach aims to exchange and improve expertise of the participants over time, which is discussed in the following section. Additionally, the tool and content structure can be improved and adapted to the project. This adaptation of tool and content is introduced in detail in Chapter 4. Considering these aspects, also the nature of the mixed-initiative workflow will evolve during the progress of a knowledge engineering project. While requiring very close and frequent interaction of different participants at the beginning, with improved competency more formalization steps can be performed by single persons at a later stage of the project. In that way, also the long term maintenance of the knowledge system is simplified.

3.1.5.3 Mutual Exchange of Expertise

In Section 1.2.1 we stated the need for a cognitive environment of learning to support the exchange of expertise. The exchange is required along two dimensions as depicted in Figure 1.1: The knowledge engineers need to get familiar with the domain, while the domain specialists need to acquire some basic understanding of knowledge engineering principles. Before discussing these processes, some aspects of learning theory, considering formal and informal learning, are introduced.

Formal and Informal Learning: Learning theory distinguishes two basic types of learning, *formal learning* and *informal learning*. Ainsworth and Eaton [AE10] describe formal learning as intentional, organized and structured activity. It is characterized by a formal educational program including prepared and scheduled lessons, grades, and certificates.

Informal learning, first discussed by Knowles [Kno50], in contrast is never organized but can be considered as spontaneous and experimental [AE10]. In practice, informal learning often is performed 'on-demand', that is some skill or knowledge is acquired when needed for the task currently at hand.

An additional intermediate form is the *non-formal learning* [AE10]. It is intentional and loosely organized but without formal credits, such as grades or certificates.

Both, formal and informal learning, play a fundamental role in human skill acquisition today. While formal learning still provides basic education for example in schools, within recent years the focus of attention in research of learning theory shifted towards informal learning. The disadvantage of formal learning is that often one cannot determine a priori, which expertise a person will need in future. Therefore, it is possible that a person acquires skills, which will never be employed. Another problem is the aspect of time. Skills acquired by formal learning will be forgotten to some extent over time, if they are not applied occasionally. This so-called 'curve of forgetting' [Fin13], which is assumed to be a negative exponential function, was first analyzed by Ebbinghaus [Ebb85]. Depending on the detailed personal parameters of the person's memory capabilities, it verifiably leads to a loss of about the half of the information within the first week. That problems is not as relevant for informal learning, happening on demand. Coming up during the last years, new web technologies and ubiquitous connectivity at the working place allow for improved support of informal learning activities.

3 Document-Centered Knowledge Acquisition

For the exchange of expertise in document-centered knowledge acquisition also both forms of learning are required. Several basics need to be learned in a formal (or non-formal) way at the beginning of a knowledge engineering project. Then the focus switches to informal learning for the acquisition of additional special skills on demand.

Exchange of Domain Expertise The exchange of domain expertise is conducted (to some extent) indirectly using the document-centered knowledge acquisition environment. A detailed description of the domain is created within the document space. The illustrative character of these descriptions should support the knowledge engineers to obtain basic insights about the domain. It should be evolved cooperatively towards a version, which is understandable for all project participants. The elements of this content category taken all together can be considered to form a kind of text book, explaining the domain at a scope which is relevant for the desired application task. In that way, new participants coming to the project can easily work into the subject domain (and the knowledge base subsequently). For the development of the domain description the domain specialists necessarily play a prominent role. The formation of the corresponding document corpus takes some effort but does not require advanced knowledge engineering skills. At the beginning, the basic skills of document editing and structuring should be trained (if necessary) as an act of formal or at least intentional learning, possibly in a workshop-like style using the document-centered authoring environment.

Many knowledge acquisition projects include a community of multiple specialists as knowledge sources. One reason therefore might be that in complex domains no single person can be expert at any subtopic. However, the involvement of multiple domain specialists requires coordination as different persons also have differing mental models of the domain, considering their semantic memory.

Convergence of Knowledge As the documents are forming a shared space in this collaborative scenario, a process of alignment is required to lead to convergence of the document space. This alignment process however is inherent to any collaborative design task, not only to knowledge engineering.

The process of collaborative knowledge building has been discussed comprehensively by Cress et al. [CK07] in the context of wikis. The theory can easily be generalized, assuming each wiki page constituting one document. Cress et al. distinguish *internalization*, where a user consumes new knowledge from the wiki content, and *externalization*, where the user inserts knowledge units into the wiki. Either activity can be considered as an extension of a knowledge space, on the one hand the users personal knowledge space and the wiki space (or document space) on the other hand. Figure 3.8 illustrates internalization and externalization in a wiki space with two persons *A* and *B*.

An extension as a simple insertion of the knowledge without modifying the existing structure of the knowledge space is called *assimilation*. An extension in contrast, which is changing the existing structure is called *accommodation*. This distinction applies to internalization and externalization respectively. According to Cress et al., when working with the wiki, the user is matching his own individual knowledge space to the wiki knowledge space. At that point

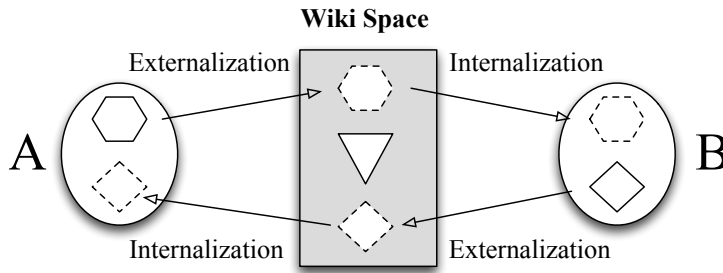


Figure 3.8: Externalization and Internalization in a wiki space according to Cress et al. [CK07].

Piaget's model of equilibration of cognitive structures [Pia77] can be applied to describe the alignment process between the different participants and the content. If a person feels the wiki knowledge space being congruent with his/her own knowledge space, no need for accommodation nor assimilation emerges, neither internally nor externally. Otherwise, if people realize some difference, they can satisfy the need for equilibration by internal or external assimilation or accommodation [CK07]. Acts of accommodation can be assumed to change the document space towards the semantic memory graph structure of the respective person. In that way, by continuous equilibration a document space, being congruent to and consistent with most participants personal knowledge space, emerges. This also implies changes within personal knowledge spaces of these participants. An empirical example for the equilibration process is discussed in detail by Cress et al. [CK08].

During that process also disagreeing opinions between participants about some aspect of the domain might occur. The (repeated) reversion of changes in the documents can be an indicator for these kind of disagreement. In Wikipedia this pattern, called 'negotiation' or 'edit war' [VWD04], is frequently observed, often leading to ongoing disturbance. However, in the closed and cooperative community of a knowledge engineering scenario as envisioned in this work, these kind of disagreements can be resolved. The document-centered approach allows to capture an alternative theory about some aspect easily. The result and argumentation of the resolution can be inserted as organizational content serving as justification for the consensus. It is desirable to rule out these kind of disagreements before the corresponding knowledge is formalized to the executable knowledge base. Later the resolution becomes more complicated as it then also needs to be considered at the symbol level.

This proceeding describes the process of the creating a unified perspective on the domain, explicitly represented by the documents. While not yet directly affecting the compiled knowledge base content, this alignment process is an important step in collaborative knowledge engineering. As the knowledge base itself also has to be an object of mutual consent, the alignment on the document content level is an important intermediate step. It is necessary, to provide a common understanding of the domain, which then will be extended to a common understanding of the knowledge base and the associated knowledge modeling tasks.

The entire domain description and its document space structure cannot be once and for all created in one working effort. It requires ongoing improvement, extension, and refinement. A process for the evolution of the document space is presented in Chapter 4.

Exchange of Knowledge Engineering Expertise The other dimension of learning considers knowledge engineering skills. This includes the task of knowledge modeling, which is usually challenging for domain experts without computer science background.

It is not intended to broadly train the domain specialists for knowledge engineering skills a priori. The initial training goal is to make the experts familiar with the idea, that the domain knowledge is formalized to an executable knowledge base using markup languages. This expertise can be provided by discussing small examples, using the actual subject domain or a related one. While these basics should be learned in a formal or at least intentional way, further skills should be developed by informal learning. Knowledge formalization is performed in a mixed-initiative workflow based on close cooperation of domain specialists and knowledge engineers. At an early stage of the project, the knowledge engineers perform the knowledge modeling task, including markup expressions and modeling rationale. A major training goal for domain specialists at this point is to understand the knowledge modeling.

Every relevant sub-topic of the domain is represented in the document space. When it is completely treated by domain description, markup expressions, and modeling rationale, this knowledge fragment can be considered as a modeling example. In that way, the example-based learning approach can be pursued. Learning from worked-out example is known to be a very effective learning method [ADRW00] and also very convenient for informal learning. This holds even more when the learner is familiar with the domain of the examples. Therefore, existing worked-out topics should boost the learning process along the knowledge engineering dimension. With this process of informal learning the domain specialists improve their knowledge modeling skills over time. One can not intend or expect that a domain specialist will become a knowledge modeling expert this way. However, real world knowledge bases usually are made up of different kind of entities, featuring different complexity and modeling methods. We claim that for this aspect also the *pareto principle*, also known as *80/20 rule* [LHB03], applies. In this context that means that 80% of the knowledge base content has comparably low complexity. In consequence, with low modeling skills (e.g., 20% of knowledge engineering expert) one can perform 80% of the knowledge modeling tasks. While mixed-initiative knowledge acquisition encourages direct knowledge acquisition, it does not aim to assign *all* modeling tasks to domain specialists. Therefore, the informal learning process does not intend to educate general knowledge engineering, but only the required skills to understand and possibly maintain a significant part of the current knowledge base at hand. Assuming the pareto principle holds in this context, this strategy can be quite effective.

3.1.6 Authoring of Multimodal Knowledge

In Section 1.3.3 knowledge formalization was introduced as an act of communication from a human user agent towards the knowledge-based system agent by environment manipulation. In this section, the details about this knowledge transfer from the human specialists to the knowledge base is discussed in more detail. At first, a suitable human-computer interaction model is introduced. Subsequently, the different levels of knowledge communications are discussed. Then the resulting requirements and possibilities for the authoring of multimodal knowledge are presented.

3.1.6.1 A Human-Computer Interaction Model of Interactive Alignment

Heim et al. describe the problems occurring when users deal with formal knowledge in the context of the semantic web [HSE11]. They introduce a general model for human-computer interaction in that context. It is derived from a human-to-human communication model called *interactive alignment* [PG04]. Heim et al. argue that semantic web applications will become more usable if they try to simulate this kind of communication model being natural to humans. That assumption can also be transferred to the development of knowledge-based systems as in either case the challenge is interaction between a human and a computer system, possessing formalized knowledge about the domain of interest. The key idea is that communication is not a single step activity, but a multi stage process of interactive alignment. After an initial message from the speaker, the listener provides feedback information about how she or he perceived the information of that message, i.e., how it has been aligned to his internal model. For this purpose, the feedback is enriched with context information illustrating the interpretation. Often no unambiguous interpretation is possible. Then multiple possible interpretations are sent back to the speaker with the demand for distinguishment. The speaker then tries to clarify the interpretation that complies to his or her own mental model. In that way, in a multi-stage message exchange process it is made sure that a shared interpretation of the discussed information is achieved. Problems not only can arise on the semantic interpretation level, but also may occur already on the syntactical analysis level. Also in that case feedback has to be given accordingly. While in human-to-human communication an alignment between mental models of different persons is formed, in the case of human-to-computer communication an alignment between the mental model of the user on the one hand and the formal model of the knowledge base on the other hand is endeavored. As computers do not understand natural language, the speech act has to be performed using another interaction method, which is less natural to the human. Therefore, it is even more necessary to have a process asserting the alignment of the respective semantic interpretations. This interaction model is helpful for the design of a knowledge acquisition interface as the alignment of the mental model of experts and the semantics of a formal knowledge representation is a key challenge of knowledge engineering. If a piece of knowledge is entered into a knowledge acquisition tool, it should provide verbose and instant feedback about its semantic interpretation. Then the user can verify whether that complies to his intention.

3.1.6.2 The Levels of Knowledge Communication

The communication process of each piece of knowledge passes through multiple stages before it can successfully be accessed by the knowledge-based system agent and actually play its role in the desired problem-solving process. In principle, on each of these levels the process of communicating a piece of knowledge can fail. We introduce the following the four levels of formal knowledge communication, also discussing detection of problems on each level:

1. **Syntax Level:** This level describes the first syntactical analysis of the perceived 'sensor data'. Problems on this level can be detected independently of all the existing knowledge only referring to the respective input data at hand.
2. **Terminology Level:** This level describes the matching of the employed term references

3 Document-Centered Knowledge Acquisition

to the known terminology of domain concepts. Pieces of knowledge can be considered as propositions about concepts of the domain of discourse. For the coherence of the knowledge base it is important that the system can unify concepts that are meant to be the identical concepts across different propositions.

3. **Knowledge Representation Level:** This level describes the compliance and consistency with respect to reasoning of the piece of knowledge with the knowledge already present in the knowledge base. The characteristic of this category is that its verification requires to know about the symbol level knowledge representation used and its semantics. Detection of knowledge representation level problems can be rather hard, often requiring a serious amount of computation efforts, depending on the inference mechanism and knowledge base size. The violation of consistency is a common example of a failure on this level, being relevant for many established symbol level knowledge representation formalisms. For more details, we refer to literature (c.f., [BKS07]).
4. **Domain Level:** This level describes the actual contribution of the piece of knowledge to the correct solution of the problem-solving task. The detection of domain level problems in the end is particular hard. In the first instance, those problems cannot be detected automatically by the system at all, as basically wrong knowledge had been told. This fact cannot be realized by the knowledge-based system agent without help. For the detection of these problems the corresponding (correct) domain knowledge in some form is required. Typically, these problems are detected by validation efforts, employing extensive manual testing or inspection by experts or test cases, that also have to be specified by domain experts. While for the other categories automated and reusable methods can be applied for detection, the validation task on domain level can only be achieved by bringing in additional domain competency.

For the fourth level, one can argue whether it indeed represents a case of faulty communication. Basically, knowledge has not been communicated wrong, but wrong knowledge has been communicated (correctly). Still, assuming the knowledge is present correctly at the information source, that is the speaker, the error must have arisen somewhere on the communication path. However, for the overall goal of efficient knowledge engineering the issue is of high importance in either case.

These stages are valid in general for all knowledge acquisition tools. However, different kinds of tools have different methods to deal with the communication problems on the different levels. Tools based on graphical user interfaces can be designed to prevent errors at the syntax and usually also on the terminology level. The document-centered knowledge acquisition approach needs to deal with these levels in a different way.

3.1.6.3 Problems in Markup-based Knowledge Authoring

In Section 3.1.6.2 we explained that knowledge being communicated between agents has to pass multiple levels. On each level communication problems can arise and need to be detected. It shows that the task of detecting these problems increases in complexity on each subsequent

level, while being a prerequisite for an interactive alignment process. Therefore, it is necessary to analyze the flavor of the problems that might arise at the different stages when employing document-centered knowledge acquisition. The knowledge source text, formulated using knowledge markup languages, needs to be compiled towards an executable version which is the application of τ (c.f. Section 3.1.2). This is unproblematic assuming the correct use of \mathcal{M} . However, in a scenario where human users, having more or less experience with the markup language, edit the content, an incorrect use has to be expected. In the following, we discuss how the different categories of problems take effect and how they can be treated using interactive alignment:

- **Syntax Level:** These kind of errors occur when some statement does not comply to the formal syntax of the markup language. Helpful error messages should be provided, possibly providing links to examples using the syntax correctly.
- **Terminology Level:** If unknown concept identifiers are used or a concept of inappropriate type with respect to the context, corresponding error messages can be generated. Further, known concept identifiers can be proposed according to similarity measures.
- **Knowledge Representation & Domain Level:** From the knowledge representation stage on, the detection task as such is independent of the knowledge acquisition tool, i.e., the markup language in this context. Therefore, only the way of presenting the feedback to the user can be discussed.

3.1.6.4 Support for Knowledge Authoring

Editing of documents is a pure and simple authoring paradigm. It is well suited for providing low barriers for participation. However, extending the document-based authoring paradigm by possibility to create formal knowledge bases by the use of markup languages is a challenging task. While the intuitive authoring paradigm is retained for informal content parts, additional help for creating markup expressions should be provided.

Authoring by Interactive Alignment Interactive alignment is a promising communication strategy also employed in human communication. The immediate feedback allows to detect and resolve misunderstands or ambiguities quickly. Therefore, knowledge authoring in document-centered knowledge acquisition also should follow this strategy. For each of the levels of knowledge communication, discussed in Section 3.1.6.2, a check for correctness and verbose feedback to the author should be provided. For the first three levels the feedback usually consists of error messages if problems have been detected. These errors, which can be located at a particular position within markup expressions, the message should prominently highlighted at this point when the document is displayed. Otherwise, especially considering the knowledge representation level, the error messages need to be presented to the user in a way easily recognizable. Interactive alignment on the domain knowledge level implies that the executable knowledge base is instantly executed with a set of predefined input parameters. If possible considering the required runtime, the result is instantly presented to the user after modifications. In that way,

3 Document-Centered Knowledge Acquisition

the actual operational semantics of the created knowledge can be compared to the intended semantics in a comfortable way. It can also be combined with automated testing using continuous integration².

After the detection and presentation of the problems, also possible solutions can be proposed to the user. In the simplest case, for misspelled concept identifier correction propositions based on edit distance on the known concept terminology are given. This strategy is also heavily applied in software engineering (e.g., [MBH⁺12]) known as IDE recommendations or simply *quick fix* options.

Autocompletion Autocompletion support is established in IDEs for software development (e.g., the Eclipse Java IDE) and has been subject of research in the area of software engineering for years (c.f. [HR04, HWM09, KM06]). These techniques are today employed in almost any kind of text input field. Especially in search engines the words, which are very likely in the respective context, are proposed to the users [Bas06] for autocompletion. These techniques reduce the typing workload for the users significantly. They are also well-suited to be employed for editing knowledge markup languages.

Special Editors We advocate the use of knowledge markup languages for the representation and authoring of knowledge base content as an alternative to the use of graphical user interfaces and forms. However, the advantages of document-centered knowledge acquisition with markups are not due to the fact that the knowledge is entered by typing formal syntax instead of using GUI elements. Typing formal syntax can at some point be considered as awkward and cumbersome. However, the advantage of markup languages is not rooted in the way it is edited, but in the way it is perceived. Its strength is its readability and the way it can be integrated with the other document content, e.g., the domain description. Further, various markup languages can be designed for specific purpose, see Chapter 4 for more details. Hence, the use of markup languages supports well the understandability of the knowledge base. We claim, that much more time is spent reading and understanding knowledge than actually editing it. Therefore, readability is the prior goal. Nevertheless, the editing of the markup expressions can be supported by helpful authoring support. One simple example is code completion as known from programming environments in software development. Further, the editing of markup expressions can also be augmented by graphical interaction elements, such as drop-down menus or drag-and-drop functions (e.g., for terminology objects). There is a wide range of possibilities to support the authoring of particular markup languages with special editors. We recommend the introduction and use of these kind of special editors for specific markups. It however should not affect the readability of the content or somehow distract the reader. That is, interaction elements should not be prominently visible in the basic view of the content. If this is desired or necessary for efficient editing, we recommend the use of external editors. Then only one link or button is required to open the respective markup expression in the specific external editor. After editing, the editor is closed and the modified markup expression is inserted at the corresponding location in the document, replacing the former expression. With the introduction of special editors, the user is able to choose between the normal text-based editor and the one or multiple special editors according to his preferences.

²<http://martinfowler.com/articles/continuousIntegration.html>

3.2 The Advantages and Challenges of DCKA

We already pointed out that the interaction paradigm most widely used for knowledge acquisition is the use of graphical user interface. DCKA does have quite different characteristics when compared to this authoring paradigm. Not all of them are obvious at first glance but show their relevance in practice. In this section, the advantages and challenges of the document-centered method are compared to the GUI-based approach. Then, the requirements of an efficient document-centered authoring environment are discussed.

3.2.1 Advantages

The following aspects make the DCKA approach attractive for direct participation of domain specialists within the knowledge acquisition activities.

3.2.1.1 Low Barriers for Basic Contributions

The level of the technical skills of the participating users typically is rather diverse in knowledge engineering projects. In Section 1.4 the importance of providing low barrier contributions especially for novice users was discussed. Perception, i.e., reading of browsing of existing content, is posing the first barrier also constituting an important prerequisite for any other knowledge engineering task of active participation. Many GUI-based tools, which allow for the creation of complex knowledge bases, require considerable need of training before use. This also holds for editing tasks that actually are very simple. In any case, the user has to get used to a completely new tool environment. Working through some documents in contrast is not very challenging and most people are already used to read and maintain content as electronic documents. Therefore, for tasks of low technical complexity, such as proof reading, no training is necessary at all. Hence, the document-centered approach provides quite low barriers for active participation (c.f. 1.2.2.2). For the next step, to actually contribute to the content, editing text documents is a rather simple editing paradigm. Being used to these kind of simple contributions without difficulty, contributors often feel encouraged to explore more complex tasks.

3.2.1.2 Incremental Formalization

The creation of a knowledge base entity with a GUI-based tool usually requires to perform the entire formalization task in one step. This involves having the respective domain knowledge at hand, do the knowledge modeling, and insert the model correctly using the graphical user interface. This complex task requires expertise in both dimensions, domain knowledge and knowledge engineering skill. The strategy of *incremental formalization*, already introduced in Section 3.1.5.2, proposes to decompose the complex task into multiple steps. That is difficult, when using a graphical user interface, but simple with the document-centered approach. It starts with the insertion of informal content describing the domain, such as text and figures. At first, it serves as a startup for the formalization process and later it forms the documentation and context of the knowledge base components. The incremental formalization workflow proceeds with the identification of those content parts, that need to be formalized to form the intended executable knowledge base. Then a tentative formalization is made, that is, the selected content

3 Document-Centered Knowledge Acquisition

is transformed towards the knowledge markup language. This formalization can be gradually refined. These distinct steps require different degrees of expertise in the domain, in knowledge engineering, and usage of the respective acquisition tool. It can involve different persons on different steps in a mixed initiative formalization process, simplifying the accomplishment of the formalization task. Hence, the incremental formalization workflow allows the participants to be able to contribute according to their respective expertise.

3.2.1.3 Freedom of Structuring

The structure of the content in the documents, i.e., the document space (c.f. Section 3.1.4), can freely be designed. Within one document the order of the content elements is free to the user and can be adapted to his mental model of the domain. Domain description, modeling rationale, and organizational information can be inserted at any place and in any style. Further, the documents can freely be interlinked with others making interrelations of content parts explicit and improve navigation. This freedom of structuring allows the documents to evolve a memorable and comprehensible structure, becoming familiar to the authors. It is one of the most powerful advantages of document-centered knowledge acquisition. These means of structuring the content are typically not present or strongly limited when using GUI-based tools.

While a good document space structure provides large benefits on comprehensibility of the knowledge base, it also comes with a burden. The good structure needs to be maintained, that is, new contributions have to comply to this 'good' structure, posing a notable responsibility on the contributor. A detailed discussion of the possibilities to maintain and improve the structure of the document space is given in Chapter 4 in context of meta-engineering for DCKA. While the value of this freedom allowing a comprehensible document space structure can hardly be overestimated, it however also contains a considerable challenge and therefore might also have been mentioned in the subsequent section discussing the challenges.

3.2.1.4 Example-based Authoring

"...learning from worked examples is of major importance in initial stages of cognitive skills acquisition."

Atkinson et. al. [ADRW00]

Assuming the case that a user is working on the knowledge base, finding a particular knowledge entity, and wants to create a new one with similar characteristics. In GUI-based tools it is often not obvious how the entity and its characteristics have been created by the given interface. Hence, for novice users even the creation of a similar entity can be a considerable challenge in this scenario. That problem can not occur in document-centered knowledge acquisition. If a knowledge base entity is found, the user can simply copy/paste&modify the respective markup expression. Therefore, no peculiarities of the tool have to be known.

Contributing to the compiled knowledge base requires knowledge formalization skills, i.e., use the markup languages and the knowledge modeling skill. However, acquisition of both skills can benefit from example-based learning. On the syntax level, the use of the markup language can (literally) be copied from the examples by copy/paste&modify. Often only the

domain object names need to be exchanged to create new valid knowledge. While the attraction of copy&paste in software engineering is known and proscribed for producing redundant code, it is unproblematic in this context of declarative code, but allows a smooth introduction to the formalization task. The modeling skill being much more challenging, however can also be improved on the basis of simple toy example knowledge bases provided within the document corpus. To support this method of learning, modeling examples can and should be included in the document-centered authoring environment. Learning from worked-out examples is typically a very effective way of cognitive skill acquisition, especially for novices [ADRW00]. At later stages of the project, already existing knowledge base parts can serve as syntax examples and modeling examples.

3.2.1.5 Quality Management

Quality management is a crucial and challenging task in the knowledge base development and maintenance process. Also in the domain of software engineering the aspect of quality management has been studied for many years. A very successful set of practices that was established within the last decade, especially in the context of agile software development [BA04, Mar09], is called *Continuous Integration* (CI). According to Fowler³ the main requirements for CI are the use of a code repository, automated building, automated tests, frequent and timely integration of changes, and easy access to the latest builds. The main benefits of CI are, that always a valid version of the system is available for deployment in a productive setting. Further, problems emerging by changes are recognized very early, making debugging easy and reducing risks of complex change operations. It continuously guarantees quality and transparency. The knowledge base authoring approach described in this paper allows for straight forward application of CI. Also GUI-based tools allow for the application of CI. There, an extra strategy for defining versions of the knowledge base needs to be established and introduced to the user. In the document-centered case, in contrast CI can be applied in a very natural way putting the documents into a centralized version control system.

3.2.2 Challenges

Above we discussed the advantages of the document-centered knowledge acquisition approach. These considerable advantages however come at the cost of additional challenges arising when using this method. In the following, these issues are briefly described together with possibilities to overcome them.

3.2.2.1 Authoring Assistance

One important aspect of document authoring is that any input inserted by a user will be accepted, that is stored at least. Correct communication of formal knowledge is a challenging task as discussed in Section 3.1.6.2. Graphical user interfaces usually are designed in a way, that prevent the user from making errors on the syntactical and the terminology level. With respect to this, the situation in DCKA using markup languages is more challenging. The use of

³<http://martinfowler.com/articles/continuousIntegration.html>

3 Document-Centered Knowledge Acquisition

a markup language for this task can be unfamiliar to the user and requires reasonable authoring assistance. Especially for unexperienced users it is important to provide good support during and after the typing process. The interaction model based on interactive alignment discussed in Section 3.1.6.1 provides a guideline principle for creating authoring assistance. In general, it is important to provide as much feedback as possible about how the user input is perceived by the system. The challenges and methods on each level of knowledge communication have been discussed in Section 3.1.6.4. Visual feedback, reflecting the systems *understanding* of the content is absolutely essential. One can state that several techniques for user assistance are existing in software engineering, e.g., autocompletion, are suitable to be applied in the document-centered knowledge acquisition context. The well-conceived application of these techniques is an important prerequisite to allow efficient participation for a wide range of persons on the knowledge acquisition process.

3.2.2.2 Content Refactoring

Refactoring is defined as changing the structure of something without changing its semantics. Refactoring has been widely discussed in software engineering [Fow99] as well as in knowledge engineering (e.g., [GT97, BSP04]). In agile knowledge engineering the need for this kind of restructuring of the knowledge base is a very typical phenomenon. When using GUI-based tools refactoring in general is easier, as there is no informal content that is organized in a document structure, which needs to be maintained. In document-centered knowledge acquisition, we need to distinguish two distinct levels of refactoring, the document level and the symbol level refactoring.

Symbol Level Refactoring: This level of refactoring resembles the refactoring operations as well known in knowledge engineering, independently from the acquisition technique. It implies to restructure the symbol level knowledge base without changing its semantics. These kind of refactoring methods usually strongly depend on the employed symbol level knowledge representation formalism and its semantics. For example, the rearrangement of proposition logics formulas (e.g., by applying De Morgan's law) is semantic retaining refactoring. Refactoring methods for particular symbol level knowledge representations can be found in literature (e.g., [GT97, BSP04]) and are not further discussed in this context. The characteristic difference to the document level of refactoring is that symbol level refactorings *do* change the symbol level knowledge base (while not its semantics).

Document Level Refactoring: Another level of refactoring arises due to the document space structure that is given in document-centered knowledge acquisition. A knowledge base KB is created by a compilation step applying τ on the current version of the document base DB : $\tau(DB) = KB$; Any document base modification $m(DB) = DB^*$ has to be considered as a document level refactoring operation if:

$$\tau(DB) = \tau(DB^*) = KB,$$

While refactoring on the symbol level changes document content (necessarily) *and* the symbol level knowledge base, the document level refactoring only modifies the document content,

leaving the symbol level knowledge base identical (and therefore its semantics). Although the mapping function τ depends on the markup language provided, such modifications in general exist. One of the positive aspects of document-centered knowledge acquisition is the fact, that informal knowledge can be entered in an unconstrained way, e.g., domain description. As these parts are not considered by τ , the requirement above holds when the informal knowledge is changed. However, there are also refactoring operations that involve the markup statements that are compiled by τ . As a knowledge base is considered as a set of (unordered) items, ordering of the knowledge slices in the documents can freely be chosen without modifying the knowledge base. Indentation or comments within markup expressions are further examples, which formally belong to this category. Often, the markup language is composed of multiple different sub-languages in a way making τ to be non-injective. That allows to define the same knowledge in different ways. For example, rules can be defined in IF-THEN text markup or in tabular style resulting in the same rules in the compiled knowledge base. The transformation of rules from one to the other markup then is a document level refactoring operation. While refactoring operations always can be carried out manually by editing the documents, it is sometimes convenient to provide mechanisms to perform the document changes automatically. These kind of mechanisms are discussed in more detail in Chapter 5.

In general one can say, that refactoring is a more important issue in document-centered knowledge acquisition than when working with GUI-based tools.

3.2.2.3 Navigation and Search

The possibility of free content organization also requires the need for solutions for navigation and search. Within the knowledge engineering process many different kinds of tasks have to be performed. Most of these tasks require the user at first to find the location of action, which is the document and the position within that document being subject of the task. To find the location of action, different strategies are possible:

- **Navigation:** The document space should be organized in a hierarchical structure, with each document covering a coherent subset of the domain also provided with a meaningful name. If the hierarchical structure is available as hyper-links the user can easily navigate through the domain knowledge to find the location treating the demanded aspect. However, this hierarchical structure, including the content partition into (named) documents, has to be created carefully from the beginning on. Still, it might occur during project progress that the structure is not appropriate anymore. Then, refactoring to a more suitable structure is inevitable to allow for efficient navigation. Beyond the hierarchical structure, cross-linking of related content elements should be maintained.
- **(Semantic) Search:** A simple and often successful mechanism to find content in a set of documents is full text search. Still, full text search has its limitations due to the ambiguity of natural language content. However, in many cases the (already existing part of the) knowledge base can be abused to support a more efficient search method. A simple way to find interesting locations about a specific domain concept is to generate a list of the positions, where the concept occurs in markup expressions using the compiler information. But also the actual relations in the knowledge base can be exploited to allow for a kind

3 Document-Centered Knowledge Acquisition

of semantic search. For example, a medical knowledge base can be used to search for locations, where some heart disease and some antihypertensive medicament is mentioned, presumed it contains the knowledge about what concepts are heart diseases and which are antihypertensive medicaments. Then all content parts containing such concept combinations can be suggested. This corresponds to the so called query expansion method, which is a wide spread semantic search technique [Voo94].

- **Bookmarks:** By the use of bookmarks, the responsibility for access to the contents of interest is left to the user. Each user is able to maintain his own library of bookmarks, each linking to a precise location within the document base. Today, many bookmark management systems are available providing categorization and keyword mechanisms. For the case that a web-based DCKA environment is used, most browsers provide an integrated system for bookmark management.

3.2.2.4 Redundancy Detection

The freedom provided by DCKA makes it possible that multiple markup expressions, which are forming the same knowledge base entities, are entered into the documents at different locations. While this redundancy might not be a problem for the knowledge repository and the reasoning engine, it poses problems for the maintenance of the knowledge base. Let e_1 and e_2 be markup expression where $\tau(e_1) = \tau(e_2) = r \in \mathcal{K}$. As $\tau(\{e_1, e_2\}) = \tau(\{e_1\}) = \tau(\{e_2\})$, the user will be confused when performing the respective document modification without that the knowledge base content or behavior changes. To prevent this kind of confusion, redundancy detection should be provided. For injective markup languages, redundancy detection can be performed on the expression level, that is comparing e_1 and e_2 . For non-injective markups (the common case in practice) redundancy detection has to be performed on the knowledge repository level, that is checking $\tau(e_1) = \tau(e_2)$. If for some expression pair this condition holds, a corresponding report should be provided to the user. In that way, he is able to resolve the redundancy by identifying and deleting unnecessary statements. However, not every knowledge repository implementation is able to detect such redundancies off-the-shelf.

3.2.2.5 Debugging

Debugging of knowledge bases is a necessary task in practice. One possibility for enabling this is extending the knowledge base testing interface to allow for debugging by displaying inference traces and values. This method of debugging is independent of the documents. Hence, the habituated view on the human-oriented layer of knowledge including domain description and modeling rationale then is not available. Still, this content is especially valuable, considering a complex task such as debugging, as it provides the context, which the knowledge had been creating in, including the modeling rationale for instance. From a technical point of view this is the simple method. In particular, any existing debuggers for the employed knowledge repository implementation can be employed. Nevertheless, in the end after a debugging session any corrections need to be done in the document content. Finding the corresponding markup expression for the piece of knowledge to be changed can be non-trivial.

A much better solution, while also implying more technical efforts, is embedding a debugging layer within the document-centered knowledge acquisition environment. That is, for each piece of knowledge, the reasoning result should be accessible within the presentation view of the document, e.g., using color codes or pop-ups. Then a user can inspect the semantics instantly after writing the markup expression of a piece of knowledge. This can be considered as the continuation of the interaction model of interactive alignment (c.f. Section 3.1.6.1) towards the semantic level. This also requires to keep track of a mapping from each markup expression to the created elements of the symbol level knowledge base ($e \mapsto \tau(e)$) to be able to retrieve reasoning traces. This mapping can easily be generated and stored during the compilation process.

3.2.3 Requirements for a Document-Centered Authoring Environment

Based on this discussion of advantages and challenges, the requirements for a document-centered authoring environment can be derived:

- **Simple Access:** Most important is the simple access to the documents. Access should be possible at low technical barriers and an intuitive interface should allow for a common way to create informal document content (e.g., text, tables, figures). While sophisticated features need to be included within the system in principle, the tool should be designed for keeping the main interface simple. Methods for designing simple to use interfaces are given by Maeda in the Laws of Simplicity [Mae06]. Hiding of complicated and rarely used features is one valuable strategy for this purpose.
- **Compatibility to Common Document Editing:** The system should comply to the characteristics of electronic document editing enumerated in Section 3.1. This also includes the compilation of the knowledge and the response to the user.
- **Markup Language:** A set of convenient and well readable markup languages, geared to the symbol level representation language, needs to be supported, including simple ones to be usable by non-expert users as well as expressive ones. For each markup, documented examples should be provided within the system from the beginning.
- **Testing Interface:** Some kind of testing interface should be included, allowing the users to instantly test the created/edited knowledge base. Beside a normal interactive interface, inline-query mechanisms can be helpful. There, an input data set for the knowledge base (for instance, a query or a problem description) is written into a document. Then every time the document is displayed, the current knowledge base is executed on that input data and the result is shown along with or instead of the input data content in the document. In that way, test input data can be reused over time. This mechanism can also be easily combined with continuous integration.
- **User Assistance:** Above, several challenges emerging in document-centered knowledge authoring have been discussed. An efficient and user friendly document-centered authoring environments needs to address these by providing assisting technologies for editing (syntax check, autocompletion), debugging (interactive debugger), visualization, versioning (backup with simple revert function), and refactoring.

3.3 Semantic Wikis and DCKA

Wikis are one of the most successful application classes coming up with the successful emergence of the so called Web 2.0. Since their introduction by Ward Cunningham [LC01] wikis have been installed as simple but effective knowledge management solutions within almost every larger organization to support communities of practice. Hundreds of wiki clone implementations have been created as commercial or open software. Some of the most important wiki engines are Media Wiki⁴ (the wiki engine of wikipedia), FOSWiki⁵, and Confluence⁶, which is a commercial solution. An overview on existing wiki engines is maintained on Wikipedia⁷.

3.3.1 Semantic Wikis

The key strength of wikis is the simple and unconstrained contribution mechanism. However, this also implies challenges as the knowledge is usually created entirely in an unstructured way. Therefore, it cannot be processed by machines (except as plain text) to help querying or refactoring of the knowledge. This flaw was addressed by so-called *semantic wikis* [SBBK09], which are an extension to the normal wiki concept. In semantic wikis, the content is not only managed as plain text but additionally a structured model of the knowledge is formed. By use of this model, the system is able to support tasks like search, navigation, queries, and maintenance much more efficiently. However, the creation and maintenance of the model in parallel to the normal (unstructured) content also poses additional workload on the users, which needs to be addressed. The semantic wiki system most widely used is *Semantic MediaWiki* [KVV06]. One major goal of the Semantic MediaWiki project was to turn Wikipedia into a semantic wiki that allows for queries on the content that can be automatically answered by the system [VKV⁺06]. However, the amount of workload to create the model and the scalability of the required query engine at the scope of Wikipedia are large challenges for this approach. Other semantic wiki engines, being rather diverse with respect to the nature of the model and way it is created by the users, are SweetWiki [BGS⁺11], AceWiki [Kuh08], KIWI [SEG⁺09], MOKI [DFGR12], or SWIM [Lan08].

3.3.2 Wikis and DCKA

An authoring environment requires a centralized repository for document management and compilation of the knowledge base and further a mechanism providing the users access to the documents. Wikis provide all characteristics of a collaborative document authoring system. In particular, wikis comply to the expected user experiences of document authoring stated in Section 3.1. Each wiki page can be considered as a document. (These terms will be used synonymously in the further.) Users can freely edit the content and the pages will not be modified by the system without user action. Further, new pages can be created easily provided with a descriptive name. The web browser as front-end and the simplistic wiki interface have been proven to pose rather

⁴<http://www.mediawiki.org>

⁵<http://foswiki.org/>

⁶<http://www.atlassian.com/software/confluence/overview>

⁷http://en.wikipedia.org/wiki/List_of_wiki_software

low barriers and are widely accepted. Therefore, we claim that wikis are perfectly suited to provide the basis for a document-centered knowledge acquisition environment.

3.3.3 Semantic Wikis and Knowledge Engineering

As discussed above, the purpose of a semantic wiki is to additionally create a formal model along with and corresponding to the content. While the use cases of semantic wikis are rather diverse, according to Krötzsch et al. [KSV07] one can distinguish two major categories of applications:

- **Semantic Data for Wikis:** In this application scenario the knowledge management aspect of the tool is in foreground. A knowledge management solution is desired that could either be served by a standard wiki or a semantic wiki providing additional functionality enabled by the semantic model. Here, primarily semantic navigation and search is important enabling the user to find and manage the actual wiki content (such as texts and images) more effectively. Hence, the purpose of the semantic model is to improve the tool's content management capabilities.
- **Wikis for Semantic Data:** Contrary, in the *Wikis for Semantic Data* scenario the semantic model itself is the actual artifact of desire. The wiki here basically is used as a development environment for the model. The additional wiki content (only) supports the development process, illustrating and documenting the domain knowledge. Depending of the application scenario, after development the knowledge model/base is often extracted from the wiki to be installed in the final productive system, such as a decision-support system for example.

The boundaries of these two general scenarios are not strict and they may overlap. For example, if a knowledge base for a decision support system is created and additionally an information system for manual research should be provided, both scenarios are present at the same time. This mixed scenario in principle can also be handled by semantic wikis effectively. While most applications and systems reported about in literature have to be categorized into the first scenario, from the view point of classical knowledge engineering the second scenario is predominant. Therefore in this work the application of semantic wikis is considered in this way, i.e. providing (a basis for) a development environment for knowledge bases.

Most semantic wiki systems available or described in literature are using a knowledge model that is based on RDF⁸. However, in principle arbitrary knowledge representation formalisms can be employed to capture the model of the knowledge, depending on the application. As already discussed in Chapter 1, the knowledge engineering approach presented in this work is not tied to a specific symbol level knowledge representation. It considers knowledge acquisition in document-centered style using a semantic wiki-based tool for various kinds of knowledge representations. In the following section, we introduce the approach with respect several kinds of different knowledge formalisms to demonstrate the wide applicability of the approach.

⁸<http://www.w3.org/TR/2004/REC-rdf-primer-20040210/>

3.4 Markup-based Knowledge Acquisition Tools

Beside semantic wikis currently there are not many tools available that follow the principle of knowledge formalization with markups in documents. One example in this direction is *Drools expert*⁹ by JBoss, which is a free business rules engine [Bro09]. It uses predicate logic style rules with predicates and variables. The Rete algorithm [For82] is employed for the rule inference. The encoding of the business processes of an organization as rules, however, is a non-trivial task of knowledge acquisition. Drools expert comes with an knowledge acquisition tool being mainly syntax-based. Figure 3.9 shows a screenshot of the tool. It allows to define the rules by a specific syntax within different source files. However, there are still several important

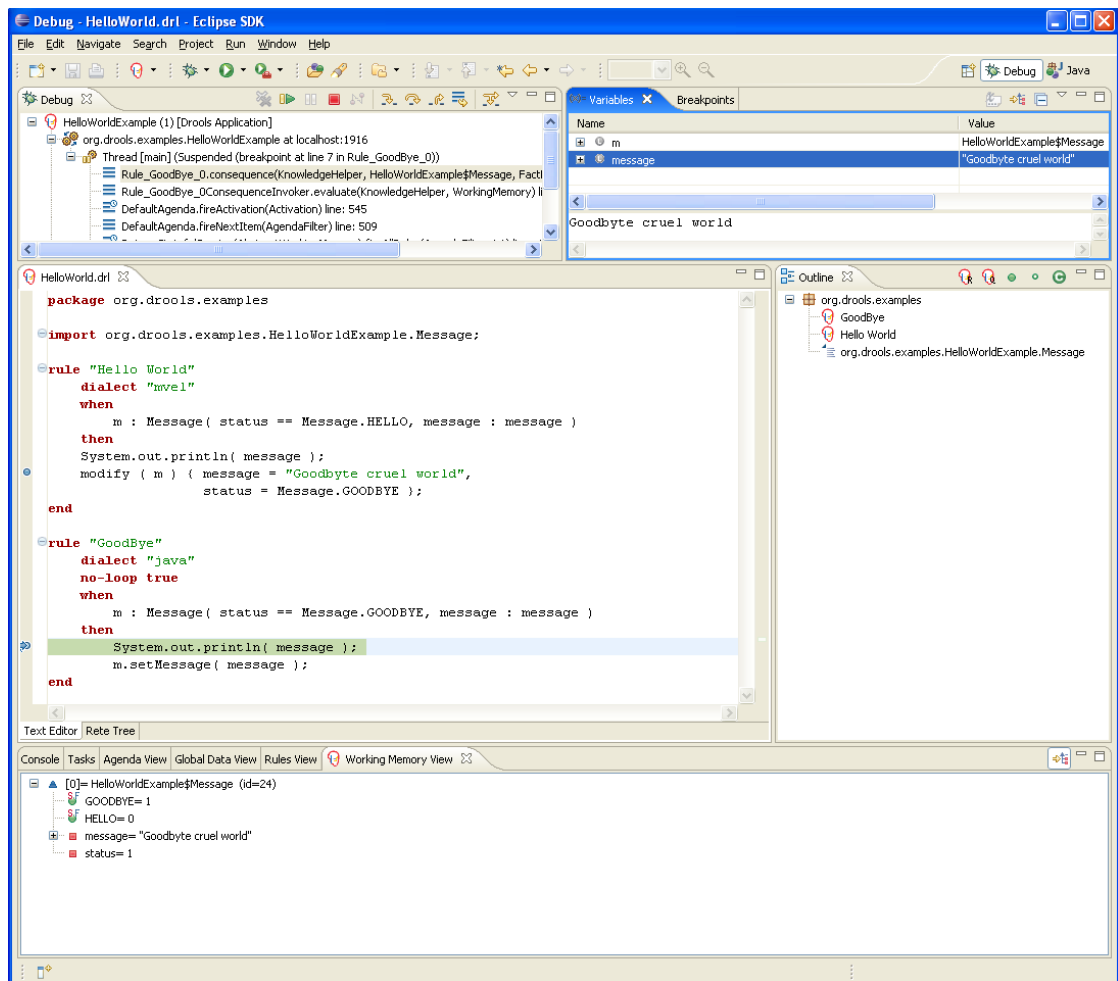


Figure 3.9: The knowledge acquisition environment of drools expert.

⁹<http://www.jboss.org/drools/drools-expert>

characteristics of a document-centered authoring environment missing, when compared to the specification given in Section 3.1. Most importantly, no informal content can be inserted into the documents, such as informal text, figures, or charts. Only the formal rule syntax may be used (, while comments are allowed when using the proper comment escape syntax). Further, the tool does not allow for collaboration on the source documents¹⁰, which is a key aspect of the mixed-initiative knowledge acquisition strategy, for example allowing for incremental formalization involving persons of different expertise. Further, the tool, being an extension for the complex and comprehensive IDE eclipse¹¹, is well-suited for users with programming background. However, for users without technical background it does not comply to the low barriers principle as the flood of complex functionalities offered by the user interface may easily frighten novice users.

Nevertheless, knowledge acquisition of business rules knowledge, such as drools, is a scenario that can be addressed by document-centered knowledge acquisition as introduced in this chapter. An appropriate authoring environment considering the aspects discussed above easily can be envisioned. Therefore, a syntax similar to the one shown in Figure 3.9 can be employed. While the incorporation of Java statements is convenient for programmers, it might not be necessary in every application scenario. If it is not required probably a more simplistic syntax for the representation of the rules might be suitable.

3.5 Different Application Scenarios for DCKA

Looking at intelligent systems at a broader scope a whole range of knowledge intensive application scenarios from the AI landscape are relevant today. Beyond classical knowledge-based systems for diagnosis or planning, applications like e-learning or data-mining (business intelligence) can be considered, showing fundamental differences in nature of the required knowledge. The document-centered knowledge acquisition approach, not being specific for a particular knowledge formalism, can be applied to a wide range of these scenarios. In the following for a diverse selection of different knowledge intensive scenarios the approach is introduced in an exemplary way.

3.5.1 Ontologies in RDFS/OWL

RDFS¹² and OWL¹³ are two related languages for formal ontology representation having arisen from the last decade of semantic web research. Due to their precise standardization and the good tool availability, they play an important role in ontology engineering today.

For either language different markups can be used to form ontology statements, including for example the Turtle Syntax¹⁴ for RDF and the Manchester Syntax for OWL [HDG⁺06]. More details about developing ontologies with document-centered knowledge acquisition can be found in [RBF12].

¹⁰except when using additional version-control plugins for eclipse, e.g., SVN

¹¹<http://www.eclipse.org/>

¹²<http://www.w3.org/TR/rdf-schema/>

¹³<http://www.w3.org/TR/owl2-overview/>

¹⁴<http://www.w3.org/TeamSubmission/turtle/>

3.5.1.1 Markup for RDFS:

The major categories of objects in RDFS are classes, properties and individuals. While some objects are predefined in the RDF Schema language, the definition of new domain objects to create the domain terminology is an important task when creating a new ontology. For this purpose, different kinds of simple markups can be provided. One possibility is the use the object category keyword (at line beginning) followed by the object term name as shown in the following markup example:

```
1 Individual Jochen Reutelshoefer
2 Class Person
3 Class Author
4 Class Writing
5 Property authorOf
6 Property hasTitle
```

In this example the object *Jochen Reutelshoefer*, *Person*, *Author*, *Writing*, *authorOf*, and *hasTitle* are defined in a straight forward way. This markup is line-based, i.e. each statement is terminated by the linebreak. In the following example, additional knowledge about some of these objects is defined in a concise way:

```
1 Individual Jochen Reutelshoefer type:: Person
2 Class Person
3 Class Author subclassOf:: Person
4 Class Writing
5 Property authorOf(Person -> Writing)
6 Property hasTitle(Writing -> PlainLiteral)
```

In line 1, additional to the introduction of the individual *Jochen Reutelshoefer* also a class assertion is given. For the class *Author* a subclass relation is defined in a similar way in line 3. In line 4 and 5, the properties *authorOf* and *hasTitle* are provided with *domain*¹⁵ and *range*¹⁶ definitions that describe to which classes subject or object resources of this property belong to, respectively.

The example page shown in Figure 3.10 shows the use of the different markups mentioned within a prototype system of a corresponding document-centered knowledge acquisition environment. Declarations of new entities are highlighted in purple while references to existing ones are rendered in green. Predefined vocabulary (RDFS/OWL) is rendered in bold black font. Individuals can be introduced by using the '*Individual*' keyword (1), while roles are defined using the keyword '*Class*' (2), as illustrated above. Object properties can be defined in a similar way and optionally class references as domain and range can be given in brackets (3). Further,

¹⁵<http://www.w3.org/2000/01/rdf-schema#domain>

¹⁶<http://www.w3.org/2000/01/rdf-schema#range>

3.5 Different Application Scenarios for DCKA

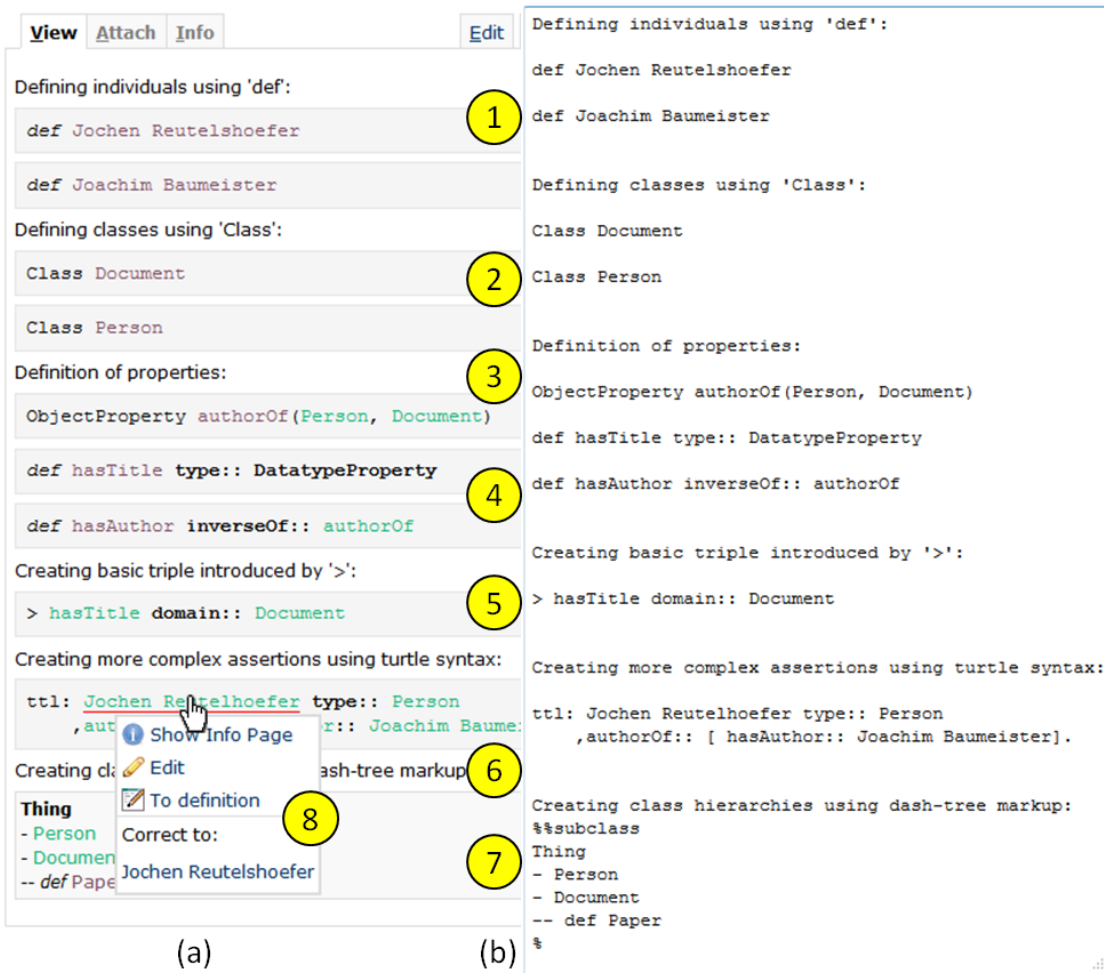


Figure 3.10: An example wiki page showing several markups to define ontology components in view (a) and edit mode (b).

3 Document-Centered Knowledge Acquisition

entities can also be defined as triples (4). Simple triple relations (5) can also be asserted by '>'. Basic RDFS/OWL vocabulary is available from the beginning. For more advanced users a version of the *Turtle Syntax*¹⁷ is available to create more complex RDF expressions introduced by '*ttl*'. For the quick and simple definition of explicit class hierarchies a dash-tree markup (7) can be employed. There each class of a tree node is considered as *rdfs:subclassOf* its dash-tree father class, i.e., its predecessor having one dash less in its prefix.

3.5.1.2 Markup for OWL

The ontology web language OWL [HPSvH03] has evolved from the former ontology representation language DAML¹⁸ and OIL [FvHH⁺01, Hor02]. The language was provided with XML-based and description logics style syntaxes. To improve readability for humans the Manchester Syntax¹⁹ [HDG⁺06] for OWL has been designed. Beside improved readability of the language, it also provides a convenient way for editing in the style of document-centered knowledge acquisition. The syntax is frame-based, that is, each ontology entity is defined as a frame describing the characteristics of the entity. The keywords for the entity definitions (e.g., '*Class:*', '*Individual:*') indicates the use of a Manchester Syntax statement.

In Figure 3.11 an exemplary document from the pizza domain²⁰ is shown describing *Pizza Margherita*. On the lower part, the Manchester Syntax is used to add the definition of a corresponding class to the ontology (a). The right hand part (b) shows the corresponding document source text.

3.5.2 Diagnostic Problem-Solving Knowledge with d3web

Another problem-solving task prominent in AI is the diagnosis or classification task, being relevant in many domains as for example disease diagnosis in medicine or fault diagnosis on technical devices. There, for a given problem description the most appropriate solution from a set of predefined solutions is to be selected. For this inference task various problem-solving mechanisms and strategies are available. More details about knowledge representation and inference for diagnostic systems are given by Puppe et al. [PGPB96].

The *d3web*²¹ system is a framework for knowledge representation and inference for diagnostic knowledge bases implemented in Java. In the following, we briefly illustrate how knowledge bases for d3web can be created with document-centered knowledge acquisition. Therefore, we show some markups to capture d3web knowledge and then discuss some patterns of knowledge organization within the document-space.

¹⁷<http://www.w3.org/TeamSubmission/turtle/>

¹⁸<http://www.daml.org/>

¹⁹<http://www.w3.org/TR/owl2-manchester-syntax/>

²⁰<http://www.co-ode.org/ontologies/pizza/>

²¹<http://www.d3web.de>

3.5 Different Application Scenarios for DCKA

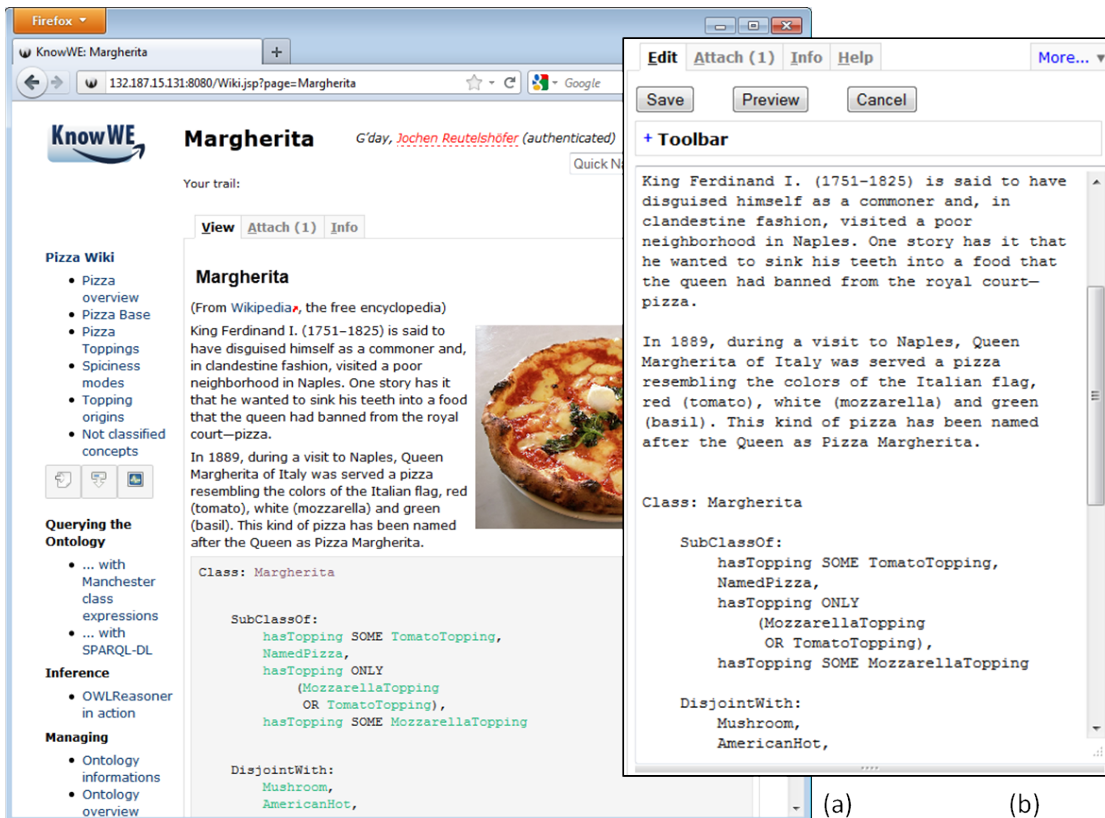


Figure 3.11: An example document about Pizza Margherita in view (a) and edit mode (b).

3.5.2.1 Markups

In the following, we show a selection of markups that are suitable for capturing different kinds of knowledge for the d3web framework. For the illustration the examples are taken from a simple sports advisor domain.

Terminology and Decision Trees Listing 3.1 shows a dash-tree style markup that allows to define hierarchical relations of entities. For each element the preceding element with one dash less in its prefix is the hierarchical predecessor of the element. Elements with zero dashes are root level elements. An example for the dash-tree markup is illustrated in Listing 3.1. It can be used to define terminology of the domain, e.g., questions and answer alternatives, but also to create derivations according to the well known decision tree principle.

```
1 %%question
2 Naive Sport Advisor
3 - Training goals [oc]
4 -- endurance
5 --- Physical problems [oc]
6 ---- knees
7 ----- Swimming (!)
8 ---- no problems
9 ----- Running (!)
10 -- increase muscles
11 --- Muscle Questions
12 Muscle Questions
13 - Favorite regions of muscles [mc]
14 -- arms
15 ...
16 %
```

Listing 3.1: A decision tree markup example from a sports advisor domain in dash-tree style.

In line 3 the once choice question *Training goals* is defined, while its successors *endurance* in line 4 and *increase muscles* in line 10 are forming the answer alternatives. Questions that are successors of answer alternatives of other questions are follow-up questions in the interview order as for example *Physical problems* in line 5. Further, solutions can be established if being defined as successors of answer alternatives as for example *Swimming* in line 7 and *Running* in line 9. By the use of the indentation, knowledge for interview control and derivation can be defined according to the decision tree principle. By use without indentation, only the terminology objects of the domains are defined in enumeration style.

Rules In Listing 3.2 an intuitive markup for defining condition action rules is shown. The condition part is indicated by the 'IF' keyword, while the action part is introduced by the 'THEN' keyword. The condition of a rule can be composed by logical operators over atomic conditions on terminology objects that have been defined before, e.g., by using the dash-tree markup introduced above. In the action part different kinds of actions provided by d3web can be defined, such as value assignments or interview control actions.

```

1 %%rule
2 // Abstraction rule r1
3 IF (Height > 0) AND (Weight > 0)
4 THEN BMI = (Weight / (Height * Height))
5
6
7 // Scoring rule r2
8 IF (Training goals = endurance)
9     AND (BMI < 30)
10    AND NOT(Physical Problems = knees)
11 THEN Running = P4
12 %

```

Listing 3.2: A rule markup for condition-action rules in d3web.

Set-Covering Knowledge Another method for representing diagnostic knowledge is set-covering models [Pup93]. There, a solution is described by a set of expected findings, so-called set-covering relations. For a given problem description, the solution with the best coverage can be computed. Listing 3.3 shows a simple markup that allows to define set-covering models in a list-based style.

```

1 Running {
2     Favorite regions of muscles = legs,
3     Favorite regions of muscles = bottom,
4     Training goals = reducing weight,
5     Training goals = endurance,
6     Calorie consumption (30min, 75kg) = high,
7     ...
8 }

```

Listing 3.3: A simple markup for set-covering knowledge in d3web.

After the described solution, a coma-separated list of set-covering relations is defined. Each relation consists of a valued finding over the defined terminology.

The markup examples discussed above only show a subset of the d3web knowledge representation, as they are meant for illustration of capturing diagnostic knowledge with markups. Further markups for d3web are discussed in [BRP07b].

3.5.2.2 Knowledge Organization

As discussed in Section 3.1 there is a degree of freedom how the content elements can be distributed within the document space. Diagnostic knowledge bases typically are consisting of the components solutions, inputs, and problem areas. For the distribution of diagnostic knowledge in the document space, the following patterns have been identified by experiences in various projects:

- **Solution-oriented Distribution:** For each possible system output (or coherent group of outputs), a document is created. It contains the definitions of the output and formal knowledge to derive this particular output. For larger systems, sub-documents can be defined that are linked from the main document.
- **Problem Area-oriented Distribution:** For each problem area (coherent and named groups of inputs to the system), a document is created. Each document contains the definitions of the problem area (e.g., symptoms concerning the problem area) and links to articles, where derivation knowledge is defined relevant to the particular problem areas.
- **Concept-oriented Distribution:** For each concept of the application domain, including solutions and inputs, a document is created. Attributes and relations of this concept are also defined in this document. Also links to related concepts are included.

More details about deriving knowledge distributions within the document space in general are discussed in Chapter 4.

3.5.3 Knowledge for Exploratory Data Analysis

Data analysis and data mining aim to aggregate and visualize large data sets and to discover new interesting patterns within the data [WF99]. The data sets are basically consisting of tuples of attribute-value pairs. Sophisticated analysis and mining algorithms have been established to find meaningful patterns within these tuples. However, these algorithms in many cases require additional knowledge about the attributes within the tuples. Background knowledge about the value range of an attribute or the relation between different attributes has to be defined to allow effective knowledge discovery. In the following, we will present markups for capturing background knowledge for Exploratory Data Analysis [Tuk77]. The examples are taken from a medical domain considering data about traveling diseases.

Default Values Often for particular attributes it is "normal" to take a certain value, that is, the default value, and more extraordinary, if other values are taken. The default value is in most cases, but not always, the most frequent value for that attribute in the data set. However, not for all attributes a default value exists. For example, consider a data set from the medical domain, in which the occurrence of any disease is represented as a single boolean attribute. Then it is usually assumed that a patient does not have a certain disease unless stated otherwise. Hence, the default value is 'false' in this case.

A very simple knowledge acquisition pattern to enter information on default values into a text document could be to use a keyword "DEFAULT" and then the respective attribute and — separated by a colon — its default value:

```
| DEFAULT Hypertension: false
```

Discretization Discretization is a basic operation which transforms a numerical attribute into an ordinal/nominal attribute. It is a standard pre-processing technique to adapt the data set to the requirements of an analysis algorithm. The discretization boundaries can easily be specified in a textual markup. They can then simply be adapted to the analyst's needs ad-hoc. Therefore we propose the following markup:

```
1 DISCRETIZATION Age [18;25;50;65]
```

This markup expression specifies that the expert considers the values of 18, 25, 50 and 65 as suitable cutpoints to discretize the numeric attribute *age*. When the respective knowledge is entered into the knowledge base, the construction of a new nominal attribute based on the respective numeric attribute using the specified discretization cutpoints is triggered.

Abnormality Information In many applications, for instance in the medical or engineering domain, there is an interval of measured values that is considered as in line with guidelines/specifications, while measurements beyonds the limits of the interval are considered as either "too high" or "too low". This combines the concept of discretization and default values: A new attribute can be constructed that discretizes the base attribute into three parts, "too low", "in range" and "too high" and additionally marks the "in range" interval as the default value. A text fragment, that specifies, that the Body-Mass-Index should usually be in the range of 20 to 25 could for example be specified by the following markup expression:

```
1 NORMAL 20 < BMI < 25
```

Value Categorization Knowledge In many data sets attributes with a wide range of values exist that could be grouped into several categories, allowing for aggregated evaluations. As these categories are not necessarily fixed, experts may want to define them ad-hoc. By defining them, similar to the discretization case, new attributes are created. In case of non-overlapping categories the category information can be saved in a single new attribute, that has a distinct value for each category. In contrast, for overlapping attributes it is necessary to create a new boolean attribute for each category, which is set to true, if the respective category applies.

For formalizing categorization knowledge we propose to use the following markup:

```
1 CATEGORIZATION disease-type FOR disease DEFAULT other/none
2 Tropical disease: Malaria, Chagas disease, Dengue
3 Childhood disease: Rubella, Chickenpox, Measles
```

The input of these markup expressions would trigger the construction of a new attribute *disease-type* that is based on the existing attribute *disease*. The new attribute takes the value *Tropical disease*, if the disease was either *Malaria*, *Chagas disease*, or *Dengue* and the value *Childhood disease*, if the value of *disease* in the respective case was one of *Rubella*, *Chickenpox*, *Measles*. If none of these categories does apply, then the case takes the default value given in the first line of the formalization pattern, in this case *other/none*.

Ordinality Information Attributes with non-numerical values that still have a natural order often are represented as nominal attributes in the systems. One example for an inherent ordering is an attribute with the values *low*, *medium*, *high*, while the order is often not reflected in the original data set. In that case this different measure of scale can be communicated to the system using the following markup:

```
1 ORDINALITY age: child < adolescent < adult < senior
```

The given information can be used for the ordering of values within evaluations, e.g., in a bar chart. Another application for this type of knowledge is that a subpopulation for a focused analysis can be specified using the ordinal scale, e.g., all instance with *age* \geq *adolescent*.

With the markups shown above, a background knowledge base for the data analysis task can be created. If the analysis engine and the document-centered authoring environment are integrated with each other, quick knowledge base refinement cycles can be driven, enabling instant testing. The required terminology of the domain terms, as for example *disease*, *Dengue*, *age*, can be imported by the data set. In that way, spell checks and code completion for these terms can be provided.

3.5.4 Training Cases for e-Learning with CaseTrain

Today, e-Learning provides a novel method for effective knowledge transfer at rather low teaching costs, given an appropriate e-Learning system once established. Depending of the way how the learner interacts with the system, an e-Learning system requires the subject domain knowledge to be encoded in a computer interpretable format. One e-Learning paradigm, being rather successful today, is case-based training [RHT⁺06]. In this section, we want to illustrate the document-centered knowledge acquisition method for the creation of training cases for the case-based training system *CaseTrain*²². There, the learner is first introduced to a problem situation of the domain and then asked a sequence of questions about this situation each possibly providing additional information. After processing the sequence of questions the user is instantly provided with the evaluation of his decisions. He is also able inspect the case including the explanations of the correct answers. From the knowledge acquisition perspective this requires for each case the computer readable definition of the sequence of questions with answers including the ratings and illustrative information. Therefore, a markup format has been defined to be able to define CaseTrain cases as Microsoft Word documents²³. These documents can be compiled to executable CaseTrain cases by an upload to a web-service.

One foundational issue of document-centered knowledge acquisition always is error handling and authoring support. This aspect can be improved by providing an online authoring environment that instantly detects markup or formatting errors while editing and suggests options for correction. Additionally, the authoring environment can easily provide content versioning (backup) and collaborative authoring. Figure 3.12 shows a prototype of a corresponding knowledge acquisition environment with an example case (in German language).

²²<http://www.casetrain.de>

²³http://casetrain.uni-wuerzburg.de/doku/format_case.shtml

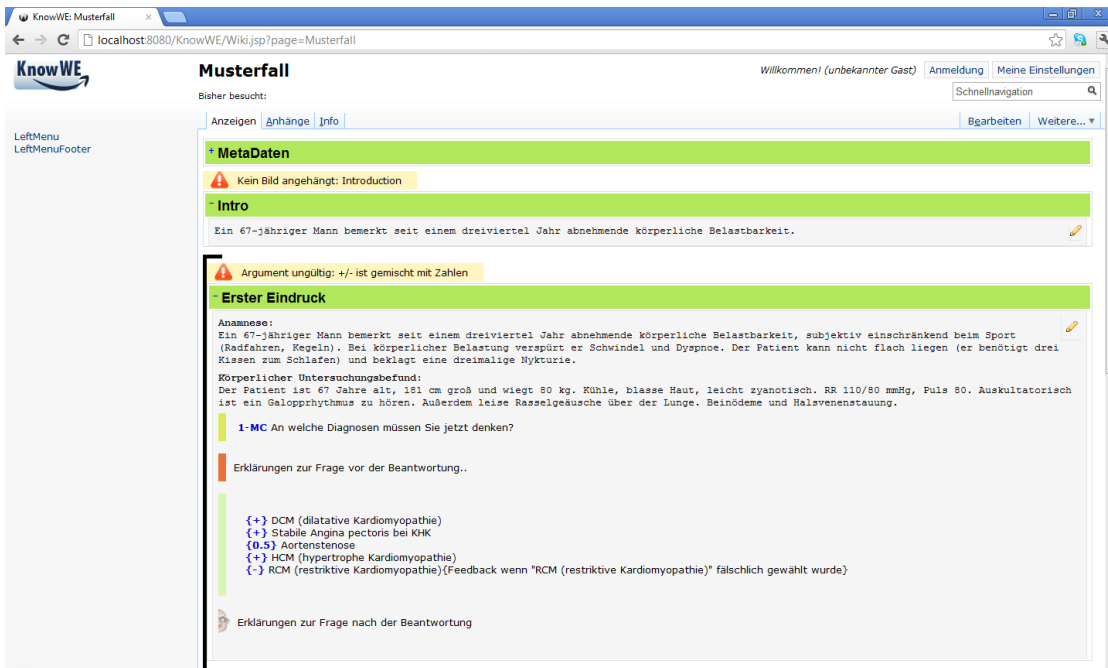


Figure 3.12: Excerpt of an example training case for CaseTrain created with a document-centered authoring environment (in German language).

3.6 CommonKADS

CommonKADS is a comprehensive methodology for knowledge management and knowledge engineering [SAA⁺01]. It has been evolved in the nineties from its predecessor KADS [SWB93], which has been developed since the late eighties. The approach is based on experiences made on numerous real-world projects and aims to provide a helpful framework for all kind of knowledge intense tasks at a broad scope.

3.6.1 A Brief Overview of CommonKADS

In the following, we summarize the most important aspects of the CommonKADS methodology, as far as being relevant for this work.

3.6.1.1 The CommonKADS Model Suite

The model suite plays a key role in the commonKADS knowledge engineering approach. Each model describes a particular aspect, while their combination provide a complete view of the project. The organization, task, and agent models analyze the organizational environment and the critical success factors for a knowledge system. The knowledge and communication models form the conceptual description of problem-solving functions and data that need to be included

3 Document-Centered Knowledge Acquisition

in the knowledge-based system [SAA⁺01]. In a further step, a detailed technical specification is made, leading to the design model, which then serves as a basis for implementation. However, not all of the models need to be employed in any project. The models are captured and evolved in the so-called model documents.

3.6.1.2 The CommonKADS Process Model

As a process model for the project management, the commonKADS methodology proposes the use of a customized version of the spiral model as proposed by Boehm [Boe88]. In every cycle of the spiral the activities review, risk assessment, planning, and monitoring are performed, increasing their scope within every cycle according to the progress of the development. The spiral model aims to combine the advantages of the waterfall model, providing clear project control, and the agile process model of rapid prototyping in a evolutionary manner, providing more flexibility to readjustments.

3.6.1.3 The CommonKADS Role Model

CommonKADS distinguishes the following roles within the knowledge engineering process. A person, however, can act in multiple roles in a project.

- **Specialist:** The specialist or expert is the owner of the knowledge. Therefore, this role acts as knowledge source for the project.
- **Analyst/Engineer:** This activities of this role include knowledge engineering related tasks as task analysis, knowledge elicitation, knowledge modeling, and inference design.
- **System Developer:** The task of this role is the implementation of the knowledge system according to the knowledge models delivered by the analyst.
- **User:** The knowledge user makes use of the knowledge-based system in the application scenario stated as the goal of the project.
- **Project Manager:** The project manager controls the development process. This also includes project planing, requirements monitoring, and risk assessment.
- **Knowledge Manager:** The knowledge manager controls the knowledge management strategies on the business level.

The CommonKADS methodology, which is forming the de-facto standard approach to knowledge engineering, also has a disadvantage when used in its pure form without early prototyping. The extensive analysis, modeling, design activities driven out at the beginning imply, that any kind of running functionality will appear at a rather late stage. As the domain specialists are not capable to understand most of these activities, there is the risk of upcoming frustration on their side. An involvement of these project participants as far as possible at an early stage is one major goal of the document-centered knowledge acquisition approach. In the following, we outline possibilities of a combination of the approaches.

3.6.2 CommonKADS and DCKA

CommonKADS provides a comprehensive methodology for knowledge engineering for knowledge management and engineering. DCKA in contrast only proposes a tool (category) for knowledge formalization and some strategies for its use. Hence, the two approaches not really can be put in comparison. Still, we try to summarize the most important aspects in the following. Further, possibilities for the combination of both are briefly discussed.

3.6.2.1 Comparison

First of all, CommonKADS and DCKA are following different knowledge formalization strategies. While CommonKADS rather strictly employs indirect knowledge acquisition, DCKA aims for a different strategy. As discussed in Section 1.2.2.1, DCKA proposes a compromise between direct and indirect knowledge acquisition, called mixed-initiative knowledge acquisition. It aims to create the socio-technical conditions that allow to involve the domain specialists in the knowledge formalization process as far as possible by the principle of active participation. In CommonKADS in contrast, after elicitation of the knowledge from the domain specialists, the knowledge is modeled and implemented by knowledge engineers and system developers in a rather technical multi stage process. This formalization process can guarantee a high quality result, but is not suited to involve domain specialists. Activities of direct knowledge acquisition are not intended in CommonKADS. In consequence, CommonKADS is a more structured approach while DCKA has a stronger focus on agility, useful for but not limited to rapid-prototyping of knowledge-based systems (c.f., [SBGB88]). In general, the DCKA does not propose the explicit distinction of the different roles, as described for CommonKADS above, but proposes contribution according to the skills of each participant respectively. However, a strict comparison of the two approach is not possible anyway as CommonKADS has a much broader scope than DCKA. Nevertheless, many aspects of CommonKADS can be adopted for DCKA, to form a combination.

3.6.2.2 Combining DCKA and CommonKADS

Especially in the analysis phase, many methods of CommonKADS are independent of the principle of DCKA. The model suite and the guidelines for their creation can be valuable for DCKA. As these models are described in documents, they can be included into the document base straight forward. The model documents are developed by the knowledge engineers. In parallel, an informal description of the relevant knowledge can be created, involving the domain specialists. It can then serve as a starting point for incremental formalization (c.f., Section 3.1.5.2). After the design has been finished, at the level of detail as desired, the implementation phase can begin. The content of the declarative knowledge base is en-woven into the domain description documents using in a collaborative effort using suitable markup languages. On this task, the domain specialists should be included as far as possible. Implementation of new markup languages, including compilation, and inference engines is driven out in parallel by the knowledge engineers and system developers and integrated into the document-centered authoring environment. A more precise description of this parallel process of markup design and implementation

3 Document-Centered Knowledge Acquisition

on the one hand, and knowledge acquisition on the other, is described in Chapter 4 called the meta-engineering approach.

In that way, many advantages of CommonKADS, such as the structured model-driven project control, can be adopted for DCKA. Considered the other way round, some advantages of DCKA, such as agility and strong involvement of domain specialists in the whole process, can be added to the CommonKADS methodology. Hence, a beneficial combination of both approaches is possible, while not retaining both in their pure form. The principle of strict indirect knowledge acquisition needs to be loosened for a reasonable combination with DCKA.

4 A Meta-Engineering Approach for DCKA

.. the question of how to break down the whole task most effectively: the demands are such that elegance is no longer a dispensable luxury, but decides between success and failure.
E.W.Dijkstra [Dij86]

The history of software engineering told that the requirements for programming code is not only that its execution produces the desired behavior but also that it can be easily and quickly be understood by the programmers. Dijkstra demanded no less than elegance for the way how code should be structured in component-based software development. Beside improved reusability a reasonable organization of the code fragments strongly increases code readability and therefore maintainability, often finally being decisive for success or failure of the software development project. Elegant—in a sense of comprehensive— structuring of the content fragments is equally important in document-centered knowledge acquisition. The general document-centered knowledge acquisition approach has been introduced in Chapter 3 as an alternative to the use of GUI-based knowledge acquisition tools. However, the question of how content and tool supposed to be set up to achieve efficient participation of the different user groups has not yet been addressed. Providing methods for improving the interaction of domain experts with a DCKA tool is the main objective of this chapter. A major problem of knowledge acquisition involving domain experts in general is that these experts need to learn the usage of a tool usually being new to them. Considering DCKA, we discussed the low barriers for basic contributions by using a web-browser and the well-known editing paradigm for documents. Still, there are things to be learnt by novice participants, e.g. the formalization mechanisms (markups). Usability of the tool is not only determined by its handling when inserting new knowledge, but the comprehensibility of the presentation of existing contents is at least as important. The user has to match the content visualization of a tool to his mental model of the knowledge to interpret given knowledge base content. This problem, also referred to as the *semantic gap* [Mer04], is known since the early days of expert system development. The major strategy to close this gap is the customization of the user interface of the knowledge acquisition tool towards the mental model of the user, also considering the domain context. The goal of such a customized tool is to present the knowledge in a way that is intuitively comprehensible by the target user group. This tool customization strategy already has long tradition for GUI-based knowledge acquisition interfaces. In this chapter, we examine the customization potentials for document-centered knowledge acquisition tools. There, customization means to deal with documents that have content, structure and (as far as possible) markups that are intuitively understandable by the user. The vision of the ideal situation shows a user that is not or hardly realizing that he is using a new tool or knowledge formalization mechanism but works with intuitively understandable documents. While this scenario does not appear to be fully realistic, it shows out that there are indeed possibilities to come a good step closer towards that point at least. The main focus of this chapter is to point out these

possibilities and to give guidelines for their application in practice. The use of graphical user interfaces for knowledge acquisition has been briefly introduced already in Section 2.1. Before looking closer at customization of document-centered tools a more detailed discussion of existing work on customized GUI-based knowledge acquisition tools is given.

4.1 An Overview of Knowledge Acquisition Tool Customization

”The conceptual model thus forms the basis of a language with which both the tool and the tool’s user can describe the contents of a knowledge base.” Musen [Mus88]

The early time of research on customized knowledge acquisition tools was beside others (c.f. [KBD⁺89]) substantially influenced by Mark Musen and his research at the Department of Medical Informatics at Stanford University. Based on the first experiences with customization of knowledge acquisition tools [MCW⁺86], he started developing the system family Protégé¹ in the late eighties. The original intent of the first generation of this tool was to reduce the knowledge-acquisition bottleneck by minimizing the role of the knowledge engineer in constructing knowledge bases [GMF⁺03]. Musen claimed that the use of customized knowledge acquisition tools is a necessary and effective way to achieve this. He describes the need of a conceptual model of the domain as a basis for a tool that is usable by experts. The resulting tasks of specification and implementation of such tools have been addressed by establishing a conceptual model of the domain in advance followed by the actual tool design and corresponding implementation [Mus88, Mus89b]. The (meta) knowledge for the model, being acquired early in a multi-stage knowledge acquisition process, should be used to generate the custom knowledge acquisition tools automatically [Mus89c] by applying so-called *meta tools*. These meta-level tools are domain independent tools that can generate custom-tailored knowledge acquisition tools from a high level specification of the domain, i.e., the conceptual model. In the nineties the research done by Musen and Eriksson focused on the development and application of such meta tools [Eri92, EM93]. Musen [Mus89a] reports about projects where customized knowledge acquisition tools generated with Protégé are used by experts to acquire skeletal planning knowledge and treatment plans (for cancer). He emphasizes that this method shows its main benefit if multiple distinct knowledge bases of similar structure and domain have to be created. Protégé has been evolved consequently for years [PETM92, GMF⁺03] incorporating different problem-solving methods and application tasks. At the time of the millennium Protégé has been reimplemented in Java with a highly extensible architecture based on a plugin framework. Since then many plugins, also by third party contributors, have been created, most prominently the OWL-Plugin [KFNM04].

The process for using meta level tools as described by Eriksson [Eri92] is outlined in Figure 4.1. At first, a conceptual domain model is established in cooperation of the knowledge engineer and the domain experts. The specification of the conceptual model is the input for an existing meta tool that generates a domain specific knowledge acquisition tool. That tool can then be used by the domain experts to actually create the knowledge base, most widely autonomously.

¹<http://protege.stanford.edu>

4.2 Design Time and Use Time: The Systems Design Dilemma

The weak point about that workflow is the specification task that is carried out at one stroke in advance. It is questionable whether a conceptual model of the domain serving as specification of a knowledge acquisition tool is able to anticipate all the socio-technical challenges that will emerge during the process of the actual knowledge base creation. Fischer [FG06] argues that despite the best efforts at design time, it is not possible to anticipate all the needs of the user of a design environment in advance. While system specification in advance is a general challenge in software engineering [MJ82], it turns out that it is particular hard for knowledge acquisition interfaces. This issue is outlined in more detail in the following section.

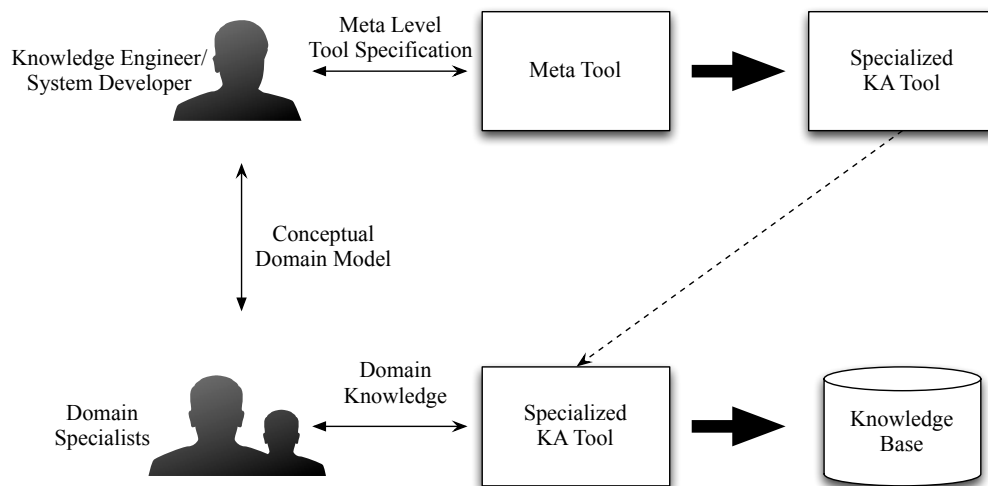


Figure 4.1: The process of generating customized gui-based knowledge acquisition tools from meta level specifications with meta tools according to Eriksson [Eri92].

4.2 Design Time and Use Time: The Systems Design Dilemma

Artefacts (especially novel ones) are designed when no experiences about their use are given, being a general problem about artefact design, not limited to software. This simple problem is known and discussed since the early days of system design not having lost its wickedness. The designer at *design time* usually does not have all information about the context of use the entity is put in at *use time* to anticipate all possible situations. For the creation of software systems large efforts have been put into the improvement of requirements engineering [Zav97] to handle that problem. Still, creating highly usable user interfaces tailored to the needs of a specific user group is a large challenge. When it comes to the design of knowledge acquisition interfaces being used by domain experts the situation is even more complicated. In common software development projects the user/client at least does have (more or less) precise conception of what he is going to do with the tool to be build and therefore is able to discuss aspects of the user interface. In knowledge engineering, domain experts not yet having had experiences with knowledge engi-

neering can not have a good notion about what they will have to do with the tool to be designed. As the knowledge engineers on the other hand do not have deep understanding of the domain context, requirements engineering is particularly hard under these circumstances. Hence, not only the design of the knowledge base itself but also the design of suitable knowledge acquisition interfaces is a large challenge in knowledge engineering, deserving explicit consideration.

4.2.1 Agile Software Development

Nevertheless, MacCracken et al. [MJ82] (and others) claimed that the specification of the requirements of some software cannot be stated fully in advance, not even in principle. One attempt to elude this problem is taking on the attitude to "*embrace change*", famously postulated by Kent Beck [Bec00]. This attitude proposes to accept the fact that specifications and requirements will change frequently as special characteristics of the scenario will not become obvious until an advanced stage of the development process has been reached. Agile development methodologies aim to provide methods to deal with these conditions [Coc02, Mar09, SB01]. The principle idea is to maintain a system that only provides a reduced number features but is runnable and usable at this scope since an early point of the development process. That system and the currently existing features any time are ready to be tested. These preliminary experiences can then be incorporated in the further development steps. In that way, feature by feature can be addressed and improved iteratively. Due to this flexibility, agile software development became increasingly popular within recent years, not only for open source projects [RSS09].

4.2.2 Meta-Design

Another approach from the field of human-computer interaction research addresses design environments considering a collaborative development process of digital artefacts by a user group of heterogeneous expertise. Therefore, it fits quite well into the context of this work. The conceptual framework called *meta-design* is proposed by Fischer [FS00] and is inspired by the way open source software is developed in an agile fashion. It proposes a development process model for the design environment where design time and use time overlap [Mac11].

Meta-design extends the traditional notion of system development to include users in an ongoing process as co-designers, not only at design time but throughout the whole existence of the system.

Fischer [FGY⁺04]

One important principle in meta-design is the notion of *underdesign* [FGY⁺04]. Underdesign proposes to not create and deliver completed and closed solutions but tries to provide social and technical instruments for the users to make system adaptations according to their needs at use time. In that way, it avoids that most of the relevant knowledge is required at the earliest part of the design process, when everyone knows the least about what is really needed. Further, meta-design is expected to form a co-adaptive process where the tool is adapted to the user but on the other hand with time the users adapt to the tool by gaining more experiences. This convergence from both sides after a certain time leads to significant increase of productivity and autonomy of the user participation.

4.3 Flexibility and Coordination in DCKA

To conduct a meta-design process one the one hand underdesign of the system is required at the beginning. On the other hand the process of coining this under-determinedness according to the experiences at use time has to be steered. These two aspects are discussed for DCKA in the following sections.

4.3.1 Flexibility in Document-centered Knowledge Acquisition

In Section 3.2.1 the possibilities of free structuring of the content in DCKA have already briefly been introduced. Now, we will discuss these aspects of flexibility in more detail considering the context of meta design for knowledge acquisition tools. The actual executable symbol level knowledge base, being the design artefact of the base level, is excluded from this meta level design consideration. On the meta level, the flexibility in DCKA provides a wide range of possibilities of how the content might be organized in different ways, not affecting the executable knowledge base. We distinguish the following three different aspects of flexibility:

1. **Knowledge Syntax:** Usually a document-centered authoring environment supports a standard set of markup languages. However, liberating oneself from a concrete system, providing specific markups, an important degree of freedom emerges. Various different markup languages can be envisioned to capture knowledge in a formal way. We assume, that the tool can be extended for supporting different knowledge markups, if considered beneficial for the knowledge acquisition process. The nature of a markup language strongly affects readability, writability, and may incorporate peculiarities of the current project or domain. Therefore, this aspect plays an important role. In contrast to the following two aspects, it requires engineering efforts on the system level. This system level design and engineering will be discussed in more detail later in the context of markup design.
2. **Support Knowledge:** Any domain specific content that is not compiled into the knowledge base, for instance including the content categories domain description and modeling rationale (c.f. Section 3.1.3), is called *support knowledge*. It serves as documentation putting the compiled markup expressions into context. Especially in the context of incremental formalization (c.f. Section 3.1.5.2) the support knowledge is a particularly important factor. What kinds of (informal) domain knowledge is contained in the document corpus and the way it is organized has a strong influence on the understandability of the content and therefore on the user's capabilities to work with the tool. The overall content should cover all domain knowledge that is relevant for the intended (compiled) knowledge base. This information needs to be partitioned into documents in a reasonable and comprehensible way. Usually this is done in a way that each document is treating one sub-topic of the domain, possibly using hierarchical structuring. In many cases, already existing material of different shapes (e.g., texts, tables, images) can be imported as support knowledge. For the domain descriptions, often the structure of existing documents to some extent can be retained if it is comprehensible and familiar to the domain specialists. The support knowledge enables the users (especially the domain experts) to find their way

through the knowledge base. Therefore, the comprehensible organization of the support knowledge is an important prerequisite for low barrier participation.

3. **Arrangement of Knowledge:** To actually form a computer interpretable knowledge base in document-centered knowledge acquisition, the formal knowledge needs to be inserted using a markup language. While complying to the given markup language there is still a large degree of freedom how to order and partition the statements within the documents. For comprehensibility it is reasonable to interweave these parts topically with the support knowledge. Hence, if the support knowledge is inserted in advance, it provides a kind of guideline, where formal relations should be placed. The markup expression of a relation should be placed as near as possible to the part of the support knowledge that describes the topic. Then the support knowledge serves as justification and documentation of the formal knowledge.

These aspects strongly determine the presentation of the knowledge base and therefore affects the user's capabilities of active participation. Hence, they need to be considered as characteristics of the knowledge acquisition tool itself. Due to the nature of DCKA the boundaries between tool characteristics and content blur. Being undetermined at project beginning these aspects constitute an inherent underdesign being the necessary prerequisite for effective meta design.

4.3.2 The Document-Centered Knowledge Acquisition Architecture

In principle, the aspects of underdesign discussed above on the one hand generate high possibilities for optimization but may on the other hand may also raise difficulties. For every contribution to the documents decisions about these three aspects have to be made. If these decisions are not made in a reasonable way the document corpus over time risks to develop into a chaotic and incomprehensible state. A good design can not be expected as a natural emergent effect of collaborative development but needs to be a coordinated effort. To prevent that risk and to support the user on contributions, a specification of a guideline, determining the variable aspects, is required. A specification of this kind defining a guideline for each of the three aspects we call a (*document-centered*) *Knowledge Acquisition Architecture* (KAA). The three aspects are considered to form a kind of state space as shown in Figure 4.2. A KAA then is one specific point

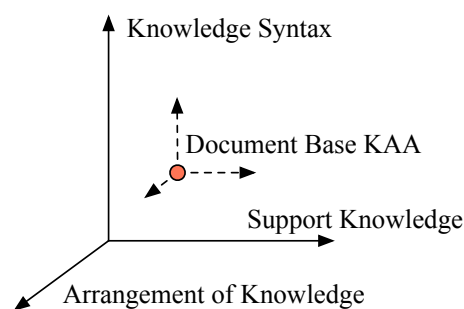


Figure 4.2: The document space allowing for customization of the KAA along three dimensions.

in this space. It determines the scope and arrangement of the support knowledge (1), the arrangement of the formal knowledge (2) and its syntactical structure (3). To guide the knowledge acquisition activities, the KAA should be explicitly formulated and written down, accessible to all involved persons. Maintaining the KAA as an additional document within the document corpus guarantees simple access for all participants. It determines in what way contributions should be performed (without being technically enforced). We do not expect that the user is always able to make his contributions compliant to the KAA specification. The knowledge engineer however has to be capable to rate and if necessary ensure this compliance. This complies to the idea of mixed-initiative knowledge acquisition: Every participant contributes at the best of his expertise in a flexible collaborative workflow, based on task decomposition. For instance an domain expert inserts knowledge about one topic, but in a way that does not comply to the current KAA. Then a participant more familiar with the KAA specification can restructure this knowledge accordingly.

The KAA specification can be compared to the editing instructions existing for the Wikipedia² project. For large categories of pages, template pages exist that describe how a page describing this kind of entity should be structured and what information needs to be included. The German wikipedia at the time of writing contains 112 such patterns³ (german: Formatvorlage) for various categories of entities such as person, movie, or chemical substance. There, a rough outline is given, including a short description what information should be stated in each section. Often also an infobox pattern is provided. In DCKA the KAA plays a similar role, i.e. guiding the user on contributions. Within the KAA many aspects need to be considered, such as where and how the terminology of the knowledge base is defined, how the derivation knowledge is formalized, or for which entities distinct pages should be created.

The KAA determines readability, comprehensibility, writability, and navigability of the knowledge acquisition environment and therefore the overall usability and efficiency. Nevertheless, the specification of a suitable KAA is a hard problem as already discussed in Section 4.2. For this specification task we introduce a dedicated process in the next Section.

4.4 The Meta-Engineering Process for DCKA

"In many situations a solution to a problem can not be created, but must be revealed through experimentation and exploration."

Schilstra and Spronck [SS01]

As the specification of an optimal KAA in advance has to be considered to be rather impossible due to the systems design dilemma, we propose to apply an agile process according to the principles of meta-design as introduced in Section 4.2.2. We present the meta-engineering process that allows to explore, specify, implement, and use an appropriate KAA for a given project. The evolution of the KAA strives to optimize the criteria understandability, maintainability, and therefore overall acquisition efficiency of the knowledge. The evolutionary process affects both, the document base (content level) and the authoring environment itself (system level).

²<http://www.wikipedia.org/>

³<http://de.wikipedia.org/wiki/Kategorie:Wikipedia:Formatvorlage>

Preconditions: We presume, that a suitable document-centered knowledge authoring tool is initially existing, that provides means of knowledge formalization (markups, testing capabilities). The software requires an extensible architecture that allows for the integration of new features easily, most importantly markups and corresponding authoring support. Further, the system should provide methods for the execution of semi-automated or automated refactoring tasks, e.g. by use of a scripting language. To coordinate the knowledge acquisition process and the meta process, a knowledge engineer experienced with document-centered knowledge acquisition is required. Also a system developer for the employed DCKA environment is required throughout the project to address the implementation efforts on system level. Of course, it is convenient that these two roles are occupied by a single person if possible.

An iterative process: Figure 4.3 shows the meta-engineering process, comprising the main activities exploration, design, and implementation. After the process has been initialized by the exploration phase, alternating design and implementation activities are carried out. The actual knowledge acquisition process building the knowledge base runs in parallel. The process is driven by iterated cooperative sessions involving the knowledge engineers and the contributors. In the following, we describe the distinct parts of the process in more detail.

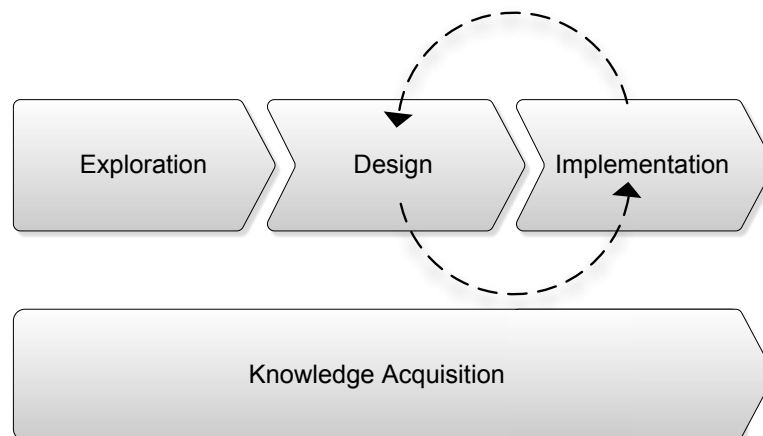


Figure 4.3: The phases of the document-centered meta-engineering process.

4.4.1 Exploration

At the beginning of the knowledge engineering project the meta-engineering process is started by the *Exploration* phase. The main purpose of this phase is to establish an initial version of a KAA for the project at hand. Therefore, different candidates for KAAs should be designed and assessed to gain some experiences how document-centered knowledge acquisition for the required knowledge base and the given domain can be arranged. For this purpose a small (but

if possible representative) subset of the domain knowledge has to be selected. Then small prototypes according to some ad-hoc defined KAAs are created for this part of the domain. At this point, only markups already provided by the tool are used. A secondary goal of the exploration phase is making the community of participants familiar with the general concept of document-centered knowledge acquisition. A few small toy knowledge bases for the selected knowledge subset are created in an exemplary way. That makes the domain experts familiar with the idea that the subject domain knowledge will be managed as documents with knowledge markups, that can be modified using the document-centered authoring environment. The activities of this phase should be carried out in close cooperation of all participants, for instance in a workshop-like format. When a KAA candidate is assessed by all available participants to be the most convenient one, it is established as the current KAA for the project and a specification document is written making it explicit. This indicates the end of the exploration phase and the actual knowledge acquisition process can be started according to this KAA. On the meta-level the design and implementation cycle is entered.

4.4.2 Design

”Meta-design is a useful perspective for analyzing projects where ’designing the design process’ is a first-class activity, i.e., creating the technical and social conditions for broad participation in design activities is as important as creating the artifact itself.”
Wright et al. [WMS02]

The design activities aim to improve the KAA gradually with respect to comprehensibility and knowledge acquisition efficiency. The initial candidate established in the exploration phase is the starting point of the evolutionary refinement driven by the agile process. Changes and extensions are discussed by domain specialists and knowledge engineers in cooperation.

Need for Change: Design activities have to be considered as a first class activity in addition to the actual knowledge acquisition process. Changes of the design of the current KAA are indicated by insufficiencies observed during the practical work with the system. Insufficiencies of this kind could be for example:

- bad interweaving of markup expressions and domain description
- markup expressions hard to read and understand
- missing domain description or modeling rationale
- documents that are overly large or have an inappropriate name
- long-winded navigation ways between strongly related contents

Therefore, all participants should watch out for these kind of deficiencies and report them to the team. Then a design improvement resolving that issue can be established in a joint discussion. The first two aspects of the KAA (support knowledge and arrangement of formal knowledge) can be evolved intuitively by reorganizing or extending the existing content. The third aspect

(syntactical shape) plays a key role in document-centered meta-engineering: The definition of a suitable knowledge markup language for the knowledge to be captured. The syntactical form can significantly contribute to the comprehensibility of the knowledge base. In the following, the design of new, appropriate markups is discussed in more detail.

4.4.2.1 Markup Design Principles

"I do see the possibility of a fascinating future in which we don't only agree that a good notation helps, but in which we actually teach how to design notations that are geared to the manipulative needs at hand." E.W.Dijkstra [Dij86]

Unfortunately, it is impossible to provide a precise procedure how an optimal markup can be constructed, as it is a highly creative task which strongly depends on the specific conditions at hand. Only general requirements and principles can be given to lead the design process. A custom knowledge markup needs to satisfy the following (technical) requirements: It has to allow for the unambiguous translation of the captured knowledge to the executable knowledge representation, c.f. Section 3.1.2. This includes the segmentation task separating the markup expressions from the other document contents. The goals of a designed markup are simple:

- Allow for intuitive and simple authoring and comprehending;
- Allow for simple and seamless embedding into the informal content of the documents;

In the following we discuss different markup design principles that are helpful for achieving these goals. We mention those being most relevant for the context of meta-engineering for document-centered knowledge acquisition from literature and our own experiences:

- **Include the User:** The main goal of the meta-engineering approach is to allow for active participation of domain experts in the knowledge acquisition process more easily. Therefore the simplest but also most important guideline is to commit to the characteristics of these users and their working context. Wile [Wil04] describes the necessity of understanding the role and background expertise of the people who will be using the designed language.
- **Simplicity:** For lowering the barriers it is reasonable and important to keep the designed markup languages as simple as possible. This naturally restricts expressiveness. However, a markup specifically designed for one limited purpose does not require high expressiveness.

"Limited expressiveness makes it harder to say things wrong."

M. Fowler, [Fow10, p.33]

Fowler emphasizes that language complexity, is the most notable source of errors when using DSLs (c.f. Section 3.1.2). Designing DSLs with minimal expressiveness, particularly tailored for a certain purpose avoiding unnecessary generality, will reduce error rates and therefore improve overall productivity. Wile [Wil04] even describes the benefit of aspiring for 80% solutions, i.e., DSLs that due to their simplicity only cover 80% of the

intended use case. If these 80% can be solved by a highly efficient DSL, there is enough time saved to deal with the remaining more complicated cases in a different way. This idea complies well to the idea of mixed-initiative knowledge acquisition.

- **Existing Domain Specific Notations:** In many subject domains own special terms and jargons are established. In combination with the jargon often structured or semi-structured notations are used within the documents shared by the domain experts in their daily work. For example, list-based or tabular notations can easily adopted or adapted to form a markup language. In this way a markup, providing minimal difference to what the domain experts are already used to, is obtained. It is reasonable to inspect the working environment of the experts, e.g., documents or domain specific software, to find inspirations for new custom tailored markups. This principle has also been discussed by Wile [Wil04] and Fowler [Fow10].
- **Readability:** Readability is a prerequisite to comprehensibility and therefore an important design goal.

“...clarity isn’t just an aesthetic desire. The easier it is to read ..., the easier it is to find mistakes and the easier it is to modify the system.”

M. Fowler, [Fow10, p.33]

Compactness requires less typing while verbosity often improves readability. Therefore, a reasonable trade-off between readability (verbosity) and compactness (little typing work) has to be drawn (c.f. Karsai et. al guideline 20 [KKP⁺09]). Software engineering studies have shown that programmers spend about ten times more time on reading code than on actually editing [Mar09]. No evaluation has been performed for knowledge acquisition tasks yet. However, considering all similarities that have been observed between software development and knowledge system development, it appears reasonable that a similar ratio holds for knowledge engineering. Therefore, it is reasonable to focus clearly on readability. Even more, the typing workload can be optimized by authoring support, e.g., code-completion techniques.

- **Consider Knowledge Maintenance:** In today’s rapidly changing world even after completion of a knowledge base the need for adaptations of the knowledge due to a changing domain (or specifications) might occur [KFK99, GGM95]. If the nature of these changes are known in advance markup design can be anticipated in a way that these particular changes can be performed easily. Assumed that a category of objects exists having a number of attributes and each object is described on its distinct document. Further assumed, that one of these attributes has to be modified due to changes in the domain. If the value for the attributes is defined on the object document, e.g. within a frame aspect markup, the change of this attribute requires to touch the documents of all the objects. Otherwise, if the values of the attributes are defined on an own page for all objects, for example using a list-based markup, only one single page needs to be touched significantly decreasing editing workload.
- **Consistency:** Make sure that keywords, operator signs, and delimiter signs are used consistently throughout all markups of the project. Inconsistencies of this kind complicate

the learning of the markup language for the user. This aspect is also mentioned by Karsai et. al in the guidelines 3 and 21 [KKP⁺09].

- **Possibilities for Comments:** In document-centered knowledge acquisition the markups are interwoven with the support knowledge. The latter can then be used for documentation and commenting. However, there are markups of different granularities. Table-based or list-based markups for example are forming large coherent blocks within the documents, making neatly attached commenting difficult within language design. Therefore, in markups of coarse-grained granularity possibilities for comments should be included. The markup design needs to incorporate a corresponding language construct allowing the parser to identify the comments as such, e.g., line-end comments (c.f. Karsai et. al, [KKP⁺09] guideline 18).
- **Simple Delimiters:** In many programming languages statements are delimited from each other by colons. This is a wide-spread practice and convention that every programmer is used to. However, some modern languages, as for example python⁴ or Groovy⁵). are designed to be more pleasant to the human eye. In the so-called *compact syntax* of Groovy no colons are necessary if each statement is written in its own line. Many people experience the resulting code more readable due to reduced syntactic noise [KGK⁺07]. This aspect is even more important when designing a language, which is primarily used by people without technical background. A line break has shown to be a simple to use and good looking delimiter that is well accepted and intuitively understood by users. Also Fowler [Fow10] rates the use of line terminators as valuable when working with non-programmers. Delimiters suggesting technical complexity, e.g., dollar or hash signs, should be omitted (if not beforehand being part of common existing domain notations). If spaces in object identifiers are excluded, even those can be efficient and good-looking delimiters.
- **Segmentation:** The ability for segmentation needs to be kept in mind when designing a markup language. It should be enabled in the least obstructive way possible by keeping the syntactic noise low. In most cases segmentation is performed by regular expressions matching the markup expressions within the content, but also other methods like custom heuristics or NLP technologies can be employed. A simple method is to start and finish a markup fragment by defined keys. These keys can then easily matched by a regular expression and the content in between can be passed to be processed as the actual markup source. More technical details about how segmentation can be performed are given in Chapter 5.
- **Natural Language Style:** It is tempting to try making the markup language look like a natural language as much as possible, e.g., by adding syntactical sugar. Fowler [Fow10] advises to abandon this endeavor. He indicates that it might put the user in a wrong context and can lead to misunderstandings. A markup language should be designed for high simplicity and readability but clearly be recognizable for the user to be a formal language. Otherwise, the user might think being editing free text. This is especially

⁴<http://www.python.org>

⁵<http://groovy.codehaus.org/>

important in DCKA where DSL content and natural language content is closely intermixed within a document.

Adhering to these principles forms a good basis for designing highly usable markups for the DCKA in the given project. However, in practice a designed markup cannot optimally satisfy each of the principles but trade offs need to be made, for instance considering *Simple Delimiters* and *Segmentation*.

Markup Design and Assessment: The design of the markup should be performed in close cooperation with the intended users, i.e. the participating domain experts. However, a knowledge engineer with basic knowledge about language design has to play the leading role in that process. We cannot expect the user to understand how and when a language works for knowledge formalization from a technical point of view [Wil04]. While involvement of the user is important, it can not be expected that she can come up with reasonable design propositions by herself. Hence, the knowledge engineer has to design some markup language candidates after carefully examining existing domain jargons, documents, and notations. When these candidates have been brought into line with the other markup design principles described, they are discussed with the user community. After the discussion, promising candidates are exemplarily used on parts of the current document base. While not yet being recognized by the system the markup can be inserted into the documents, possibly using a special font or a verbatim environment if provided. This allows to get an impression of its handling, readability, and integration with the content. As this specification and assessment process does not imply any implementation efforts on system level (yet), multiple candidates of different markups can be 'tested' this way at low workload costs.

When a markup has been assessed as appropriate by the involved contributors, a cost-benefit estimate should be made. The costs reflect the implementation workload of the markup potentially including authoring assistance, such as custom editors or code-completion. The benefit side is measured as how much improvement the markup brings to knowledge acquisition and comprehensibility compared to the previous formalization possibilities. One should also consider how often the markup will probably be used throughout the overall knowledge acquisition project. A markup of brilliant elegance will not repay if it there is only a dozen occasions where it can be employed. Nevertheless, a cost-benefit analysis can be difficult. The implementation costs usually can be overseen, but it is quite hard to forecast the benefit that the step will bring in the (far) future. Even if the advantages for the current situation and the subsequent development steps are quite limited, possible value can emerge considering long-term maintenance of the knowledge system. Eventually, in a couple of years another person, who is not that familiar with the knowledge base, might want to update some part. Then an optimized custom markup makes correct removal or introduction of a knowledge base entity much more intuitive. Also when the knowledge base might be restructured towards a new KAA design in the future, a concise and comprehensible markup will make life easier. However, we do not know in advance how many times the benefit of the discussed design step might arise. Often not even the overall life time of the system can seriously be predicted. This seems to be a general problem with the cost-benefit estimations on KAA changes in meta-engineering with document-centered knowledge acquisition. However, this problem can be considered similar to the question for structure improving

refactoring operations on the code base in common software engineering. One simple heuristic, which seems to be applied often in practice, is that structural improvements are considered profitable if their total costs are negligibly small when compared to the overall project expenditure. One argument for this heuristic is the worst-case consideration: If the improvement is implemented and for some reason one does not benefit from it in the future, the loss is limited to the implementation costs. On the other hand, if it is not implemented and the suboptimal structure on day causes major problems for the development process, the loss is potentially unlimited.

After the cost-benefit analysis has been evaluated positively, the markup can be included into the specification of the project's KAA. It should be described verbosely in the KAA document with abstract and concrete examples and for what parts it should be used. This introduction to the KAA ends this particular design task. The markup design activities are not only focused on the invention of entirely new markups. Often previously designed markups are just improved or extended.

"Indeed some people find that trying to describe a domain using a DSL is useful even if the DSL is never implemented. It can be beneficial just as a platform of communication."

M. Fowler, [Fow10, p.35]

A markup can also be helpful if it is not implemented (by now) for some reason. It might be that the cost-benefit estimate is negative or cannot precisely be rated at the moment due to missing information. Then the decision about the expense of the implementation task can be delayed to a later point when the relevant conditions have become clear. That complies to the general idea of the meta design principle, making important decisions as late as possible to incorporate all experiences made so far. The specified markup, even if not being processed by the system, still can be used by domain experts to express themselves. It forces them to formulate the knowledge in a precise and unambiguous form. That already is very valuable for a knowledge engineering process as knowledge in this form can quite easily be translated into available markup by the knowledge engineers. It is then reasonable to retain and maintain both variants closely together to be able to verify their correspondence easily. Using unimplemented markups in that way can be considered to form another intermediate step within the workflow of incremental formalization discussed in Section 3.2.1.

Continuation of the Meta-Design Process We expect that the meta-engineering process gradually improves the structure of the document base towards better comprehensibility. Improvements are triggered by deficiencies detected during work. This process will lead to a point where all participants feel to get along with the current structure. While this might allow for effective work at the current point, this in many cases will not suffice on the long run. It can be compared to the code structure and quality in software engineering projects. If code quality is at a state where the people that developed the code are 'getting along' with it, it might be still hard to work into it, especially for new participants. For the same reason why in software development a continuous improvement of the code quality is (should be) endeavored, it should also be pursued in DCKA. Experiences in projects of the past have shown that long-term maintenance of knowledge bases is a considerable challenge [KFK99, GGM95]. Especially, the step off of

a domain expert, which was the major contributor for a knowledge base (c.f. leaving expert issue [HA08]), can lead into the maintenance trap as it is hard for other experts to familiarize oneself with the legacy knowledge base structure. Hence, everyone should be aware that continuous design improvements should be pursued. Therefore, corresponding propositions need to be generated even if the current design already feels 'quite okay'. That will assert that new members will be able to enter the project at a late phase seamlessly and in that way guarantee long-term success of the created knowledge system.

4.4.2.2 Markup Aspects

The range of possible domain specific markups is highly diverse and therefore some categorization is desirable. There are several aspects that often reoccur, independently of the domain or knowledge representation. To establish a terminology to describe markups, we in the following introduce four basic markup aspects:

1. **Relational Aspect:** A relational aspect is given if the purpose of a markup is to establish relations between two or more domain objects. Examples for these kinds of relations are rules, logical axioms or formulas. Usually a large part of the knowledge base is formed according to this aspect, typically including the fragments determining the knowledge base reasoning behavior. Hence, the use of easily comprehensible markup and its embedding with support knowledge and documentation is particularly important. The following example shows a relational markup expression from the family ontology⁶:

```
1 CHAIN: hasUncle = hasParent o hasBrother
```

The example creates a property chain for the *hasUncle* property. It is introduced by the keyword "*CHAIN:*", which also serves for the segmentation in a line-based way. The actual markup expression after the keyword is a statement about the three entities *hasUncle*, *hasParent*, and *hasBrother*, forming a kind of ternary relation.

The second example shows a relations markup expression from an exemplary medical domain:

```
1 IF temperature > 39
2 THEN callDoctor = true
```

It contains a simple rule, which can be considered as a special kind of relation between the entities *temperature* and *callDoctor*.

Relational markup not necessarily form one relation by one distinct markup expression. The following markup example shows a table-based markup for heuristic rules from a car-fault diagnosis domain. For each cell one rule is generated with the column header entry as condition and the cell entry as value assignment for the row-header entry as action. Hence, multiple relations, which are interwoven in the markup, are generated by a

⁶<http://www.cs.man.ac.uk/stevensr/ontology/family.rdf.owl>

4 A Meta-Engineering Approach for DCKA

coherent markup fragment. This is one possibility how multiple relations can be defined in a compact way.

1	Clogged air filter	Ignition timing
2 Engine start = does not start	P5	N5
3 Driving = unsteady idle speed	P4	P1

2. **Frame Aspect:** A markup has a frame aspect if a coherent block of markup more or less completely describes an object by enumerating a number of its attributes. This presumes that the objects described belong to a category providing a common set of such attributes. These attributes, corresponding to *terminals* or *slots* [Min75] of Minsky's frame theory, are filled either by data type values (e.g., numbers, strings) or relations to other objects of the knowledge base. Hence, the role of the markup is to fill these slots for the object described in this frame. There are two major strategies of doing these assignments, either by *order convention* or by *explicit slot naming*. The latter is basically a list of attribute value pairs, as for example in a feature description of a technical device. In assignment by order convention an order of the slots is predefined and the values are defined in a sequence using a delimiter symbol. An advantage of the order convention method is that it is rather compact. Further, when typing markup expressions the user does not need to remember the (exact) names of the slots or their representative keywords or key-signs. The following example shows a frame markup expression with assignment by convention style:

```
1 DESCRIBE JochenR: Person, Jochen Reutelshoefer, 28.06.1981
```

It describes the entity with the identifier *JochenR* in a very compact way. Depending on domain and users in some cases the attributes of the assignments become clear to the reader by context. However, if the defined order is violated in the definition, senseless assignments are created by the system, possibly undetected by the compiler. In practice however, often not all attribute slots are necessary for all objects or at least not known from beginning. In this case of (multiple) optional slots the order convention method fails or require some kind of placeholders. Then the other method, explicit slot naming, is more convenient, being flexible with respect the definition of just a subset of slots and their ordering. The following example shows a frame markup using explicit assignment describing the same object as above:

```
1 DESCRIBE JochenR
2   class: Person
3   name: 'Jochen Reutelshoefer'
4   birth: '28.06.1981'
5   spouse: Stephanie
6   parent: ...
```

Here, order of the assignments is irrelevant and the risk of confusion is eliminated by the

cost of additional space. Another advantage is, that it allows for dynamic schemas. For the order convention-based markup the set of attributes are fixed at development time of the markup. The explicit slot naming method, if implemented accordingly, allows for dynamic extension of the attribute set, e.g., assume that the attribute *spouse* (or any other) is just defined in the document base at runtime. Hence, order convention-based frame markup can only be recommended to be used for very small and fixed attribute sets and when the assignments are clear from the context. Frame-based markup for instance is well-suited when for each of the described entities a distinct document is formed. There the frame markup and corresponding support knowledge, describing the object in an informal way, can be inserted.

3. **Definitional Aspect:** A markup has a definitional aspect if it is used to define new objects to the knowledge base, e.g., a disease, a symptom, or a technical component. A reasonable document-centered knowledge acquisition authoring environment should provide a compilation mechanism, which is able to verify object references for their correct spelling. In this case, markups that explicitly introduce new objects with a given identifier are required. In most cases the definitional aspect is combined with either a frame or a relational aspect defining additional knowledge being relevant for the knowledge base. The following example shows a definitional markup fragment:

```
1 CLASS Person
2 INDIVIDUAL Jochen
```

By these expressions the class *Person* and the individual *Jochen* are introduced to the system and can then be used in other markup expressions (e.g., relational ones). Without the definition the compiler should expect misspellings and report an error accordingly.

The second example shows the definition of a once-choice question with corresponding range:

```
1 QUESTION Relationship [oc] <unmarried, married, divorced>
```

4. **Implicit Knowledge Aspect:** A markup has an implicit knowledge aspect if part of the knowledge is encoded in the markup compilation process. This implicit knowledge adds further information to the knowledge encoded by the markup statement. As it is hard-coded within the compilation script, this knowledge is hidden from the users of the authoring environment. This aspect conflicts with the major principle of DCKA to have all knowledge accessible and modifiable at low barriers. Therefore, the aspect of implicit knowledge should only be used rarely and only for knowledge that is unlikely or rather impossible to ever change over time, such as basic laws of physics or geometry for instance. While the knowledge becomes invisible and unmodifiable, on the other hand one can save the effort to provide documents and markups for the implicit knowledge. When extensively used, this aspect pushes DCKA towards the strategy of indirect knowledge acquisition (c.f. Section 1.2.2.1).

4 A Meta-Engineering Approach for DCKA

One example for a markup with implicit knowledge aspect can be found in the context of the ESAT project (c.f. Section 7.1.4.2). There, display devices of various sizes are represented within the knowledge base by stating the screen width and length for each display. For the knowledge-based system also the area of the screen is relevant. As the knowledge of calculating the area of a rectangle from width and height is unlikely to change over time, it has been added as implicit knowledge aspect to the markup. Hence, for each display device defined by the markup, its area is calculated and attached to the knowledge object during the markup compilation process.

In most cases a markup does not strictly belong to one of the three categories but embodies multiple aspects at certain degrees. In particular, the frame and definitional aspect often appear in combination where an object is newly defined and subsequently its attributes are described in a frame style manner.

4.4.3 Implementation

The goal of the implementation activities is to put the specifications, added to the KAA during the design activities, into practice. Therefore, the current document corpus needs to be made compliant with the updated version of the KAA. In that way a custom-tailored knowledge acquisition tool is evolved. The implementation task can be divided into two distinct categories of activity: The *system level* performing changes on the authoring system implementation and the *content level* which is changing the structure of the documents.

4.4.3.1 System Level

The system level of the implementation efforts comprises changes, in most cases extensions, of the authoring environment software itself. Usually it implies the implementation of new markups, that have been specified during the design activities. This task typically includes parsing, syntax checks, translation, and further authoring assistance. As a reasonable strategy, at first basic editing assistance, in particular syntax check and highlighting, for new markups should be addressed to facilitate contributions using it. Then, compilation to the executable knowledge base should be integrated to enable testability of the knowledge. Finally, additional authoring assistance like special editors or code-completion can be added. As these kind of implementation tasks are a substantial part of the meta-engineering process, it is necessary to be able to perform them cost-effectively. In the customization approach of GUI-based knowledge acquisition tools, meta tools have been developed to allow for quick and simple generation of graphical user interfaces as discussed in Section 4.1. In this case of markup-based formalization, meta tools allowing for the declarative implementation of markup languages should be employed as far as possible. More details on how this can be achieved is given in Chapter 5.

Software Development Perspective: The structure of the meta-engineering process frequently demands development efforts for new markup features. Further, it is desirable to have newly created features available in the authoring environment as soon as possible. Hence, a feature by feature approach, as proposed by many agile software engineering methodologies (c.f. [BA04,

SB01]) including frequent system updates, is appropriate for this customization effort. A centralized web-based system architecture, as for example given when building up on a wiki-based system, makes deployment of system updates easy. The server system can be patched with the new version without the users and their client machines being strongly involved. While modifications on the authoring environment are being made there is not much need for the contributors to adapt to a new situation, as the documents remain unchanged by a system update. The modified system behavior will only take effect if the new markup feature is actually used.

A carefully designed system architecture can be robust against possibly occurring implementation errors dragged with a system update. As long as the basic view, edit, and save mechanism is safely provided by the system, the work of the contributors is not significantly disturbed even if some markup processing routine is erroneous. Having a stable, well-designed, extensible, and documented DCKA system to build up on strongly facilitates the system level implementation efforts.

4.4.3.2 Content Level

The content level implementation activities aim to adapt the document contents to comply to the KAA. Considering Figure 4.2 it is basically a transformation of the content along the three dimensions from the current state to the point described by the most recent KAA specification. Changes of the first dimension implies insertion or modification of the support knowledge. The dimension *arrangement of knowledge* performs component-based reorganization of the document contents according to the document structures defined in the KAA. For dimension *knowledge syntax* transformations on already existing formalized knowledge to the newly designed markup have to be performed. The KAA should define for which knowledge base components some new markup should be used. If parts of these components are already formalized using markup available previously, they have to be transformed.

For very small amounts of content these refactorings can be performed manually. However, if the document base has grown larger, refactoring of a major part needs to be supported by automated or semi-automated mechanisms. If a suitable data structure is used to capture the multimodal knowledge, methods from the general field of term rewriting [Klo92] can be applied. In compiler construction different term rewriting techniques have successfully been applied for code generation [AGT89]. Further, algorithms for effective tree rewriting [GMSZ08] and tree traversal have been established [vKV03]. Using rule-based rewriting technologies (c.f., XSLT⁷) allows to define the transformation process concisely in a declarative way. The content level refactoring task can be performed using these techniques with minor adaptations. In contrast to compiler construction, here also the non-formal content elements need to be considered for the transformation.

4.4.4 Knowledge Acquisition

The actual knowledge acquisition process is run in parallel to the meta-engineering process and aims to populate the knowledge base model. Except for the KAA specification, the meta-engineering process does not specify any more details about how the knowledge acquisition

⁷<http://www.w3.org/TR/xslt>

process should be organized. We recommend employ some agile process model (e.g., [Knu02, Bau04]) and to create modules that can be tested by itself. For any testable module corresponding tests should be created along as known from many software engineering practices [Mar09]. If the employed document-centered knowledge acquisition environment provides a continuous integration framework, a secure workflow based on regular automated testing can be established. In that way, regression during the distributed knowledge acquisition process can be prevented.

Document-centered knowledge acquisition with meta-engineering allows for knowledge acquisition activities from the point where the exploration phase is finished and an initial KAA is established. For early contributions however the tool customization is still in progress. Any time during the evolutionary refinement of the authoring environment, contributions to the document base can be made. Any contribution should be performed according to the current version of the KAA. As this is sometimes not trivial the users will not always be able to do so. Therefore, the role of the *knowledge gardener* is required.

Knowledge Gardening: During the knowledge acquisition activities it is the task of the knowledge engineering to supervise the contributions. In particular at an early stage, some users will have problems to contribute according to the KAA and with knowledge formalization in general. Hence, for any contribution the knowledge engineer has to check whether corrective action is needed. While he cannot verify the correctness of the domain knowledge on the subject level, he is able to rate whether this contribution complies to the contribution guidelines specified by the KAA. Further, he needs to participate within the incremental formalization workflow, i.e. writing or correcting markup expressions. This role can be compared to the *gardener* which is a social role in the context of general knowledge management often used in wiki systems [LFL05]. A wiki gardener is expected to fix errors and to rearrange content that is out of place. The extended gardener role of the knowledge engineer monitors compliance with the KAA and takes action in case of need. The knowledge acquisition process does not progress in a uniform manner. The gardening task demands a rather high attention in the early phase of the meta-engineering process, working with novice users and a not yet fully customized tool. The co-adaptive process however, leading to more experienced users and an optimized tool, will reduce the amount of cases where the knowledge gardener needs to take action. In the end, the domain experts are empowered to make many compliant contributions autonomously.

4.4.5 Conclusion

The meta-engineering approach proposes to run a meta-level process in parallel to the knowledge acquisition process which aims to customize the knowledge acquisition environment and document structure according to the project context. The intent of this customization is to achieve improved comprehensibility of the knowledge base to:

- allow for active participation of experts within knowledge acquisition more easily.
- enable long-term maintenance easily even if initial contributors are replaced.

The major goal of the meta-engineering approach for document-centered knowledge acquisition is not to make the knowledge engineer dispensable or to reduce his workload, as initially

intended by Musens work [GMF⁺03]. The meta-engineering process implies a number of conceptually and technically challenging tasks (markup design and implementation, content refactoring) to be performed by the knowledge engineer. However, all these tasks, as happening on a meta-level, do not require domain knowledge. They can be pursued applying computer science skills and techniques. A reusable tool infrastructure to support these tasks can be build up being independent of the domain. We claim that taking on the burden of these tasks pays off considering the two major advantages mentioned above. In principle, the meta-engineering approach to *some extent* decouples the constant need of cooperation of knowledge engineering *and* domain expertise. It enables the domain specialists to do some more contributions autonomously, bought at the cost of additional (information science related) tasks which can be solved by computer scientists without domain knowledge. This (partial) decoupling increases flexibility of the workflow and the overall productivity of the individual. The costs of the technical tasks imposed by the meta-engineering process are decisive for its profitability. The existence of mechanisms allowing for the efficient execution of these tasks are a requirement for rating the value of the meta-engineering approach. Therefore, the analysis and introduction of corresponding mechanisms is a further important aspect of this work. More details on these technical aspects are discussed in Chapter 5.

4.4.5.1 Language Complexity vs. Model Complexity

We have emphasized that the use of appropriate markups makes the knowledge easier to understand for domain experts. However, the inherent complexity of complex decision systems cannot be wiped away by that approach. A knowledge base usually contains a large set of components where each plays a particular role for the overall behavior. Some of these components are more, some are less hard to understand for the participating persons. The difficult components will retain their difficulty also being represented in the convenient markup language in documents with suitable support knowledge. Fowler emphasized the need for differentiation between language complexity and semantic model complexity [Fow10]. Language complexity describes the complexity of the syntax and the comprehensibility of how it is translated to the model, i.e. knowledge base. The semantic model determines the behavior of a given knowledge base at execution time, no matter how the knowledge base has been created, be it by DCKA or GUI-based. Hence, on that level language design cannot help to improve comprehensibility of the knowledge base. Achieving comprehensibility of the semantic model is an important task for further research but not in the scope of this work. Probably, this problem has to be addressed specifically for each single knowledge representation formalism, e.g., rule language or ontology representation language, by developing proper teaching mechanisms. However, appropriate markup helps to be able to create and maintain the knowledge base or model one tries to build. In practice, it is not always possible nor necessary that each contributor clearly understands all aspects of how the semantic model works. It is sufficient if the contributor knows how the knowledge base has to "look like" to show the desired behavior while not exactly knowing how the reasoning algorithm performs. Assuming for example a medical expert states that *Endocarditis* belongs to the class of *heart diseases*. He does not have to know exactly how the inheritance mechanism works or the transitive closure of the class hierarchy is calculated. It is within the responsibility of the knowledge engineer to deal with these topics when designing the knowledge base architecture.

For the medical expert in this case it is intuitively clear what this piece of knowledge means for the knowledge base. In DCKA the less complex knowledge components can be created by simple markups in a comprehensible way. Components of higher complexity are created by more expressive markups but without adding complexity to the overall handling of the tool. In sum, if it is clear what pieces of knowledge have to be created, then a suitable knowledge acquisition tool can help to achieve it, possibly using customized user interfaces.

4.4.5.2 Meta-Design: GUI-based Tools vs. DCKA

In principle, the iterative customization of a tool during the knowledge acquisition process according to the meta-design idea could also be applied for other authoring paradigms as for instance GUI-based knowledge acquisition tools. However, there are a number of aspects making the customization steps simpler and more smoothly for DCKA. Already at the beginning the question is with what kind of tool to start off. Either it is a general tool or an initial version of a customized tool is implemented. The former usually poses barriers even for basic contributions and afterwards makes smooth adaptation to a (completely) different tool hard. The latter corresponds to the process proposed by Eriksson shown in Figure 4.1 to be followed by a process of incremental tool redesign and adaptation. It implies expenditure of considerable implementation costs before experiences from knowledge acquisition can be included, which is bearing a high risk. In DCKA one can begin with an empty document set at no implementation costs. During the evolutionary meta design process a GUI-based tool can step-wise be adapted towards a customized knowledge acquisition environment. However, for every design change of the interface, implementation efforts have to be spent and time passes before it can be tried out in practice. The main advantage of DCKA in that phase is that any design changes can be tried out and assessed instantly (accepting that new markup is not yet processed). Hence, it allows for design and assessment sessions in real-time involving the users. In that way, the customization efforts can be driven more effectively as more design proposals can be rated in much shorter feedback cycles. To sum up, the DCKA approach is more suitable for an evolutionary meta-design process as due to its nature the start up is free of risk and the adaptation phase allows for more targeted customization due to 'real-time assessment'.

4.5 Extending Semantic Wikis

As already mentioned in Section 3.3, wikis, especially semantic wikis, provide a good basis for document-centered knowledge acquisition. For the intensive customization efforts demanded by the meta-engineering process a clear view about extending semantic wikis is helpful. In the following, we briefly discuss the most important aspects about extending semantic wikis, while more details can be found in [RLHB09].

We briefly outline a conceptual view on semantic wikis in general, followed by the discussion of the possibilities and challenges of extending semantic wikis. Figure 4.4 shows the three components of what we call the "knowledge pipeline" in semantic wikis. It shows the flow of the formalized knowledge from the contributing user to the consuming user through the Knowledge Formalization Component, the Reasoning Component, and the Knowledge Presentation

Component.

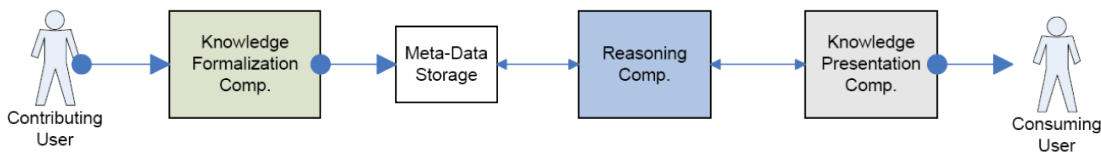


Figure 4.4: Sketch of the "knowledge pipeline" of a semantic wiki.

- **Knowledge Formalization Component:** The knowledge formalization component allows the user to formalize parts of the (textual) knowledge. This usually is done by markups, forms, or graph representations. The knowledge is extracted from this input and transformed into a target representation, which is commonly stored explicitly (e.g., in RDF) to allow for efficient reasoning. This transformation implicitly defines the semantics of the formalized knowledge having the target reasoner and the ontology in mind.
- **Reasoning Component:** A reasoning component uses the formalized knowledge created by the knowledge formalization component and is able to deduce higher-level information from it. While many semantic wikis employ an RDF-reasoner, there are several other reasoning approaches present, that are beneficial in particular application scenarios.
- **Knowledge Presentation Component:** This component describes the way how the additional functionality provided by meta-data and reasoner is used to supply the user with the right (high-level) information in a suitable form. This includes as an important aspect the visualization of the results. The reasoning capabilities can be used to provide semantic navigation, querying, rendering fact sheets, meta-data browser, and more.

These components together provide the additional value of a semantic wiki when compared to a standard wiki. In the following the possibilities of extending each of these components are discussed in more detail.

4.5.1 Dimensions of Semantic Wiki Extensions

In the following we discuss each of the three components and their potential for extension:

1. **Formalization Extension:** Given any methods (e.g., markup) to insert atomic formal relations, in a technical point of view any knowledge base can be created. However, the widespread employment of semantic technology is hindered by the formalization task being not simple and efficient enough [Wag06]. One way to counteract is lowering the barriers of knowledge formalization. The development of (domain specific) high-level markup languages with comfortable editing support can help to make knowledge definition compact, transparent and efficient. Another possibility for reducing the workload

of the domain specialists is the integration of (preconfigured) text mining methods, that propose formalizations based on the informal text content. Thus, the users only have to decide whether to confirm or dismiss a formalization proposition.

2. **Reasoning Extension:** Although basic reasoning engines are currently available there are still challenges with respect to scalability and expressiveness [KSV07] to be addressed. Further, there is ongoing research to cope with inconsistent knowledge, incompleteness and uncertainty [HvHfT05, MHL07, Kli08]. For some applications it will be valuable to replace or enhance the basic reasoning engine by a prototype from research work.
3. **Presentation Extension:** The challenge of these kind of extensions is to present the user the right high-level information in the right form at the right time (without overstraining him). These extensions must be specified according to the use-cases addressed by the intended application. One frequent application might be precompiled (possibly parameterized) use-case specific queries decorated by a GUI component for execution and having a visualization component attached for result presentation (e.g., table-based, graph-based, highlighted).

When designing an extensible semantic wiki architecture these three levels of extension need to be considered as shown in Figure 4.5.

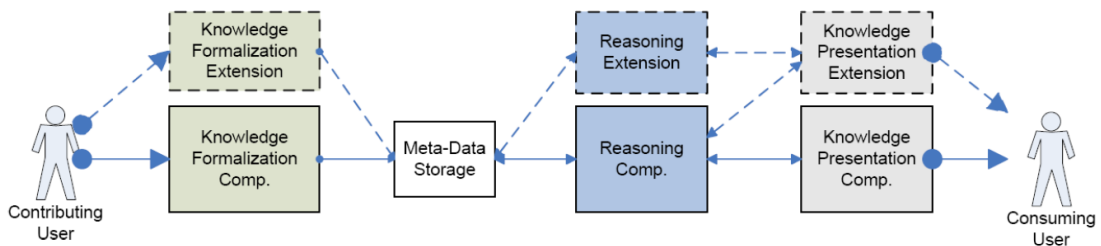


Figure 4.5: Sketch of the "knowledge pipeline" of a semantic wiki with extensions.

Examining the possibilities of the extensions on each level it becomes evident, that an extension on one component can extend the entire functionality of a semantic wiki system. Thus, the three components can be extended separately or combined. This leads to the semantic wiki extension space sketched in Figure 4.6. Assuming that the core semantic wiki system itself already provides some functionality in each component/dimension, extensions on the three dimensions can separately or combined contribute to the total functionality of the semantic wiki. Hence, an extensible semantic wiki architecture should allow for (independent) extension of these three dimensions. If the core functionality of the extensible semantic wiki nearly fits the requirements, single dimensions can be extended denoting "refining" extensions. Heavy-weight extensions along all three dimensions might have their own language, reasoning and presentation functionality.

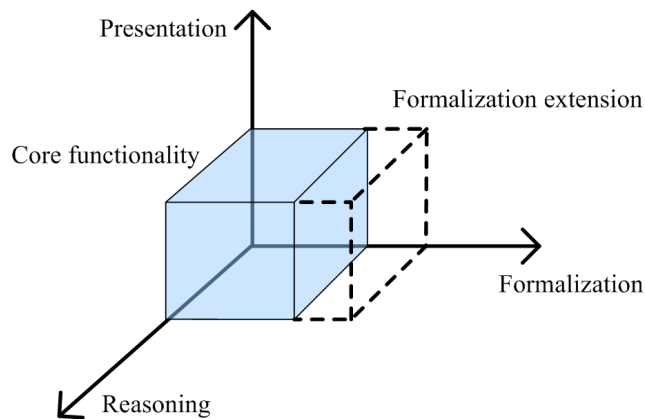


Figure 4.6: The semantic wiki extension space.

4.5.2 Decorating Semantic Wikis

As already mentioned in the context of meta-engineering we claim, that the possibility for extending semantic wiki systems precisely tailored to a application scenario regarding the domain, community, and use-cases is important. One can assume that there are many domains where semantic wiki technology could be employed beneficially in principle. One must not assume that every possible user is able and willing to get used to concepts like semantic wiki, ontology, RDF, SPARQL or DL-Reasoning. Nevertheless it is possible to create semantic wikis that allow for efficient knowledge sharing and use for these user groups at the cost of some customization. Assumed that an appropriate knowledge repository and reasoning engine implementation has been selected by the knowledge engineers. Then the customization demands emerging from the design activities of the meta-engineering process can be assigned to the categories formalization and presentation. This typical extension pattern that we call *decorated semantic wikis* is shown in Figure 4.7.

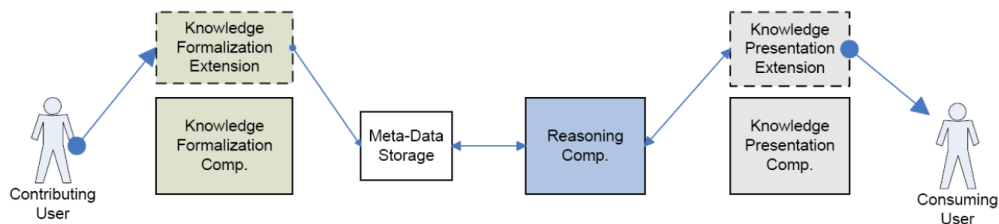


Figure 4.7: Extension pattern of a "decorated" semantic wiki application.

The core functionality for knowledge formalization and knowledge presentation, that is not customized for the project, should be hidden from the untrained user to reduce confusion.

4.5.3 Challenges towards an Extensible Semantic Wiki

In the following, we discuss the three most important aspects when designing an extensible semantic wiki:

1. **Basic Functionality:** In order to allow powerful extensions with advanced features with little implementation costs, it is necessary to decide what semantic core-functionality comes along with the basic semantic wiki architecture. Enabling applications that have to deal with large data sets and high user activity, requires a slim and scalable text-processing and reasoning engine. Reasoners that focus on processing of more expressive or inconsistent knowledge are often consuming more computational power and need to be included by an extension if necessary.
2. **Usability:** One of the most important reasons for the wide acceptance and success of wikis is their high usability and the low training costs for new users. Semantic wikis are bringing new possibilities to wikis and thus are inevitably adding some complexity to its usage. Adding these new functionalities to the wiki interface with the least mental overload is a critical issue in semantic wiki design. The core strength of the wiki approach being simple otherwise can easily vanish. In every case it is sensible to enable 'non-semantic' users to work like in a 'non-semantic' wiki with no adaptation, allowing them to discover single additional functionalities step by step. We propose to hide all of its advanced functionality at the beginning. Advanced features (e.g., fact sheets, meta-data browsers) can be added to pages using tags or configuration settings by more experienced users when necessary.
3. **Extension Mechanism:** Various extension mechanisms on the software engineering level are applicable to create feature-rich and extensible software. However, there are several challenges specific to the context of semantic wiki functionality. The mechanism should be able to support light-weight "refining" extensions in a very simple way and at the same time still allow for complex extensions (e.g., with own markup, meta-data representation, reasoners and result visualization). The use of a flexible plugin-framework is highly recommended. In general, an unpleasant issue in modular software engineering in general is the (programming-)language barrier. For technical reasons combinations of software components written in different programming languages are often insufficiently manageable and inefficient. Unfortunately, this poses some kind of barrier for employing various implementations of semantic technologies to an extensible semantic wiki architecture.

In Chapter 3 we already discussed the value of semantic wikis for the document-centered knowledge acquisition approach. The meta-engineering approach established the customizability and therefore extensibility as a major requirement. In this Section, we discussed the characteristics of semantic wiki extensions on an abstract level. Concrete methods, i.e., data structures and algorithms, to support the technical task of document-centered knowledge acquisition also considering the customization demands are discussed in the following in Chapter 5.

5 Techniques for the Implementation of DCKA

In Chapter 2 the general principles of document-centered knowledge acquisition have been discussed including the requirements of a corresponding authoring environment. Further, with the meta-engineering approach a refined development process model was introduced in Chapter 3. In practice, for the implementation and use of a document-centered authoring environment, especially when employing the meta-engineering approach, several technical issues need to be addressed. Some of them have already been pointed out in Section 3.2.2 and Section 4.4.3. In this chapter, these issues are discussed in more detail and solutions are proposed. While many problems can be solved using standard techniques, for some aspects custom-tailored techniques proved to be practicable. At first, we want to once more summarize the topics where technical tasks arise:

- **Parsing:** The source text, i.e., multimodal knowledge, within the documents needs to be processed. In a first step, the segmentation separates the markup expressions from the informal content elements. The markup expressions need to be parsed into a structured representation, which is the basis for further processing steps. In case of errors, helpful error messages should be generated.
- **Knowledge Base Population:** After parsing, the markup expressions need to be transformed and inserted into the knowledge base repository. As well as the parsing, this compilation step has to be performed after each document modification to keep the knowledge base up to date and ready for testing.
- **Refactoring:** The need for document-level refactoring can emerge in document-knowledge acquisition quite often. It can imply the reordering of content elements within documents but also transformation or merging of content element, for example transformation into another markup language. While this can be performed by manipulating the documents manually, automated mechanisms are strongly advised to keep the workload low.
- **Authoring Assistance:** As the use of markup languages for knowledge formalization is challenging for many users, suitable authoring assistance is important. This primarily considers autocompletion on typing, but also recommendation for corrections after typing. Sometimes advanced editing components for a markup are suitable using further user interaction methods, such as drop-down menus or drag-and-drop mechanisms.
- **Debugging:** A debugging mechanism is necessary for the development of complex knowledge bases. This aspect has already been discussed in Section 3.2.2.5. The challenge is to make the debug information for each piece of knowledge available in the document view, where it is defined.

In this Chapter, the focus is set on the first two topics, parsing and knowledge base population. This basis being set, the other activities can be implemented rather straight forward using standard techniques. At first, however, we compare the situation to the one given in standard software engineering.

5.1 Comparison with Software Engineering

All these technical challenges are not new in principle and well-studied in the domain of software engineering, e.g., compiler construction [ALSU06]. However, the basic conditions in DCKA show important differences when compared to those from software engineering. In the following, we discuss the most important differences to allow for a further valuation of the use of existing technologies known from software engineering:

- **Frequent Introduction of new Markup Languages:** General purpose programming languages are carefully designed independently of a concrete application project context. The implementation is a process that due to its complexity is often taking years of development before the first practically usable version becomes available. The widespread employment of these languages by software developers worldwide in various domains makes the high initial effort worthwhile. Further, once established programming languages usually do not change often, except for minor (backwards compatible) extensions. In the context of meta-engineering for document-centered knowledge acquisition the general setup is quite different. The process model actively encourages the introduction of new markup languages or the modification of existing ones while the project is running. Further, those languages are often project specific and therefore employed in a narrow context. The agility of the approach also requires the development to happen in short periods of time.
- **Simplicity of Markup Languages:** The markup languages have a much lower complexity when compared to general purpose programming languages. Only because of this, the recurring development of project specific languages in short time frames is possible. This is reasonable, as the simplicity is one of the major design objectives of custom-tailored project specific markup languages. Hence, a major difference to the challenges in classical compiler construction is to implement simple languages quickly.
- **Set Characteristic of Knowledge Bases:** A fundamental difference within the computational model of common (imperative) computer programs and knowledge-based systems is that in knowledge-based systems the control flow in general is separated from the 'program content' (knowledge base). Due to this declarative nature of knowledge, a knowledge base can be assumed as a set of (order-) independent knowledge slices. Therefore, dependency resolution is much simpler than in common imperative programs of sequential nature.
- **Incomplete Knowledge Bases:** If a software program discovers a compile error no executable version is generated. This is reasonable as it is not reasonable to execute a program that does not even compile. Similarly, a knowledge base that contains errors or

has missing fragments will not entirely fulfill its purpose. However, especially in an incremental evolutionary development workflow, it is often reasonable to compile and also execute a knowledge base even if parts are erroneous or missing [SS01]. In general, the performance of a knowledge base is rated by competency (to answer certain questions). Further, development usually extends the competence of the knowledge base. However, a correct subset of a large knowledge base can still provide enough competency for answering certain questions. Hence, it is reasonable to compile all non-erroneous parts to an executable knowledge base even if there are errors on some parts (possible areas of new development). That is possible due to the set characteristic of a knowledge base as discussed above. Therefore, one further requirement of knowledge compilation is to identify erroneous parts and then to continue compiling the remainder.

- **Multimodal Knowledge:** In document-centered knowledge acquisition the non-formal content elements within the source documents, while being compiled to the executable knowledge base, play a fundamental role for the development process. The document-centered authoring approach proposes strong inter-mixture of formal and informal content, such as plain text, figures, and charts (c.f. Section 3.1.3). While in programming the informal content, i.e. the comments, are marked by comment escape signs, it is important for the knowledge engineering approach that informal content does not need to be marked up in any way. This strategic requirement arises from the goal to provide low barriers for novice users and allow for incremental formalization (possibly using startup documents for seeding). Hence, formal content parts need to be identified in a first segmentation step, whereas in programming usually comments are detected (and extracted) in a first analysis step. This raises the *segmentation task* that separates the formal content, which needs to be compiled to the knowledge base, from the informal content. This task is not too challenging but necessary, as a good comprehensible structure of a document knowledge base usually implies a close inter-mixture of the formal knowledge parts with documentation. Segmentation has to be the first step in the knowledge compilation process chain.
- **Semi-Formal Languages:** Sometimes it is helpful to support processing of semi- or unstructured content, that cannot be described by a formal language. Then heuristic parsers and information extraction methods including natural language processing technologies can be employed. They are used to formalize the knowledge or to generate formalization propositions driving a semi-automated formalization workflow with user interaction. In these cases, the methods from classical compiler construction are not applicable.

Considering all these aspects, it is clear that the task of processing knowledge documents for document-centered knowledge acquisition significantly differs from the compilation of general purpose programming languages as known from standard software engineering. It therefore demands for the use of adapted techniques accordingly. The nature of the knowledge compilation problem allows for many markups to build knowledge compilers at rather low development costs. Therefore, in the following suitable parsing and compilation techniques are discussed.

5.2 Overview: Techniques Presented

The technical tasks for supporting document-centered knowledge acquisition, especially considering the meta-engineering approach, are quite diverse. In this chapter we introduce in detail a collection of non-standard methods that are well suited to support several of these tasks:

- **The KDOM Data Structure:** KDOM (Knowledge Document Object Model) is a data structure that is well-suited to capture multimodal knowledge. The capture of document-contents as KDOM is helpful for knowledge base population, refactoring, and debugging.
- **A Top-Down-Parsing Algorithm:** This parsing algorithm is able to create the KDOM data structure efficiently. Beside its capability to handle multimodal knowledge, it also provides the flexibility to handle frequent introduction of new markups, also at run-time by using of a plugin mechanism. For the algorithm multiple extensions are provided either improving runtime or simplifying parser implementations.
- **An Algorithm for Incremental Knowledge Markup Compilation:** The algorithm provides the check for terminology level errors and the knowledge base population in an incremental way. It is generic and therefore suitable for all kinds of knowledge markups and symbol level target knowledge representations.
- **A Meta-Model for Efficient Markup Implementation:** The previous techniques can be combined to a holistic approach of multimodal knowledge processing. We provide a method for the cost-efficient implementation of (new) knowledge markups in a declarative way by the introduction of a meta-model. It makes use of the KDOM data structure, the top-down parsing algorithm, and the incremental compilation algorithm.

5.3 Parsing of Multimodal Knowledge

Parsing is an important step in the analyses of source text as it allows to create a structured representation of the input string sequence. This structured representation can serve as a basis for a wide range of operations that are important in DCKA, including for example population of the knowledge repository, refactoring, authoring assistance, and debugging. The most common form of structured representation used in compiler construction is the abstract syntax tree (AST). Corresponding techniques and algorithms, i.e., syntax directed translation, have been studied thoroughly in the area of compiler design [ALSU06]. These techniques can not be applied in a straight forward way for parsing of multimodal knowledge, as the input source is not entirely written in one formal input language. A segmentation step, separating the markup expressions from the informal content, is required. In the following, we introduce a parsing mechanism that allows to perform the creation of a structured representation meeting all the requirements of document-centered knowledge acquisition.

5.3.1 The KDOM Data-Structure

As a core representation of the document contents we propose to use a data-structure called the *Knowledge Document Object Model* (KDOM). KDOM is a hierarchical breakdown of the docu-

ment as a double-linked typed tree, which is well-suited to support population of the knowledge repository, refactoring, authoring assistance, and debugging.

Definition 1 (KDOM) A *KDOM* is defined as set of nodes, where a node $n \in KDOM$ is defined as a tuple

$$n = (id, content, type, parent, children).$$

Thus, each node stores a unique id, some textual content segment of the input, a type (describing the role of the content), one parent node (with exception of the root node having no parent), and an (ordered) list of children nodes. A valid KDOM of a document is given if:

1. The text content of the root node equals the text content of the document.
2. The following constraints are true:
 - a) $textConcatenation(n.children) = n.text$ for all $n \in \{KDOM \setminus LEAFS\}$,
LEAFS being the subset of KDOM with an empty children set.
 - b) $n.text$ complies to $n.type$ for all $n \in \{KDOM\}$,
i.e., the text part of the node n can be mapped to the corresponding type. □

Having the document content as a KDOM data-structure, compilation and refactoring tasks can be performed by navigating the KDOM tree. Instead of analyzing the content string, the types can serve as an orientation by providing the context. In the following, we describe an algorithm that creates this KDOM data-structure for an input document using a given type system (*KDOM schema*), with a parser function attached to each type.

5.3.2 A Top-Down Parsing Algorithm

In this section, we introduce a top-down parsing technique that is not based on grammars and tokenization as known from standard syntax directed translation techniques. It is rather performing recursive top-down segmentation of the source text based on a schema of types. This *KDOM schema* is a tree of types defining the markup languages accepted by the system. Each type features a meaningful unique name, describing the role of the content element, and a parser function that is able to detect the syntactical element within a chunk of input source. The algorithm sketched in Listing 5.1 can be used to create a KDOM tree for a given input text according to a predefined KDOM schema tree.

```

1 parse (parentNode) :
2
3     forall (childType : parentNode.type.getChildrenTypes())
4         forall (string : getUnallocatedFragments (parentNode))
5             childrenNodes = childType.findOccurrences (string)
6
7         forall (childNodes : childrenNodes)
8             attachToParent (childNodes, parentNode)
9             childType.getParser().parse (childNodes)
10
11     forall (string : getUnallocatedFragments (parentNode))
12         createPlaintextNode (string, parentNode)

```

Listing 5.1: A recursive algorithm to build up a KDOM syntax tree.

5.3.2.1 Description

The *parse* function is initially called with a root node containing the entire the document content. Then all children types are searched for using the parser component provided by this type (*findOccurrences* in line 5). In case of detection corresponding children nodes are attached to the parent in line 8, allocating the respective text segment. For these children nodes the algorithm is called recursively to parse the next level of nodes underneath these children nodes by using the the parser provided by the respective types. For all unallocated fragments of the source text nodes of the type *PlainText* are created to make the content tree complete. Then the recursion terminates. The resulting syntax tree complies with the definition of the KDOM data-structure as specified above in Definition 1.

The KDOM schema plays a very important role in this process. A KDOM schema tree always has the type 'root' as root node. All available top level markups are child nodes of this root node in the schema, determining which kind of markups are recognized by the system. In that way, the first recursion level of the algorithm performs the segmentation task separating markup segments from informal segments, while creating plain-text nodes for the latter. For the further, the parsing process continues using the parser specified by the respective type.

The auxiliary functions used within the algorithm are explicitly described in the following:

- ***getChildrenTypes***: This function provided by types returns all its children types according to the KDOM schema in the order as defined.
- ***attachToParent***: This function attaches a newly created node to a parent node. The first argument is the new node and the second one is the parent node. The new node is inserted in to the children list of the parent. The order of the children in the list is important for obtaining a valid KDOM structure. As the nodes are not detected in the order the appear in the text of the parent, a child needs to be registered at the correct position in the list. This position can be found out easily iterating on the existing children and inserting before the first child, which represents a fragment coming after the one to insert.
- ***createPlaintextNode***: This function creates a node of the type *PlainText*. The first argument is the parent node of the plain-text node and the second one is the text fragment.

After its creation the plain-text node should be attached to the parent by using the *attachToParent* function.

- ***findOccurrences***: A type provides this function to call the parser component on the passed text sequence. For each detected occurrence a new node of this type is created. These nodes are then returned as a list.
- ***getUnallocatedFragments***: During the parsing process of a node children nodes of various types are detected one by one in some order. For each child type that is available, it is necessary to find the text fragments of the parent nodes, which are not yet consumed already by existing children nodes. The function *getUnallocatedFragments* collects these string sub-sequences of the parent node by iterating on the list of current children and collecting the gaps in between.

5.3.2.2 Performance

A strict theoretical analysis of the algorithm determines its complexity being in $\mathcal{O}([t * f * c]^d)$, with t as the average number of children-types, f as the average number of unallocated text fragments, c as the average number of detected nodes for a type and d as the depth of the schema/content tree indicating the recursion depth. This defines a complexity class appearing rather unfavorable and requires further consideration with respect to scalability. The average number of children-types t is constant and depends on the markup languages, i.e., the KDOM schema. In practice it turns out to be lower than ten in most cases. The number of unallocated text fragments f and number of detected nodes c do scale up with the size of the overall length of the source input but only for the first level of the recursion. After the segmentation of the first level, small segments are processed independently of the overall size of the document. The depth of the schema d is again depending on the markup language often being in a magnitude of ten in practice. The actual computation workload is caused by the calls of the parser components (*findOccurrences*). The algorithm does not make any constraints how this function has to be implemented for a certain type, as for example using a regular expression or a hand coded function. However, the runtime of a parsing function, no matter which way it is implemented, necessarily depends on the length of the text input. Assuming an average branching factor of b in the resulting tree, this implies that the average size of the node contents decreases by $1/b$ for each level, c.f. Constraint 2a of Definition 1. The parent node content size can be considered as an upper bound for the length of the input string for the *findOccurrences* function. In sum, while the count of parser function calls in worst case grows exponential with the depth, the input argument string length decreases logarithmic with b . However, a performance valuation of this algorithm based on theoretical analyses is insufficient, as the actual runtime strongly depends on the KDOM schema determining d and t and the structure of the input, determining f and c .

An empirical examination of the parsing process of the ESAT Wiki (c.f. Section 7.1) containing a knowledge base of 576 rules and a large number of test-cases distributed on 74 pages/-documents shows the following characteristics: the average number of found children nodes c is 1.60; the average number of *findOccurrence* calls ($f * t$) for one parse call is 1.46 (excluding root level); the average leaf depth d is 12 while the maximum depth found was 35. (The KDOM tree has a higher depth as the schema as a recursive schema has been used, c.f. Section 5.3.7.2.) The

average parsing time (using a standard laptop) is 18 ms per page while we found a maximum of 319 ms on a very large page (>1000 lines of syntax). This runtime results are practicable for use in real-world projects. With the computational power available today, this parsing approach does not cause performance problems presuming reasonable implementations of the parser functions within the types. Documents of exceeding size should, for the users' sake, be partitioned into multiple documents anyway .

Nevertheless, the speed of this algorithm can be significantly improved. In the next section an incremental version of this parsing algorithm is introduced. It is reusing the old KDOM tree, making the approach faster in case of the old version of KDOM tree being available.

5.3.2.3 Example

For clarity, in the following a comprehensive example is given, showing how a KDOM tree is created for a document using the algorithm of Listing 5.1 with a specific KDOM schema. Figure 5.1 shows an exemplary KDOM schema allowing for simple rules, comments, and relation tables to be processed explicitly. We assume that to each type the parsing function is attached, which is able to detect the occurrence of the relevant expressions within an input source sequence. Assume this KDOM schema is applied on the document shown in Figure 5.2, which is an excerpt from a knowledge base for car-fault diagnosis. The resulting KDOM tree is sketched in Figure 5.3. The numbers are indicating what parts of the document are included in the KDOM nodes. At root level eight segments have been created, including two comment segments, two rule segments, one table segment, and three plain-text segments. While the plain-text segments are leaves in the tree, the other segments are broken down further, according to their respective sub-type-hierarchy in the KDOM schema.

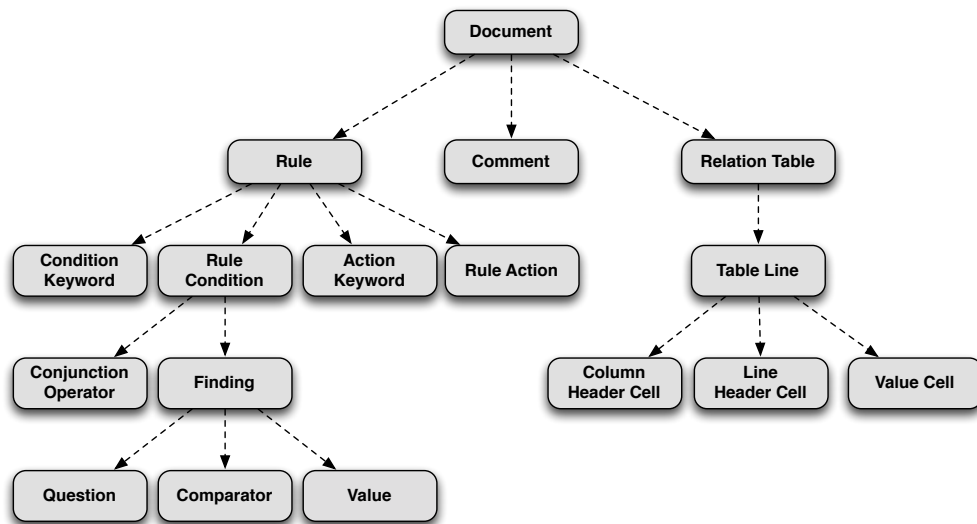


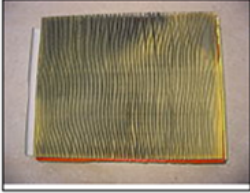
Figure 5.1: An example for a KDOM schema.

General# (1)

The (combustion) air filter prevents abrasive particulate matter from entering the engine's cylinders, where it would cause mechanical wear and oil contamination.

Most fuel injected vehicles use a pleated paper filter element in the form of a flat panel. This filter is usually placed inside a plastic box connected to the throttle body with an intake tube.

Older vehicles that use carburetors or throttle body fuel injection typically use a cylindrical air filter, usually a few inches high and between 6 and 16 inches in diameter. This is positioned above the carburetor or throttle body, usually in a metal or plastic container which may incorporate ducting to provide cool and/or warm inlet air, and secured with a metal or plastic lid.



clogged air filter

Typical Symptoms#

Typical symptoms, which indicate a clogged air filter, are the following *driving issues*: unsteady idle speed, weak acceleration, starting problems, (2) increased mileage (based on *average mileage* and *the currently measured mileage* or abnormal exhaust fumes.

Comment: *This rule should be moved somewhere else*

```
(3.1) (3.2.1) (3.2.2) (3.2.3) (3.2)
IF (Exhaust fumes = black AND Fuel = unleaded gasoline)
THEN Clogged air filter = Suggested (3)
(3.3) (3.4)
```

A typical starting problem which is connected to this problem is a barely or not *starting engine* in combination with a *starter* that turns over.

A clogged air filter can cause black exhaust fumes which will turn the color of the exhaust pipe to sooty black.

Relevant symptoms:

- engine start (4)
- exhaust fume color
- driving
- real mileage

(5)

(5.1)	Clogged Air Filter (5.2)
Driving = un (5.3) speed	+
Driving = weak acceleration	+
Starter = turns over (3.4)	(...)
Exhaust pipe color evaluation = abnormal	+
Exhaust fumes = black	+

Comment: *The table above needs some verification* (6)

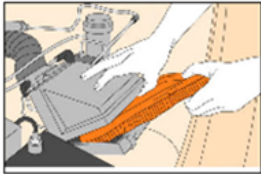
```
(7.1) (7.2)
IF Driving = weak acceleration
THEN Clogged air filter = Suggested (7)
(7.3) (7.4)
```

Repair Instructions# (8)

A clogged air filter needs to be replaced by a new one. Therefore the air filter housing has to be found. It will be either square (on fuel-injected engines) or round (on older carbureted engines) and about 12 inches (30 cm) in diameter.

After locating the housing the screws or clamps on the top of it have to be removed. Now the old air filter can be removed. At this point the housing should be cleaned from any dirt and debris with a clean rag.

Finally the new air filter can be put in and the top of the housing can be screwed or clamped back on.



air filter change

Figure 5.2: An example document from a car-fault diagnosis knowledge base.

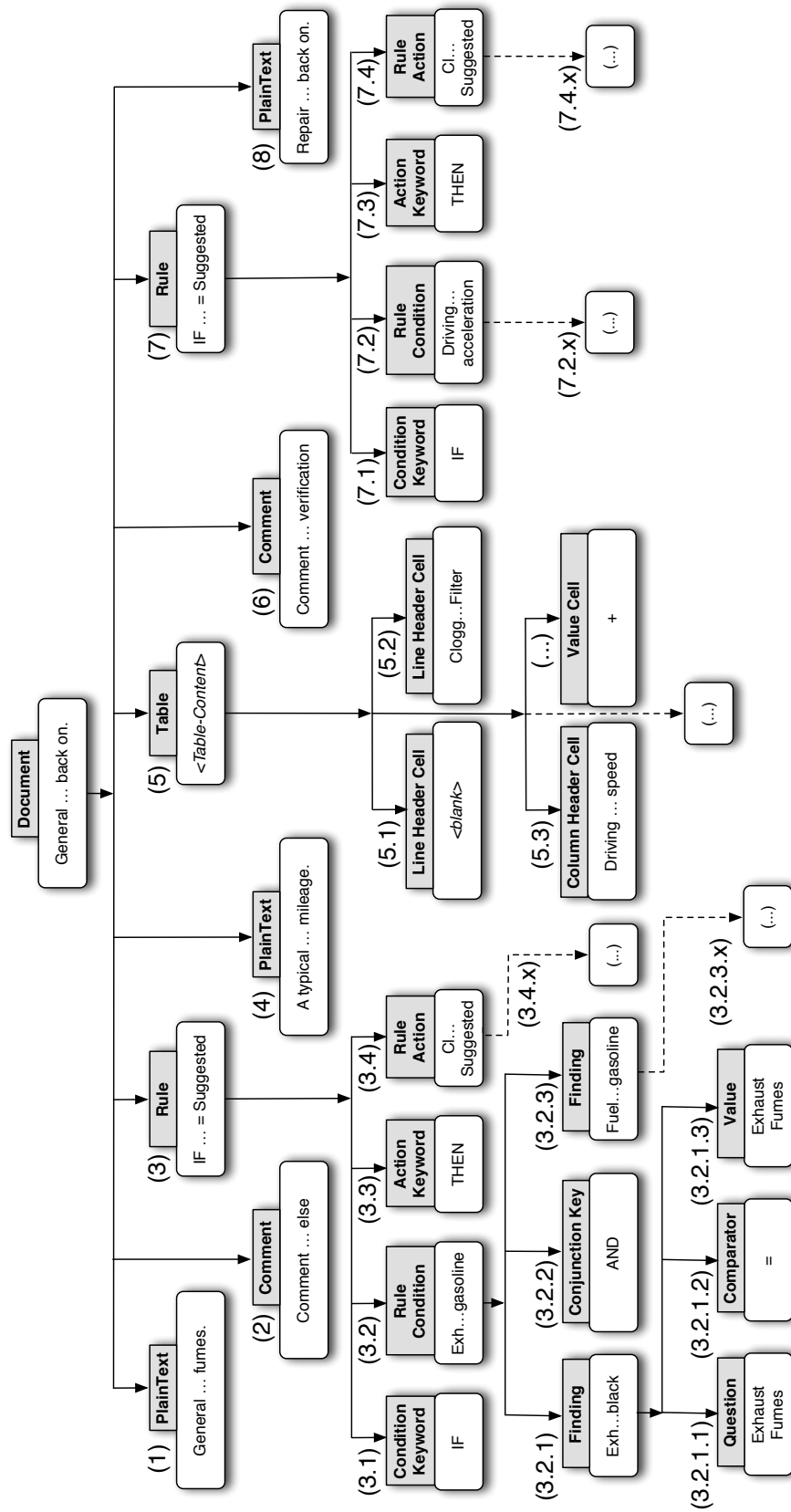


Figure 5.3: A KDOM tree for the document shown in Figure 5.2.

5.3.2.4 Discussion

The presented top-down parsing mechanism is valuable for the document-centered knowledge engineering approach as it compiles to the special requirements discussed in Section 5.1.

- **Segmentation:** Due to the nature of the knowledge documents containing multimodal knowledge, the task of segmentation is necessary to distinguish informal parts from parts demanding further processing (e.g., compilation into a knowledge base). For grammar-based parsers, fragments of informal content need to be treated in a special way during lexical analysis, if it is not explicitly marked up as comments. The top-down parsing algorithm performs the segmentation implicitly in the first recursion step. For contents that do not match any specific predefined types, the algorithm creates nodes of the type *PlainText* not affecting the further parsing process. For each kind of markup to be available at top level, its main type is registered in the KDOM schema tree as child of the root as shown in Figure 5.1. Then the algorithm will automatically include the markup for the segmentation.

- **Extensibility:** The KDOM schema completely defines the available markup language in the system. As the requirements for the markup languages often differ from project to project, the ability for flexible modifications is desired. With the presented approach sub-schema trees can be introduced to the system flexibly, even at runtime if necessary. With a suitable plugin mechanism new children types can be hooked up into the initial schema tree at any level. Grammar-based parser generators in contrast usually require the code generation of the parsers at compile-time. Modifying the markup language of a given system build using a standard plugin mechanism is not possible then.

- **Flexibility:** In document-centered knowledge acquisition contents of very different nature have to be processed. This potentially includes 'informal' markup languages with best-guess or semi-automated formalization. The necessary natural language processing or information extraction algorithms can easily be embedded due to the generality of the parser component of types. The *findOccurrences* function easily can call some NLP-component to create a corresponding syntax tree.

- **Compatibility:** The parsing approach can easily be combined with traditional grammar-based parsers to handle certain (sub-) markup languages. Therefore, a type is required that allocates a source segment complying to the sub-language. When the parse-function is called for a new node, the type determines the kind of parser that is used. While this can be the same top-down-parser again, also a grammar-based parser complying to the interface can be called, creating the syntax sub-tree for this node. In this way, the top-down parsing approach and the grammar-based parsing approach can easily be combined, possibly the first performing primarily the segmentation task and the latter parsing complex formal markup language parts.

- **Practicability:** As discussed above, performance is not a critical issue of this parsing approach. However, for the practical application, especially in the meta-engineering context, the development effort, which is necessary to implement a parser for a markup language as a KDOM scheme, is relevant. Experiences show that simple markup languages can be implemented as KDOM schema very easily raising only low development costs. Markup languages of increased complexity require more experiences within KDOM schema development and a good understanding of the parsing algorithm which is interpreting the schema. The limits of KDOM schemas are further discussed in Section 5.3.7.3. For languages of high complexity the development of a grammar-based parser should be considered alternatively. In Section 5.5, we present a language to enable developers to design KDOM schemas in a declarative manner to speed up and simplify the development process.

These aspects make the top-down-parsing algorithm introduced in Listing 5.1 a very valuable document-centered knowledge acquisition and the meta-engineering approach. To this basic algorithm still several improvements can be made, discussed in the following. Each extension leads to a minor modification of the basic algorithm. For clarity, always the entire modified algorithm is shown and the modifications compared to the original version are discussed.

5.3.3 Extension 1: Incremental Top-Down Parsing

In the agile knowledge engineering approach described in this work often small changes to the documents are made, for example in the collaborative incremental formalization scenario discussed in Section 3.2.1. When aiming at fast processing of these small changes an incremental approach for the parsing task appears beneficial, as most parts of the syntax tree will again appear in the new version. The basic idea of incremental parsing is to make use of the old syntax tree to create the new one by minimal efforts. Performance is not the only advantage of incremental parsing. After the parsing, in document-centered knowledge acquisition, the knowledge has to be inserted into the knowledge base repository. Instead of building up the entire knowledge base newly after each change, also an incremental knowledge base update is possible. Then a minimal change set of the content is required as a starting point of the knowledge base update algorithm. This information can easily be generated within the incremental parsing process. More details about incremental knowledge base update are given in Section 5.4.

In Listing 5.2, we present the incremental version of the algorithm discussed in Section 5.5.4, which reuses entire KDOM sub-trees of the old version of the document and additionally creates the resource-delta, that is required for the incremental knowledge base updating:

```

1 parse (parentNode) :
2
3   forall (childType : parentNode.type.getChildrenTypes ())
4
5     forall (string : getUnallocatedFragments (parentNode))
6       childrenNodes = childType.findOccurrences (string)
7
8     forall (childNode : childrenNodes)
9       subtreeNode = lookUpInOldKDOMTree (childNode)
10      if (subtreeNode != null)
11        mark (subtreeNode, REUSED)
12        attachToParentNode (subtreeNode, parentNode)
13      else
14        mark (childNode, NEW)
15        childType.getParser().parse (childNode)
16
17  forall (string : getUnallocatedFragments (parentNode))
18    createPlaintextNode (string, parentNode)

```

Listing 5.2: The incremental version of the parsing algorithm for creation a KDOM syntax tree.

5.3.3.1 Description

The algorithm is identical to the original one except for the inner loop starting in line 9. There, sub-trees are looked up in the old syntax tree and inserted into the new one if found in line 12. If no corresponding sub-tree can be found, the parsing algorithm is called recursively with the particular text section continuing the normal top-down parsing process. The information for resource-delta is created in the lines 11 and 14. Any section from the old syntax tree that is not marked as reused 'REUSED' is considered to be deleted.

Figure 5.4 illustrates this principle. The overall input sequence of the document is represented by the bar on the top. The part in white has been modified with respect to the previous version of the document, while the rest is identically retained. Further, the splitting operations, creating the nodes for the next level, are indicated by arrows. KDOM nodes created by the algorithm are symbolized by rectangles. The triangles represent sub-trees for the respective source that can be adopted from the KDOM tree of the previous version of the document.

The additional auxiliary functions of the modified algorithm are described in the following:

- **lookUpInOldKDOMTree:** This function searches the KDOM tree of the previous version for a node, which contains exactly the same text sequence. If such node is found, it is returned, *null* otherwise.
- **mark:** The function marks the passed node with a flag, in this case 'REUSED' or 'NEW'. These flags are not relevant for the parsing process at all and therefore could be omitted for the parsing task. The flags together are forming the resource delta indicating a minimal set of changes compared to the previous version of the document. This resource delta can be used by an incremental knowledge base update mechanism after parsing, discussed in Section 5.4.

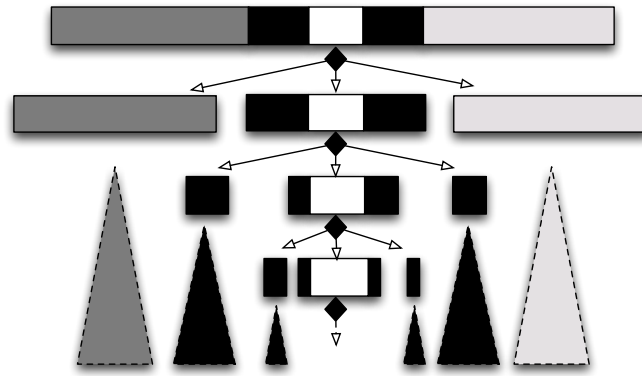


Figure 5.4: Illustration of the reuse of KDOM subtrees in the incremental parsing algorithm.

5.3.3.2 Performance

The reuse of whole subtrees makes this algorithm efficient, especially for small single changes of the document. For a rough complexity estimation, we assume the branching factor b of the syntax tree to be constant. When having n nodes in the tree the non-incremental algorithm (that is line 10 returns always *null*) requires n parsing steps P . Assuming a small single modification in the document the incremental process on any level performs only one parsing step and $b - 1$ look ups L . Figure 5.4 illustrates the case with $b = 3$. For a tree depth of $\log(n)$ that is an overall complexity of: $\log(n) \times P + \log(n)(b - 1) \times L$. The lookup in a typed parse-tree can be performed in logarithmic time with respect to the node count using path search or even in constant time if a hash table is maintained. Therefore, the result of this approximation is in logarithmic time in contrast to linear to the non-incremental process with respect to P . However, the parsing steps do not all require equal computation time. The first recursion step of the algorithm performing the segmentation on the root level of the document has a higher complexity. Then for $b - 1$ of the created nodes at each parsing step, the subtrees can be reused from the old syntax tree, providing significant speed up.

5.3.3.3 Discussion

This version of the algorithm will lead to exactly the same output KDOM tree as the original one. It guarantees to be at least as fast as the basic algorithm described in Section 5.5.4. If the document is modified completely, the algorithm will perform exactly as the original one. For small modifications in large documents it provides significant performance improvements by the reuse of sub-trees. Additionally, it delivers the resource delta for a incremental knowledge base update algorithm. No matter whether incremental knowledge base update is intended, this version of the algorithm can be recommended in general for its better performance.

5.3.4 Extension 2: Cardinality Constraints

The KDOM schema hierarchy basically declares which syntax fragments may occur as child elements of some expression, but not how many times. The basic algorithm the *findOccurrences* might detect an arbitrary number of occurrences for the respective type, e.g., all matches when using a regular expression. However, depending on the specification of the markup language to be implemented, often particular language elements are expected to appear exactly once (or in some other cardinality). This cannot be considered by the basic algorithm. However, these kind of cardinality constraints allow to define markups more simply and precisely, also helping to generate meaningful error messages. For this reason, we propose to extend the algorithm for cardinality constraints.

- **Min-Cardinality-Constraints** The parsing algorithm is designed to build up a syntax tree for the input in any case, and does not generate syntactical errors so far. As no cardinality is given the implicit cardinality is zero to infinite, which in fact makes the expressions optional. Usually a markup language contains important expressions which are required for the overall markup expression to make sense and for reasonable processing of the input source. To overcome this issue and generate helpful error messages, types of non-optional elements may be provided with a min-cardinality constraint. It describes how many times this element represented by this type needs to be present in the source code at least. This can easily be checked after the call of the *findOccurrences* function. If a constraint is violated an error message can be generated and attached to the parent node.
- **Max-Cardinality-Constraints** In principle, max-cardinality constraints can be treated analogously to the min-cardinality constraint described above. However, in contrast to the min-cardinality-constraints, the max-cardinality constraints could also be handled by the parser function. It could recognize (and prevent) these violations, i.e. too many occurrences of the particular syntax element, in advance. Therefore, the max-cardinality-constraints are not strictly necessary, but can simplify the implementation of the parser function. It can be convenient to check the max-cardinality constraints within the general parsing algorithm instead of in every parsing function.

5.3.4.1 Description

Listing 5.3 shows the modified algorithm. In line 12 the max-cardinality constraint is checked. In case of a violation, an error message is attached to the respective node and the recursive parsing process is not continued. Otherwise, the parsing process is continued normally. In line 17 the min-cardinality constraint is checked and an error message is attached to the parent node accordingly. In the following the additional auxiliary functions are described:

- ***violatesMinConstraint***: This function checks whether min-constraints have been violated. The min-constraints need to be accessible from the respective type as an integer value.

```

1 parse (parentNode) :
2
3     forall (childType : parentNode.type.getChildrenTypes())
4         List typeNodesTotal;
5
6         forall (string : parentNode.getUnallocatedFragments())
7             childrenNodes = childType.findOccurrences(string)
8             typeNodesTotal.append(childrenNodes)
9
10            forall (childNode : childrenNodes)
11                attachToParent(childNode, parentNode)
12                if (violatesMaxConstraint(childNode, childrenNodes))
13                    attachErrorViolatesMaxConstraint(childNode, type)
14            else
15                childType.getParser().parse(childNode)
16
17            if (violatesMinConstraint(childType, childrenNodes))
18                attachErrorViolatesMinConstraint(parentNode, type)
19
20    forall (string : parentNode.getUnallocatedFragments())
21        createPlaintextNode(string, parentNode)

```

Listing 5.3: The KDOM top-down parsing algorithm with cardinality constraints.

- ***attachErrorViolatesMinConstraint***: This function attaches an error message to the passed node indicating, that required elements of the corresponding type are missing. This message can be looked up and displayed by document rendering engine of the system.
- ***violatesMaxConstraint***: This function checks whether max-constraints have been violated. The max-constraints need to be accessible from the respective type as an integer value.
- ***attachErrorViolatesMaxConstraint***: This function attaches an error message to the passed node, indicating that too many elements of the corresponding type have been found. The message can be looked up and displayed by document rendering engine of the system.

5.3.4.2 Performance

The evaluation of the cardinality constraints does not require any significant amount of computation time. Therefore, the overall performance is not affected by this extension.

5.3.4.3 Discussion

This extension allows to generate syntactical errors during the parsing process. However, to provide sufficient user support, further error handling needs to be considered during the compilation process. Further, this extension is important for the extension discussed in the next section.

5.3.5 Extension 3: Backtracking for Top-Down Parsing

The task of the parser function *findOccurrences* is to determine whether (a part of) some content belongs to the respective type. The basic parsing algorithm 5.5.4 works deterministic as a taken step cannot be retracted. Therefore, a decision at a particular step may become a difficult task for non-trivial markup languages. It might require a more detailed analysis of the source before the decision, whether the source content complies to this type, can be made. This analysis often includes parsing-like analysis activities in advance, before the actual deep parsing is performed in the subtree. It implies that the parser functions on higher level have increased complexity also doing tasks which are intended to be performed by the types on the lower levels. This problem can make the implementation of markup languages as KDOM schemas awkward as the parser functions on different levels may become somehow redundant. A strategy to solve this problem is backtracking. Applying this strategy to KDOM parsing releases the parser functions on the higher levels from the duty of taking final decisions. Rather, the higher level type allocates the source content causing the continuation of recursive top-down parsing of the successor types and defines a mark for backtracking. Then it is the task of the successor types to decide, whether the source content really complies to the language element. For that purpose, the constraint mechanism introduced in Section 5.3.4 can be employed. Types which are mandatory for the respective languages element can be provided with a min-cardinality constraint. A violation of this constraint will trigger the backtracking mechanism up to the backtracking point defined previously. On that level the node allocation is then removed and the content is passed to the next registered type continuing the normal parsing process. Therefore, the parser function is turned into a boolean function that returns *false* only in that cases, when a min-constraint is violated.

5.3.5.1 Description

Listing 5.4 shows the modified algorithm. To implement the backtracking mechanism, the parsing function is designed to return a boolean value. That value tells whether the analyzed text segment complies to the expected pattern of the KDOM-schema subtree. If the value *nodeMatches* is *true* in line 15, the algorithm behaves exactly as the basic version discussed earlier. If constraint violations have been found in the subtree and *nodeMatches* is *false*, then the current type is checked, whether it is a backtracking point in line 19. Types which are designed to make use of the backtracking mechanism need to be marked as such on implementation, that is *isBacktrackPoint()* has to deliver true then. If it is a backtracking point, the node is not attached to the parent, but stored for potential later use. Then the next type will be tried out for this source fragment. If it is not a backtracking point, the backtracking will continue. Therefore, the node is attached to the parent and *backtrack* is set to *true* in line 23. This will lead to termination of the current call with *false* in line 40. The min-cardinality constraints are checked in line 26. In case of violation *false* will be returned, initiating the backtracking process.

Within the different variants of the top-down-parsing algorithms several auxiliary functions are used. They are described in the following:

- ***getNextUnallocatedFragment***: This function works similar to *getUnallocatedFragments* used by Listing 5.1, but only returns the first unallocated sub-sequence detected.

```

1 boolean parse(parentNode) :
2
3     tmpNodes
4     backtrack = false
5
6     forall (childType : parentNode.type.getChildrenTypes())
7         childrenNodesTotal
8
9         forall (string : getUnallocatedFragments(parentNode))
10
11             childrenNodes = childType.findOccurrences(string)
12             childrenNodesTotal.addAll(childrenNodes)
13
14             forall (childNode : childrenNodes)
15                 nodeMatches = childType.getParser().parse(childNode)
16                 if (nodeMatches)
17                     attachToParent(childNode, parentNode)
18                 else
19                     if (type.isBacktrackPoint())
20                         tmpNodes.add(childNode)
21                     else
22                         attachToParent(childNode, parentNode)
23                         backtrack = true
24
25
26             if (violatesMinConstraint(childrenNodesTotal))
27                 attachErrorViolatesMinConstraint(parentNode, type)
28                 createPlaintextNode(parentNode, parentNode.text())
29                 return false
30
31
32 fragment = getNextUnallocatedFragment(parentNode)
33 while (fragment != null)
34     tmpNode = getTmpNodeForFragment(tmpNodes, fragment)
35     if (tmpNode != null)
36         attachToParent(tmpNode, parentNode)
37     else
38         createPlaintextNode(fragment, parentNode)
39     fragment = getNextUnallocatedFragment(parentNode)
40
41 if (backtrack) return false
42 return true

```

Listing 5.4: The KDOM parsing algorithm with backtracking.

- ***isBacktrackPoint***: This function determines whether a certain type has been declared to be a backtracking point by its developer.
- ***text***: This function returns the text of a node.
- ***getTmpNodeForFragment***: Looks up whether a section had been created and temporarily been dismissed during the backtracking process. As no valid match has been found for the fragment, the latest candidate can be used and provided with meaningful error messages accordingly.

5.3.5.2 Performance

The backtracking mechanism adds additional complexity to the algorithm, as any type can be tried out whether the content matching the sub-schema or not. In practice, the runtime strongly depends on the concrete implementation of the markups. Runtime can become an issue when many markups, which use the backtrack mechanism with the min-cardinality constraints at considerable depth, are competing. This situation, however, can be avoided by design. For example, the mechanism should not or only sparsely be used on the top level (segmentation level). In practice, the number of competing types/markups is often low (<4), even one in many reasonable cases. The backtracking dept, which is the depth where the min-constraints are located, often is about two to four. In these cases, despite of an exponential complexity class, the additional computation load is insignificant.

5.3.5.3 Discussion

The trade-off provided by this strategy is, that the KDOM schema parsers become easier to implement at the cost of additional runtime. It was already pointed out that parsing runtime usually is not a critical issue in this context. Therefore, in principle one can benefit from this strategy, reducing the complexity of the parsing functions. However, when handled careless the complexity of the algorithm can cause serious runtime problems. Hence, a good trade-off position needs to be taken. A reasonable strategy is to use it in combination with the incremental parsing mechanism and then to have no backtracking types on the segmentation level. Competing types at lower levels in general are rather rare, so escalating backtracking is prevented. Further, in practice it is important to check that for each min-cardinality constraint defined a corresponding backtracking type has been defined. Otherwise the algorithm will backtrack back until to the root node creating no reasonable KDOM tree.

5.3.6 Implementation Architecture

In the following a possible architecture for the implementation of a top-down parsing mechanism for a document-centered authoring system is outlined. Figure 5.5 shows a class diagram that describes the structure of the required components. The class *Node* allows to represent the nodes of the final KDOM tree. Each node is provided with a unique *id* for identification. A node can have multiple children nodes to form the tree. To each node also its father node is associated,

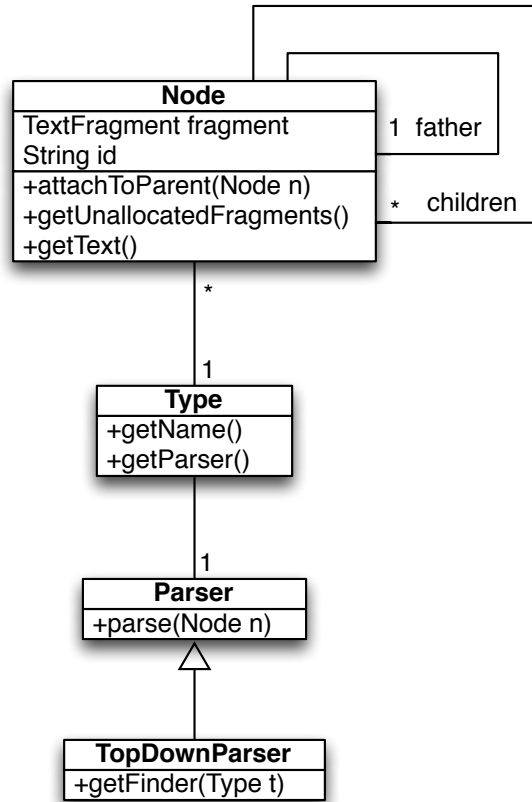


Figure 5.5: Class diagram for the necessary components for top-down parsing.

except for the root node (c.f. Definition 1). The *TextFragment* at least needs to store the offset of the text with respect to the father node and its length. This allows for the implementation of the method *getText()* returning the actual text sequence represented by this node. Also, for each node exactly one type, represented by the class *Type*, is attached. To distinguish between different types, each type has a name return by the method *getName*. It is also possible to create a new subclass for each new type. Then the class name can be returned by *getName*. Also hierarchies of types can be created, allowing for similar types to be treated in a similar way by use of polymorphism. The type of a node provides an instance of the interface *Parser* by the *getParser()* method. That parser will be used to parse the node content to create the KDOM subtree as discussed previously. The *TopDownParser* is one implementation of the *Parser* interface, which performs top-down parsing based on KDOM schemas as illustrated in Listing 5.1. For markups that are parsed in a different way, e.g., using syntax-directed parsers, additional implementations of the *Parser* interface can be created and attached to the second level type of the respective markup. The other methods have already been discussed in Section 5.3.2.

The doubly linked tree of *Node* objects plays a fundamental role in a system of this kind. It forms the basis for various kinds of important activities such as knowledge repository population,

refactoring, content rendering, debugging, or the support for special editors. Therefore it is reasonable to provide a small API of methods for navigation and search in this tree, which can be used by the implementations of all these activities. Typical tasks are the retrieval of a node by *id*, the search of all nodes of a certain type in a sub-tree, or for a given node the retrieval of the nearest ancestor of a given type.

For the activities, which are actually modifying the document content, being typically refactoring and special editors, it is important that the modifications are not performed directly on the nodes structure. Even if the changes are performed in a way that keeps the KDOM tree consistent (c.f. Definition 1), it is difficult to guarantee that the tree structure after modification is the one that would have resulted from parsing the new document content. Therefore, it is strongly recommended to assemble the entire document content string including the modification and then generate a new KDOM tree by applying the parsing algorithm on the full document content. Especially when using the incremental parsing algorithm introduced in Section 5.3.3, the computation workload by parsing should be limited.

5.3.7 Tutorial: Implementing Markups as KDOM Schemas

In Section 5.3.2 the top-down parsing algorithm has been introduced. The comprehensive discussion of its different versions aimed to support the development of a document-centered authoring environment core as outlined in Section 5.3.6. This core then is reusable for any kind of knowledge acquisition projects in any domain. Therefore, the implementation of such kind of core is rather seldom. A technical task appearing much more frequently, especially considering the strategy of the meta-engineering approach discussed in Chapter 3, is the implementation of new markup languages for an existing document-centered authoring environment core.

As already mentioned, the KDOM schema and the top-down parsing algorithm not only provides a convenient way for the segmentation, but also allows to implement actual markup parsers. While it needs some practice, the implementation of markup as KDOM schemas is quite easy for simple markups. However, some basic computer science techniques are required, for example the use of regular expressions [Fri02]. The implementation of a markup should be done by a knowledge engineer with computer science background. It is not intended nor necessary that a domain expert without technical background performs this task. While domain experts have to be involved closely within the design of the markup, its implementation can be performed by technically experienced project participants in a secluded way.

In the following, a small tutorial is given by showing several examples of markup implementations as KDOM schemas. Then recursive KDOM schemas are introduced, allowing the implementation of more expressive markups. Finally, the limits of KDOM schema parsing are discussed.

5.3.7.1 Introduction by Examples

Example 1 For the first example, we assume that we want to implement a simple markup allowing to define classes, as for example in ontology engineering. A new class could be introduced by the keyword 'CLASS'. For the segmentation we assume that this class declaration is written at the beginning of a line and that it is terminated by the next line break.

Solution A A possible KDOM schema implementation of this markup could look like in Figure 5.6. For each type within the schema a 'Finder' is attached, defining the parser function detecting the occurrence in the text. The *Root* type is always present as a starting point of the schema. It has a predefined finder attached called 'ALL' indicating that this finder captures all available source text into one node. The actual implementation of the example markup starts with the type *ClassMarkup* being a child type of *ROOT*. It is provided with a finder based on a regular expression, which detects the expressions as defined above within the text. The type *ClassKeyword* is defined as a child type of *ClassMarkup*, provided with a regular expression finder matching the keyword including the subsequent space characters. The second child type (order matters!) of *ClassMarkup* is the type *ClassTermName*. It makes use of the 'ALL' finder again, as anything that is left after the keyword is supposed to belong to the term name of the defined class.

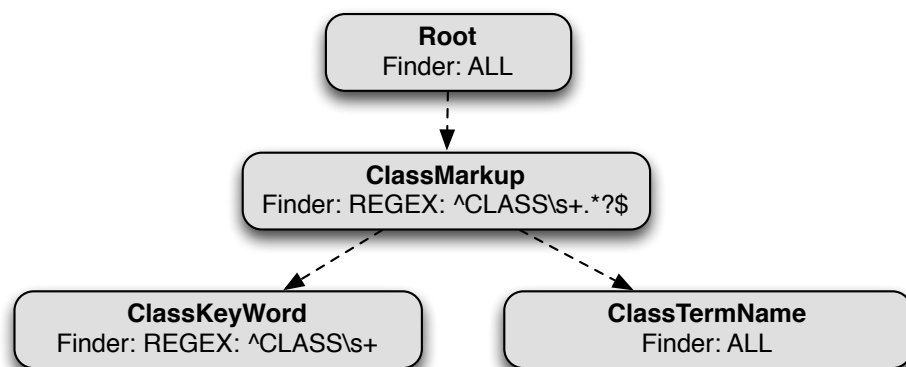


Figure 5.6: A KDOM schema implementation for the exemplary class markup.

To illustrate the result of the markup implementation, it is applied to the following exemplary document. It defines the classes *Person* (line 2) and *Disease* (line 4) and in between arbitrary documentation content has been stated.

```

1 Lorem ipsum dolor
2 CLASS Person
3 consetetur sadipscing
4 CLASS Disease
5 sed diam nonumy
  
```

Applying the algorithm from Listing 5.1 (or either of its derivatives) with the KDOM schema of Figure 5.6 will result in the KDOM tree shown in Figure 5.7. For every node its type is shown in the gray box, while the text content is shown in the white box underneath. Line breaks are indicated by '\n' and (invisible) space characters by underscores. This structured representation

of the document is helpful for example for the population of the knowledge base, for highlighted rendering of the document, or for refactoring.

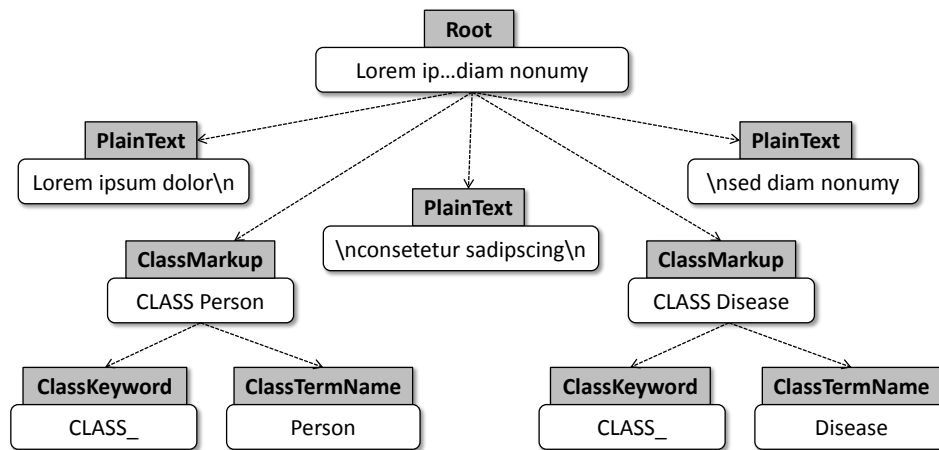


Figure 5.7: The resulting KDOM tree for the example input document with solution A.

Solution B There are usually many solutions for coming up with a KDOM schema to parse some markup. One flaw about solution A is, that the keyword 'CLASS' appears multiple times, once in the finder of the *ClassMarkup* type and once in the one of the *ClassKeyword* type. This is undesirable redundancy imagining the keyword has to be changed one day. However, this problem tends to appear often in KDOM schema parsing as often syntactical patterns need to be reused in the sub-types. To overcome this issue we propose the strategy of sharing regular expressions. It makes use of the possibility, that regular expressions deliver different capturing groups [Fri02].

In this way, another solution can be created shown in Figure 5.8. On the right, the regular expression is explicitly defined with the name *ClassRegex*. It is used with the type *ClassMarkup* with the capturing group 0, which refers to the entire match. Further, it is also used by the type *ClassTermName* with the capturing group 1, referring to the first group defined by round brackets. In this case, no explicit type for the keyword is required.

The output KDOM tree of solution B is shown in Figure 5.9. It is almost identical with the KDOM tree produced by solution A. The only difference is, that the nodes containing the keyword 'CLASS' now are nodes of the type *PlainText*. Another difference is the order in which the nodes have been created. While this is not important for the outcome, this aspect is important to understand the overall KDOM schema parsing process. In solution A, first the nodes containing the keyword are created, followed by the node for the term name. During the execution of solution B in contrast, first the nodes for the class term name are detected and created, before the remaining keyword is put into a plain-text node. This should not make a difference

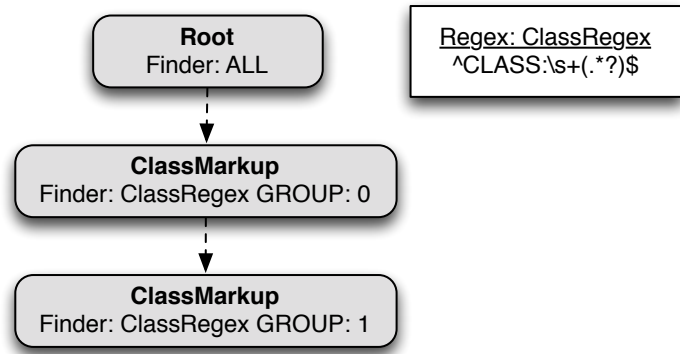


Figure 5.8: Solution B: An alternative KDOM schema implementation for the class markup.

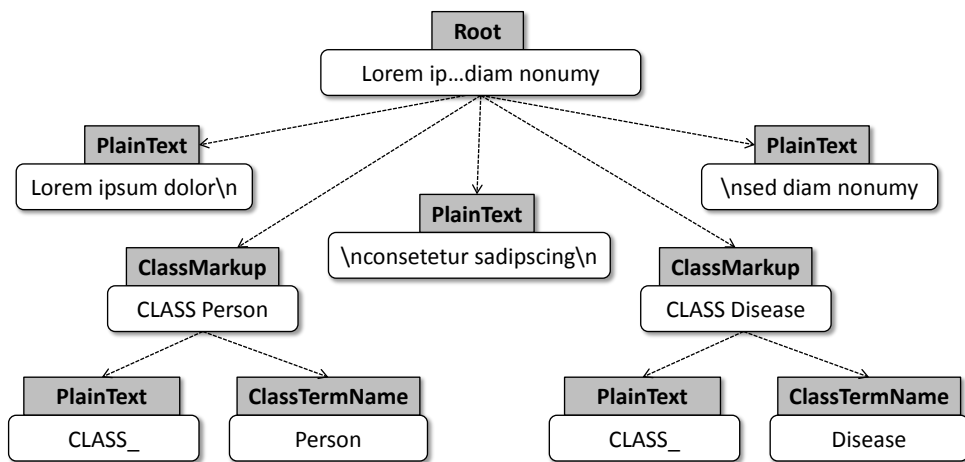


Figure 5.9: The resulting KDOM tree for the example input document with solution B.

for the further processing. It is only important that there are typed nodes for *ClassMarkup* and *ClassTermName*.

Example 2 In the second example the parsing of a simple comma-separated list is illustrated. We assume that we aim to define a list of abstract terms, using the keyword 'Abstract:'. This can be implemented by the KDOM schema shown in Figure 5.10.

The KDOM schema applied on the following exemplary document results in the KDOM tree shown in Figure 5.11. The parsing of lists of arbitrary length works with the rather simple KDOM schema because the algorithm always detects all occurrence of a type and creates the corresponding nodes. Hence, at first the nodes for all separators are created, then for all the remaining text sequences nodes of the type *TermName* are created.

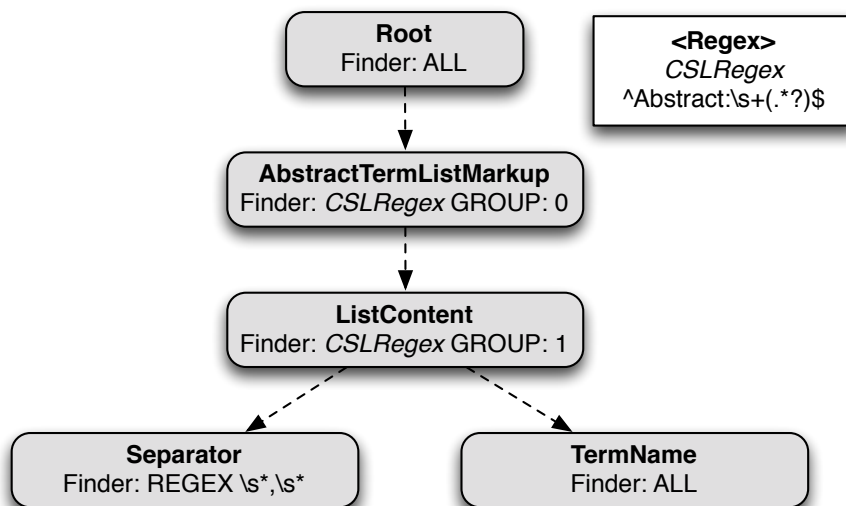


Figure 5.10: A KDOM schema for parsing a comma-separated list of abstract terms.

```

1 Lorem ipsum dolor
2 Abstract: term 1, term 2,term 3
3 consetetur sadipscing
4 Abstract: term 4
5 sed diam nonumy
  
```

The two examples just discussed illustrate the basic elements of markup implementation as KDOM schemas. While the examples work as illustrated, in practice one has to take care of several more issues to achieve reasonable robustness. For example, trailing white spaces following a term name before the line break need to be trimmed away explicitly as causing potential problems with the compilation process otherwise. If separators, as the comma in example 2, occur in term names, the given implementation will fail. Usually, this is treated by quoting the term name, which again needs special extension of the schema. These issues however, can be handled by providing some auxiliary functions for trimming and quote scanning.

5.3.7.2 Recursive KDOM Schemas

The examples shown so far address very simple markup languages. According to Section 5.3.2 the depth of the resulting KDOM syntax tree is predefined by the depth of the schema tree. This obviously poses strong restrictions on the complexity of languages that can be represented. In particular, languages defined by recursive grammar rules, e.g., a grammar rule where the non-terminal symbol from the left side also occurs on the right hand side of the rule, cannot be represented so far. These kind of languages, however, often play an important role in knowledge

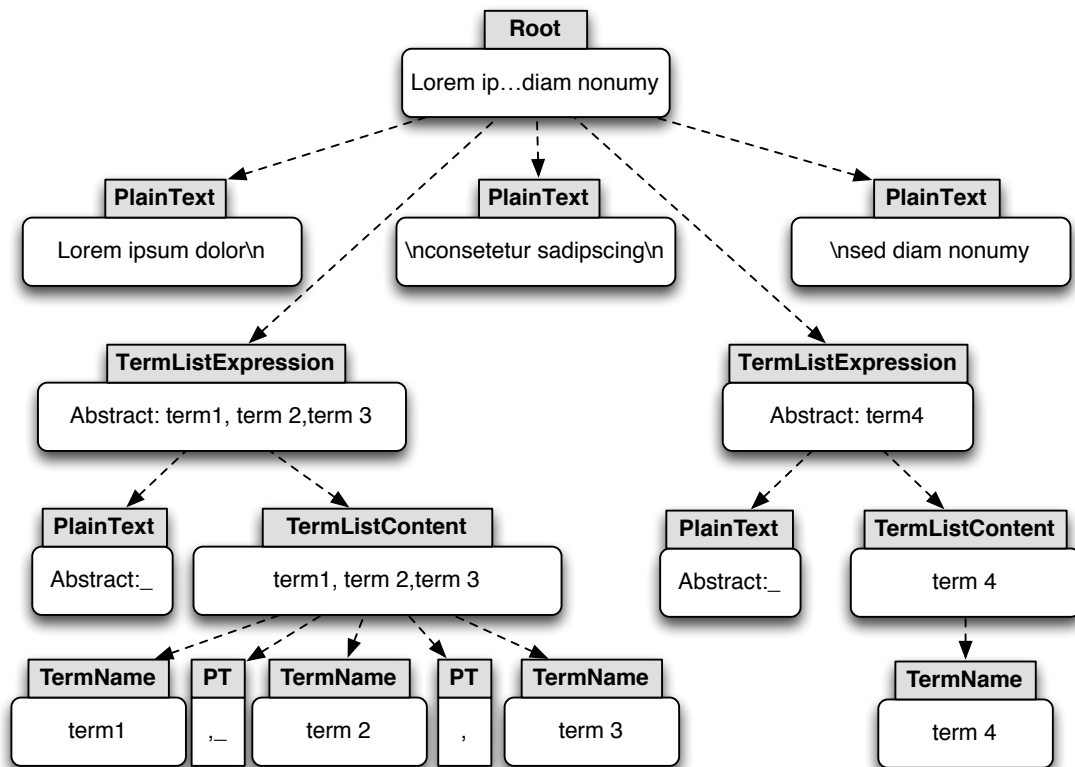


Figure 5.11: The resulting KDOM tree for the example input document with parsing comma-separated lists of abstract terms.

engineering, for example for building complex logical conditions by combining boolean operators. Applying a minor extension to the KDOM schema representation, these kind of languages can also be represented. Up to now a KDOM schema has been denoted to be a directed tree. By relaxing this claim by allowing directed graphs extends the expressiveness of KDOM schema to recursive schemas. There, existing types are registered as child type of one of its ancestors. In this way recursive grammar rules can be represented, allowing for the definition of languages of infinite depth. The algorithm proposed in Section 5.5.4 is capable to process recursive KDOM schemas without modifications. Nevertheless, additional expressiveness requires some more attention on schema modeling. While the parsing of tree schemas always terminates creating a valid KDOM tree structure, for recursive schemas termination of the algorithm is not guaranteed. When assuming that for each descending parsing step the segment text length decreases, termination is guaranteed obviously. Due to modeling errors, e.g., extensive use of the *ALL* finder, this assumption not necessarily holds and can lead to infinite loops. However, when applied correctly this modeling pattern allows for straight forward implementation of many kinds of helpful markup languages.

Example 1 Figure 5.12 shows a recursive KDOM schema for parsing complex formulas of propositional logics using negation, conjunction, disjunction and brackets for arbitrary structuring.

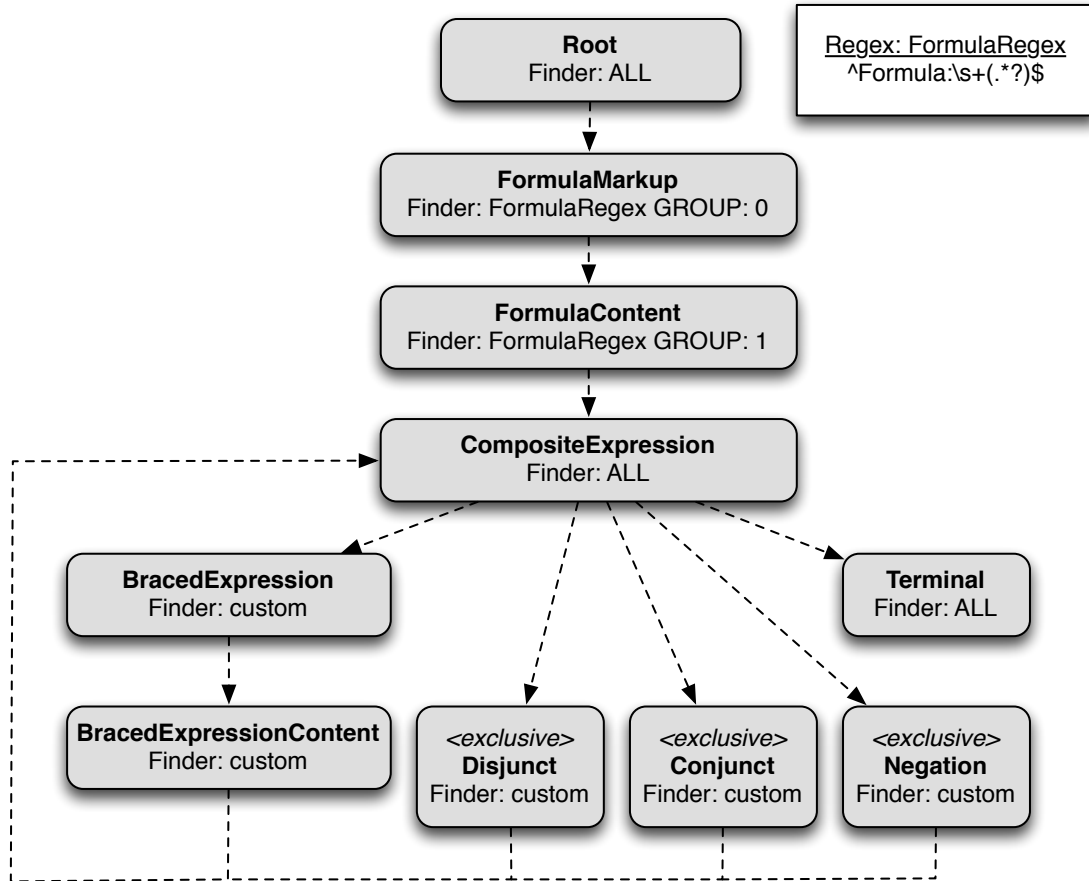


Figure 5.12: The KDOM schema to parse proposition logics expressions of arbitrary depth.

The types *BracedExpressionContent*, *Disjunct*, *Conjunct*, and *Negation* have back-links to *CompositeExpression*, that is *CompositeExpression* is a child type for each of them. The type *Terminal* is provided with the 'ALL' finder and terminates the recursion when none of the other types are detected. The types *Disjunct*, *Conjunct*, and *Negation* are provided with an *exclusive* flag, indicating that for remaining sibling fragments the parsing process is not continued. In this case the flag prevents the keywords ('AND', 'OR', 'NOT') to be detected by the *Terminal* type. If this *exclusive* flag is not supported by the parsing engine, the corresponding keywords have to be modeled explicitly as types.

Several parser components cannot be defined straight forward as a regular expression as also regular expressions have limited expressiveness. In this case it is necessary to count (brackets) which is not possible with regular expressions. However, due to the flexibility of the parsing

mechanism it is possible to define custom finder functions in a couple of lines of code. For the formula (*NOT (A) AND (B OR C)*) the resulting KDOM tree is shown in Figure 5.13. The type names are abbreviated and the top nodes of the types *Root*, *FormulaMarkup*, and *FormulaContent* have been omitted. One can see that a full depth parse tree has been generated. It can easily be used to create the corresponding formula in a knowledge base repository by a recursive tree traversing algorithm.

Example 2 Another example for a recursive KDOM schema is the parsing of so-called dash-tree structures. With dash-tree-based markups one can express hierarchical structures as already mentioned in Section 3.5.2.1. A dash-tree markup can be used for example to define a hierarchy of classes. The following example shows a small exemplary class hierarchy from the domain of history:

```
1 HistoryConcept
2 - Group of Persons
3 -- Dynasty
4 -- Social Stratum
5 -- Ethnic Group
6 - Geographic Object
7 -- City
8 -- Landscape
9 --- Island
10 --- Lake
```

The recursive KDOM schema to parse dash-trees of arbitrary depth is sketched in Figure 5.14. The finder functions in this case need to be implemented additionally. The *SubTree* type obtains an arbitrary dash-tree part and detects its current root(s). For each root a *SubTree* node, including all lines underneath the root, is created. Then in the next parsing step for each *SubTree* node the root line is consumed by the *DashTreeElement* type. The remainder, i.e., the sub-tree without the root line, is then again passed the *SubTree* type, which again analyses all sub-trees (of the next level). The result is a KDOM tree having the hierarchical structure explicitly represented in the tree structure.

5.3.7.3 Limits of KDOM Schema parsing

Most markups of low and medium complexity can be implemented as KDOM schemas. As shown in Section 5.3.7.1, simple markups can be implemented by very simple schemas. The extensions for cardinality constraints (c.f. Section 5.3.4) and backtracking (c.f. Section 5.3.5) simplifies the implementation of non-trivial markups as KDOM schemas. Nevertheless, with raising complexity of the language to be implemented the complexity of a KDOM schema implementation strongly increases. Not only the size (number of types) increases, but also the

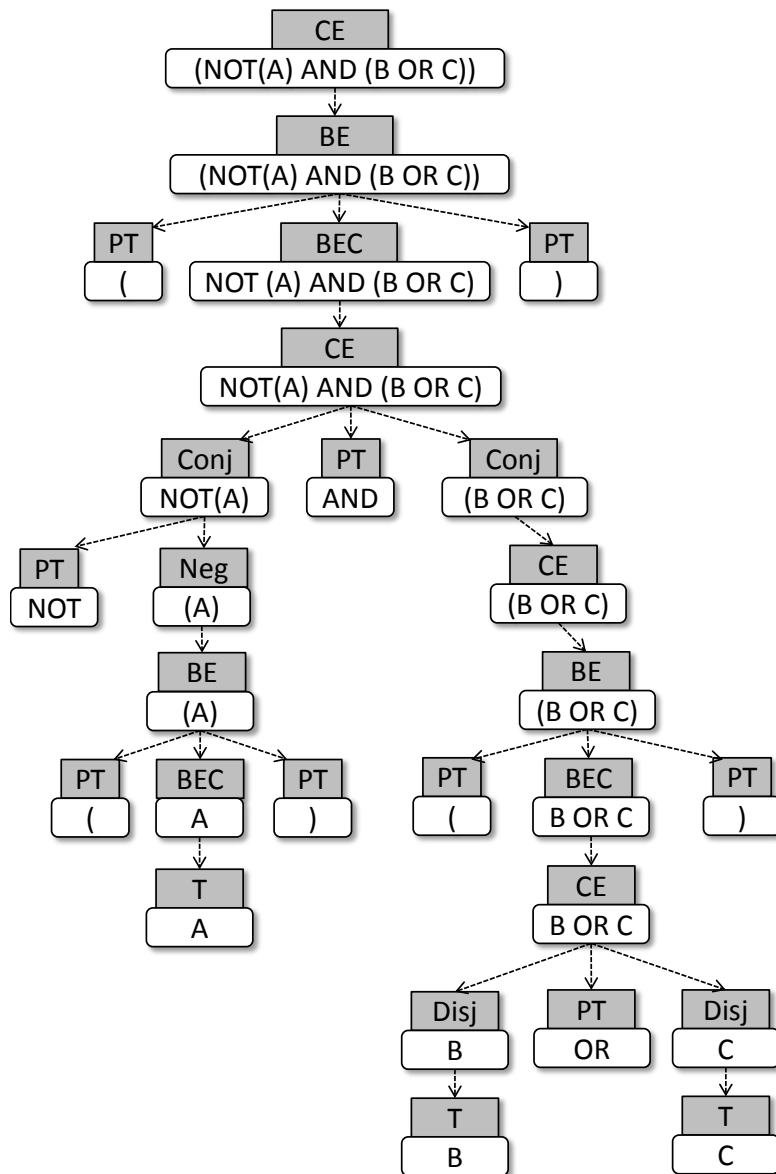


Figure 5.13: The resulting KDOM (sub-)tree for the input formula $(NOT(A) AND (B OR C))$.

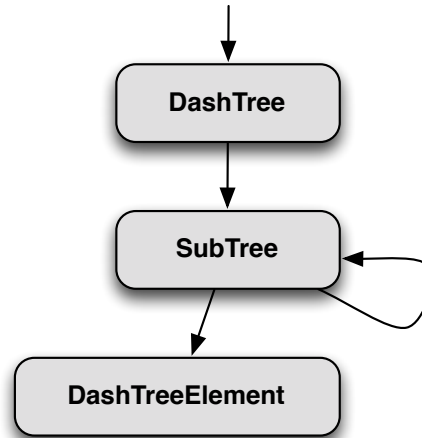


Figure 5.14: The recursive KDOM schema to parse dash-trees of arbitrary depth.

complexity of the customized finders that are required can become challenging. The resulting schemas for markups of higher complexity can become difficult to create and to maintain. Hence, for these kind of languages one should consider to go for a syntax-directed parsing approach [ALSU06]. Therefore, so-called parser-generator tools are available, as for example ANTLR [Par07] or yacc [Joh75]. For the use of these kind of tools the language is defined as grammar rules in a dialect of BNF [Knu64]. From this declarative language specification the source code of a parser is generated. While loosing some flexibility compared to KDOM schema parsing, as for example the ability to extend a (sub-)language at runtime by plugins, this method for parsing formal languages has been proven very reliable for decades. Even for languages of high complexity the declarative specification as a grammar is understandable and maintainable due to its clear semantics. The integration within the top-down parsing process, while costing some efforts, is possible as discussed in Section 5.3.2.4.

There is not only a practical limit for KDOM schema parsing as schemas and finder functions become complicated. There is also the theoretical limit, as some languages can never be recognized by KDOM parsers. The main reason for this is, that KDOM schema parsing works without tokenization of the input sequence as an initial step before the actual parsing process as known from grammar-based parsing. Figure 5.15 shows an example where KDOM schema parsing cannot be applied conveniently. While arithmetical expressions can be handled in general by a recursive KDOM schema, similar to the proposition logics example discussed in Section 5.3.7.2, the use of minus as a unary sign causes problems. On the left hand side of Figure 5.15 the minus is a binary operator and needs to be handled superior to the division in the parse tree due to the order of operation rule. For the parser functions dealing with subtraction it is not possible to decide in a straight forward way which of the cases is at hand. A thorough analysis of the context would be required. The grounding of the problem is, that minus is used in two different roles, a binary operator on the one hand and a unary sign on the other.

A workaround at the expense of the user would be to establish the convention that the unary use of minus as a sign requires to be put in brackets, i.e., $2/(-5)$. That case can be handled by

KDOM schema parsing easily.

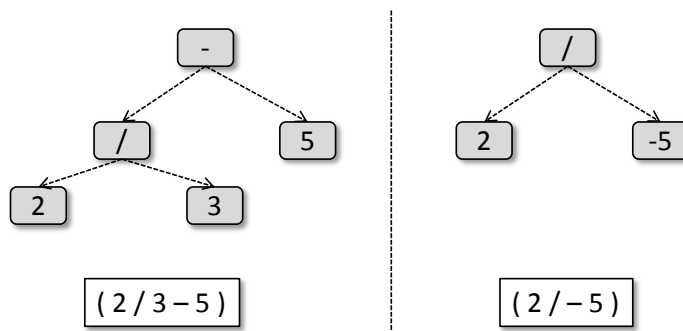


Figure 5.15: Example for an expression causing problems for KDOM schema parsing.

In sum, the implementation of parsers for languages of higher complexity is a non-trivial task in general. The choice of technology at some point also depends on the experiences of the developer in the respective technology. In case of doubt or if the language is likely to be extended to higher complexity later, the established syntax-directed parsing technology should be chosen. The implementation costs for complex languages is by magnitudes higher than for simple languages. Therefore, the efforts for the integration of the syntax-directed parser within the top-down segmentation mechanism can be estimated low in comparison.

The meta-engineering approach discussed in Chapter 3 proposes the frequent and timely introduction of new markups. These markups are designed for a special purpose showing limited expressiveness as discussed in 4.4.2.1. Further, they are created with simplicity as a primary design goal. Therefore, the markups designed within the meta-engineering process are typically characterized by a particularly low complexity. Hence, these markups most likely can easily be implemented as a KDOM schema.

5.4 Terminology Resolution and Knowledge Generation

The aspect of compilation considers the population of the knowledge repository after a structured representation of the input has been created by a parsing mechanism. This needs to include the detection and handling of errors. The different levels of errors that can occur within the process of knowledge markup compilation have been discussed in Section 3.1.6.2. In the previous section a parsing mechanism has been discussed, including some means for the detection of syntax level errors. In this Section, also the terminology error level is considered for error detection. Before introducing a compilation mechanism, we state more precisely the requirements for the compilation of multimodal knowledge in document-centered knowledge acquisition:

1. **Compatibility with Common Document Editing:** In Section 3.1 the characteristics of editing electronic documents have been discussed. The DCKA approach aims for a seamless extension of that wide-spread authoring paradigm to keep barriers low and therefore needs to comply to these characteristics. For the compilation mechanism especially

5 Techniques for the Implementation of DCKA

the *State by content* characteristic is relevant, saying that the overall behavior (including knowledge base behavior) only depends on the current state of the documents, not how this state has been reached. Additionally, it is important that the partition of the content elements into documents, as well as the order of elements within a document does not affect the resulting knowledge base.

2. **No Insertion of Invalid Knowledge:** Pieces of knowledge that are known to be invalid, e.g., due to misspelled term names, are not inserted into the knowledge repository.
3. **Instant Response:** The user expects instant and complete feedback after document modifications. A feedback about how the knowledge repository has been changed due to the modification is desired. Further, it is important that the knowledge base is instantly ready for testing [SS01]. Therefore, the compilation process should be performed in an imperceptible amount of time. Ideally, this should also hold for large document/knowledge bases.

In this section, we present an algorithm that complies to these requirements, by also detecting and resolving terminology level errors. The algorithm performs incremental knowledge base updates after each document modification. It uses the change delta with respect to the previous document version to make minimal changes to the previous version of the knowledge base. In this way, the complexity of the compilation of the knowledge base content for a document modification is uncoupled from the overall knowledge base-/document size. This allows to guarantee instant response also for large knowledge bases considering requirement 3, also providing a diff for the knowledge base change to be presented to the user. The second requirements basically says, that the current state of the document completely determines the content of the knowledge base. For the incremental algorithm this implies that in any case it needs to create the same compilation result as a standard compilation algorithm creating the knowledge base from the current document base from scratch. Based on this claim, the correctness of the incremental update algorithm can be rated. However, calculating the correct change operations on the knowledge base for arbitrary document changes at any state is challenging. To verify the correctness of the proposed algorithm, we decided to provide a formal proof of correctness. Therefore, in the next sections we introduce a formal description of the knowledge compilation problem that is suitable to support the conduction of a formal proof. It extends the descriptions of knowledge representation and markup languages given in Section 3.1.2.

Requirement 2 demands that invalid knowledge is not inserted into the knowledge repository. Therefore, the terminology level errors need to be analyzed. In the following we discuss reference resolution in the context of *closed-world authoring*.

5.4.1 Reference Resolution in Closed-World Authoring

For most third generation programming languages variable names are checked for being correctly defined and an error is shown if the check fails. In some untyped languages (e.g., javascript) these checks are not performed. Then the programmer runs danger to misspell a variable which leads to a lazy initialization of a new one carrying a default value. That will cause unintended

behavior at runtime with the reasons often no easy to detect. Also in document-centered knowledge acquisition these kind of problems, called terminology level errors, can occur and can be ruled out in a similar way. There, objects are identified by a unique term name within the markup. We say, an authoring environment provides *open world authoring* if arbitrary object identifiers can be used ad-hoc, i.e. the world of object terms is unrestricted. However, with respect to user assistance it is advantageous to add terminology checks to assert that the objects have been referenced correctly, i.e., were not mistyped. An authoring environment provides *closed-world authoring* if the set of referenced objects is restricted to an (extensible) set of explicitly defined objects. In these environments, a markup expression can be invalid if referring to non-existing objects. In this case a compilation error is returned to the user. This requires a set of declared objects, which should be extensible easily. Therefore, it is reasonable to define the object declarations in a similar way as the rest of the knowledge in the document base using markup language. This is similar to compilation of programming code, where in many third-generation-languages the compiler requires a variable to be correctly declared before usage.

5.4.2 An Abstract Model for Knowledge Bases

The diversity of possible symbol level knowledge representations briefly has been discussed in Chapter 1, together with the statement that this work is independent of the representation at hand. However, for the discussion and proof of an algorithm for incremental compilation a precise description of the nature of the parts of a knowledge base is necessary. In the following, we define a formal model which on a very general level, summarizing the characteristics of most kinds of symbol level knowledge representations, as far as relevant for the compilation process. The compilation mechanism can be applied to any symbol level knowledge representation that can be mapped unambiguously to this formal model. One fundamental claim at this point is, that a knowledge base consists of an (unordered) set of words, as already briefly discussed in Section 3.1.2. These "small pieces" of knowledge, which are further called *knowledge slices*, associate domain objects and data type values to form some kind of propositions.

We formally define \mathcal{O} as the universal set of all possible domain objects and \mathcal{D} as the universe of all data type values. \mathcal{R} is considered as the universe of all association types. A knowledge representation \mathcal{K} is defined by a set of allowed association types $R_{\mathcal{K}} \subset \mathcal{R}$. All tuples $(r, o_1, \dots, o_n, d_1, \dots, d_k)$ with $n \geq 1$ and $k \geq 0$ are forming the possible knowledge slices of \mathcal{K} . A knowledge slice captures an association of type $r \in R_{\mathcal{K}}$ between the objects o_1, \dots, o_n and the values d_1, \dots, d_k . A *knowledge base* $A \subset \mathcal{K}$ is then given by a set of such tuples/knowledge slices.

This rather general knowledge model of representing a knowledge base includes a wide range of declarative knowledge representations, e.g., logic-based formalisms or production rules. However, structures containing implicit orderings, such as lists, are difficult to represent as multiple knowledge slices in a straight forward way. In particular, the abstract knowledge base model is unsuited to cover sequences of commands as it is known from various imperative programming languages. To map knowledge representations with smaller order sensitive structures to this model, a workaround is possible: The entire order-sensitive list/sequence can be modeled one large knowledge slice. In that way, all the following methods can be applied, possibly at the expense of performance, depending of the size of these structures.

5 Techniques for the Implementation of DCKA

Additionally, we define the reference function for knowledge slices

$$Ref : \mathcal{K} \mapsto 2^{\mathcal{O}}$$

that maps the knowledge slices to the set objects, that are associated with this slice. It can be considered as a projection on a tuple returning the sub-tuple (o_1, \dots, o_n) . This auxiliary function will be required for terminology reference resolution by the knowledge base update algorithm.

Examples: We provide some examples how existing knowledge representations can be mapped to this formal model. A simple RDF-based knowledge representation contains (at least) two association types to form triples: $\mathcal{K}_{RDF} = \{TR_{3,0}, TR_{2,1}\}$

$TR_{3,0}$ allows to associate three URIs to form one triple and $TR_{2,1}$ two URIs and one data-type value. The following RDF triples can then be described as knowledge slice tuples as discussed above (namespaces omitted for brevity):

Jochen livesIn:: Wuerzburg $\Leftrightarrow (TR_{3,0}, \text{Jochen}, \text{livesIn}, \text{Wuerzburg})$

Wuerzburg hasZipCode:: 97070 $\Leftrightarrow (TR_{2,1}, \text{Wuerzburg}, \text{hasZipCode}, 97070)$

Jochen, livesIn, Wuerzburg, hasZipCode $\in \mathcal{O}$; *97070* $\in \mathcal{D}$

As another example, a simple condition-action rule knowledge representation \mathcal{K}_{Rule} is mapped. We consider the following rule:

IF A AND B THEN C

This way to associate terms, which might be called $AND_{2,1}$, should be contained in \mathcal{K}_{Rule} : $AND_{2,1} \in \mathcal{K}_{Rule}$. Then the rule above can be written as tuple as follows:

$(AND_{2,1}, A, B, C), A, B, C \in \mathcal{O}$

It is necessary, that the mapping is unambiguously. This might require to extend the relation set \mathcal{R} accordingly, possible to an infinite set.

5.4.3 A Formal Model for Knowledge Authoring

Let E be the set of all possible text segments on a document base DB . $M_{\mathcal{K}_R}$ is a markup language, that is suited to capture knowledge slices of \mathcal{K} unambiguously as text expressions. We then define the segmentation parser function

$$P_{\mathcal{K}_R} : DB_M \mapsto 2^E$$

that is able to create a set of minimal text segments (identified by the position in the source) from DB according to $M_{\mathcal{K}}$, where each segment is a syntactically independent expression that represents one or multiple knowledge slices. As the range of this function is a set, the knowledge slices can freely be organized in the source text according to modeling conventions established for the current project by a knowledge engineer (c.f., KAA 4.3.2).

Further, we define a compilation function which implements λ , i.e., creates a set of knowledge slices of \mathcal{K} for a text expression of $M_{\mathcal{K}}$ to :

$$C_{\mathcal{K}}^{M_{\mathcal{K}}} : E \mapsto 2^{\mathcal{K}}$$

While in many cases only one knowledge slice is compiled from one text segment, depending on the markup structure and the target knowledge representation, it is possible and sometimes convenient to create multiple knowledge slices from one expression. Then however, all these knowledge slices are excluded from the knowledge base in the case that the expression contains one or multiple errors.

We can then define a document-centered knowledge authoring-system for a knowledge representation \mathcal{K} as a triple:

$$T_{\mathcal{K}} = \{M_{\mathcal{K}}, P_{\mathcal{K}}, C_{\mathcal{K}}^{M_{\mathcal{K}}}\}$$

5.4.4 Closed-World Authoring Reconsidered

Being introduced In Section 5.4.1, the concept of closed-world authoring can be defined more precisely by applying the presented formal model of knowledge authoring. An authoring environment provides *closed-world authoring* if the set of referenced objects is restricted to an (extensible) set of explicitly defined objects $O \subset \mathcal{O}$. Analogously, we say an authoring environment provides *open-world authoring* if arbitrary object identifiers can be used ad-hoc, that is \mathcal{O} is unrestricted. In this case, all text expressions for knowledge slices, that are syntactically correct, are considered valid and are compiled into the knowledge base. To provide optimal user assistance in the following the problem of terminology resolution for closed-world authoring environments are analyzed. There, a text expression e can be invalid if referring to non-existing objects ($Ref(C(e)) \not\subseteq O$). For the authoring environment this implies that the markup language and the parsing function are extended to support object definitions. Additionally, we introduce the object compilation function C_o that creates objects from object definition expressions:

$$C_o : E \mapsto \mathcal{O}$$

Based on these definitions, the current set of defined objects in a document base is given by:

$$O = \{o \in \mathcal{O} : o \in C_o(e), e \in E\}$$

Since in closed-world authoring not only knowledge slices are compiled, but also object definitions, the compilation process is two-layered. The explicit definition of domain objects in the source text introduces two additional issues, especially considering, that object definitions might be arbitrary located in the document base.

5.4.4.1 Strict Object Definition

An object could possibly be defined at multiple locations in the document base, which may have unintuitive implications for the behavior of the authoring environment towards the user, e.g., the deletion of an object definition will not remove the object from the set of defined objects. Even worse, by overwriting one object definition the user will generate an additional object without being aware. Therefore, we postulate strict object definitions, that is, we consider *all* definitions of an object as invalid if more than one exist.

5.4.4.2 Complex Object Definitions

Sometimes an object heavily relies on other objects only making sense if these other objects are (validly) defined (see example below). We denote the set of objects, which is required for a meaningful definition of o as $Ref_o(o)$. We call object definitions relying on other objects *complex definitions*. In that case, an important service for the user is to give feedback whether one of the dependent objects is missing and to propagate the invalidity chain. This leads to an extension of the validity concept for object definitions: An object is valid, iff it has a unique definition e and all objects referenced by e are also valid.

In the following, we present a small markup example demonstrating the definition of simple and complex objects and knowledge slices from an excerpt of an exemplary knowledge base calculating the body-mass-index:

```
1 def weight
2 def height
3
4 def BMI = weight / (height * height)
5
6 def underweight
7 IF BMI < 18.5 THEN underweight
```

This document excerpt defines four objects (lines 1,2,4,6) and two knowledge slices (lines 4,7). Please note, that in line 4 both, a complex object definition (defining BMI with dependencies on $weight$ and $height$) and a knowledge slice (associating the objects BMI , $weight$ and $height$) are implied. We consider the typing error in the right-hand-side of the expression in line 4: Of course, an error will be displayed in line 4. But as BMI is not added to O because one of the referenced terms is not in O , the knowledge slice in line 7 will also not be compiled to the knowledge base but shows a message telling the user, that BMI is not correctly defined. In general, the expressions should be structured in a comprehensible natural order. However, according to the above definitions, the ordering/location of the expressions is completely irrelevant for the compilation result.

5.4.5 The Knowledge Compilation Task Summarized

The goal of the compilation task is to generate a valid version of the knowledge base (compilation) with respect to the current document base. We call a knowledge base a valid compilation with respect to a set of source text documents if it contains exactly all valid knowledge slices defined by the document base. However, the validity of knowledge slices and complex object definitions is defined with respect to O , which itself is compiled from the document base. This two stage compilation process is a non-trivial task: Even if only one document is modified slightly the modification can affect entities, which are defined in other documents. For example, the deletion of an object definition can cause other (complex) object definitions or knowledge slices to become invalid. The addition of an object definition can cause the object to become

invalid (if it is a duplicate) or can cause knowledge slices to become valid (because the definition was missing before). Any expression in another document might be affected in the worst case. If an algorithm always creates valid compilations from source documents we call it *sound*. A sound compilation algorithm processing the whole document base and building up O and A can be defined in a straight forward way. However, document-centered authoring systems are designed for agile development allowing for frequent changes and immediate feedback and testing capabilities. It is obvious that the runtime of such an algorithm scales linear with the absolute size of the knowledge and document base, leading to insufficient performance on large knowledge bases. In the next section, we show how to create sound compilations of modified document bases (substantially) more efficiently using the incremental approach.

5.4.6 The Resource Delta

Incremental compilation describes the translation of text sources based on the small changes of the document base instead of a complete rebuild. In order to perform an incremental compilation of a document-set using closed-world authoring, a parser component and an incremental knowledge base update algorithm resolving terminology dependencies are required. The task of parsing multimodal knowledge documents has been discussed in Section 5.3. While the parsing mechanism introduced in Section 5.3.3 provides a convenient solution, in principle it is not relevant which parsing technology is employed. The only requirement is, that the information about the changes with respect to the previous version, the so-called resource delta, is available. Also for syntax-directed parsing incremental algorithms are available, e.g., as proposed by Wagner et al.[WG98]. However, also a non-incremental parsing mechanism can be employed if the resource delta can efficiently be calculated, for example using tree comparison techniques. Using a non-incremental parser not even poses a drawback with respect to scalability as only the one document, that has just been modified, is parsed before each compilation step. We assume that documents have a bounded size, i.e., the number of documents scales with the project size but not the sizes of the documents themselves. Even for reasons of comprehensibility documents of exceeding sizes should be partitioned.

For a document base DB_{old} that is modified to a document base DB the resource delta is formally defined as:

$$\Delta(DB_{old}, DB) = \{N, D\},$$

where N is the set of new expressions defined by $N = P(DB) \setminus P(DB_{old})$ and D is the set of deleted expressions $D = P(DB_{old}) \setminus P(DB)$. By distinguishing D and N with respect to object definitions (D_o, N_o) and knowledge slices (D_a, N_a) , we also denote $\Delta(DB_{old}, DB) = \{N_o, N_a, D_o, D_a\}$.

5.4.7 The Genericity of the Incremental Knowledge Base Update Task

The value of the algorithm introduced in the following comes from its general applicability: It is independent of the markup language *and* the target representation. That means, once implemented into a document-centered authoring system, it can be used to compile any markup to any kind of knowledge repository.

This independence is possible for two reasons:

5 Techniques for the Implementation of DCKA

1. Due to the set characteristic of a knowledge base (cf. Section 5.4.2) a knowledge slice can be translated to the target representation independently from others, i.e., without regarding the context.
2. Referenced domain objects in closed-world authoring form (possibly complex) dependencies to arbitrary other parts of the code. However, this single dependency class is inherent to all knowledge representations and markups and therefore can be treated in a general way.

The independence of the markup language in fact is achieved by the the definition of the general reference functions the reference functions (Ref , Ref_o). Those obtain the object references from the syntax tree, without knowing all details of the markup language. The parsing function $P_{\mathcal{K}R}$ depends on the markup language as it obviously needs to know the syntax for parsing. The knowledge slice compile function $C_{\mathcal{K}}^{M_{\mathcal{K}}}$ depends on both, the knowledge representation and the markup language. However, the compile function $C_{\mathcal{K}}^{M_{\mathcal{K}}}$ can be used like an abstract interface function by the incremental update algorithm, independently of what or how knowledge slices are created. This is illustrated in Figure 5.16. The left hand side deals with parsing and delivering the resource delta based on parsed markup expressions to the incremental compilation algorithm located in the center. The algorithm the computes for which expressions knowledge slices need to be inserted into the knowledge repository and calls the compile function $C_{\mathcal{K}}^{M_{\mathcal{K}}}$ accordingly. Further, it computes for which expressions the knowledge slices need to be removed from the knowledge repository. On the right hand side, the actual insert and remove operations are performed, according to the knowledge repository employed.

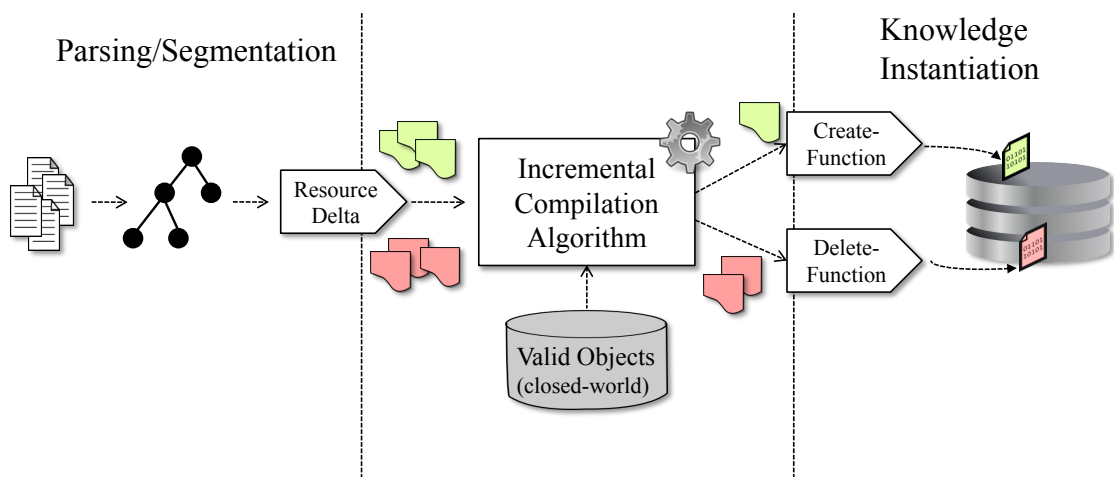


Figure 5.16: The workflow of the generic incremental compilation algorithm.

5.4.8 An Incremental Knowledge Base Update Algorithm

In the following listing, we present an algorithm, that updates a valid compilation of the old document base to a valid compilation of the new version, given the corresponding resource delta. A basic version of this algorithm is also published in [RSLP11]. Beside the resource delta $\Delta = \{N_o, N_a, D_o, D_a\}$, the input of the algorithm is a set of objects O_{old} and knowledge slices A_{old} of the prior compilation and the *Reference Manager* (RM). The *Reference Manager* is an auxiliary data structure, that provides the information which objects are referenced by other objects or knowledge slices in the source text (i.e., the dependency graph). The output of the algorithm is a set of objects O and knowledge slices A , forming a valid compilation of the new document base. Additionally, whenever an invalid object definition or knowledge slice is detected in the algorithm, the error messages for the respective text segments can be generated.

5.4.8.1 Description

The algorithm presented in Listing 5.5 aims to identify a minimal subset of entities that needs to be removed or created in order to form a valid compilation of the new document base with respect to O and A . The general strategy of this update function is to resolve the object dependencies for updating O . While doing so it identifies not directly modified knowledge slices that have to be reconsidered. Afterwards, the actual knowledge base is updated. First the reference manager gets updated with the new resource delta in line 2. Then in line 4, we iterate on the deleted object definition expressions D_o and either call recursive removal or resolution on this object definition, depending on the validity of the respective object. Further, this is applied in a similar fashion to the set the new object expressions N_o in line 7. Please note, that both, adding or deleting an object definition can result in some object becoming valid. The functions *resolveRecursively* and *removeRecursively* traverse the object dependency graph from the passed object on, and create and remove respectively the (complex) object definitions and mark the affected knowledge slices (by inserting elements into D_a and N_a). The hazard-filter (line 14) removes expression pairs, that would lead to deletion and subsequently insertion of an identical knowledge slice: $C(e_a) = C(e'_a), e_a \in N_a, e'_a \in D_a$. Finally, we remove the deleted knowledge slices in D_a and insert the new knowledge slices N_a if all referenced objects are valid at that time in line 17 and 20 respectively.

5.4.8.2 Termination

Cyclic object definition dependencies using complex definitions may occur within the text. However, all object definitions involved in a dependency cycle are invalid: Before a cycle can be closed, it is invalid because of the missing part. When the missing part is added it will not become valid, as its dependency was invalid in the prior version (that is line 48 will evaluate to false). Therefore, the function *resolveRecursively* terminates as validity is checked in line 31 and no valid cycles can exist (and the dependency graph is finite). The function *removeRecursively* terminates for similar reason: In line 44 we assert that recursion only proceeds if the object is in O_{old} . As O_{old} only contained valid objects and no valid cycles can exist, the function terminates.

5 Techniques for the Implementation of DCKA

```

1 function updateKB( $\Delta, O_{old}, A_{old}, RM,$ )
2   updateReferenceManager( $\Delta, RM$ )
3    $O := O_{old}; A := A_{old}$ 
4   for each  $e_o \in D_o$  do
5     checkObject( $e_o$ )
6
7   for each  $e_o \in N_o$  do
8     checkObject( $e_o$ )
9
10  for each  $e_a \in N_a$  do
11    if (not  $Ref(C(e_a)) \subseteq O$ ) //check for validity
12      remove  $e_a$  from  $N_a$ 
13
14  hazardFilter( $D_a, N_a$ )
15
16  for each  $e_a \in D_a$  do:
17    delete  $C(e_a)$  from  $A$  //remove from KB
18
19  for each  $e_a \in N_a$  do
20    add  $C(e_a)$  to  $A$  //insert into KB
21
22  // auxilliary functions
23  function checkObject( $e_o$ )
24    if (hasValidDefinition( $C_o(e_o)$ ))
25      resolveRecursively( $e_o$ )
26    else
27      removeRecursively( $e_o$ )
28
29  function resolveRecursively( $e_o$ )
30    if (hasValidDefinition( $C_o(e_o)$ ))
31      if ( $C_o(e_o) \notin O_{old}$ )
32        add  $C_o(e_o)$  to  $O$ 
33        for each  $e_a \in RM.getReferencingSlices(e_o)$ 
34          add  $e_a$  to  $N_a$ 
35        for each  $e'_o \in RM.getReferencingDefs(e_o)$ 
36          resolveRecursively( $e'_o$ )
37
38  function removeRecursively( $e_o$ )
39    if ( $C_o(e_o) \in O_{old}$ )
40      remove  $C_o(e_o)$  from  $O$ 
41      for each  $e'_o \in RM.getReferencingDefs(e_o)$ 
42        removeRecursively( $e'_o$ )
43      for each  $e_a \in RM.getReferencingSlices(e_o)$ 
44        add  $e_a$  to  $D_a$ 
45
46  function hasValidDefinition( $o$ )
47    defs =  $RM.getDefinitions(o)$ 
48    return  $\#defs == 1 \ \& \ Ref_o(o) \subseteq O \setminus o$ 
49
50  function updateReferenceManager( $\Delta, RM$ )
51    for each  $e \in N_o \cup N_a$ 
52       $RM.registerReferences(e)$ 
53    for each  $e \in D_o \cup D_a$ 
54       $RM.deregisterReferences(e)$ 

```

Listing 5.5: The incremental knowledge base update algorithm.

5.4.8.3 Efficiency

The potentially most expensive operations are the remove- and insert operations of knowledge slices, depending on the employed knowledge repository and its performance characteristics. The algorithm inserts or removes any knowledge slice only once at most. Given the correctness of the algorithm (i.e., exactly all valid knowledge slices are contained in the knowledge base), it follows that the algorithm is optimal with respect to knowledge slice insertion/removal. As the graph structure of the dependencies can be stored in the reference manager RM using hash tables, the lookup operations on RM during dependency resolution can be performed in constant time. The update of that information (*updateReferences*) only takes linear time to the size of the resource delta. The runtime in practice is strongly determined by the amount of change operations on the knowledge base (cf. evaluation in Section 7). If dependency resolution can be considered as small compared to change operations, the complexity of the algorithm is linear to the knowledge base change set. This is a significant improvement when compared to the naive full-compile approach which is linear to the size of the document base and knowledge base.

5.4.8.4 Proof of correctness

We prove the correctness of the algorithm by showing that A and O , as the result of the update algorithm applied to A_{old} and O_{old} and Δ , contains exactly the same entities as a valid compilation O^* and A^* . O_- and A_- are the sets of objects respectively knowledge slices from the old compilation that are still valid and therefore are retained in the compilation: $O_- = O \cap O_{old}$; $A_- = A \cap A_{old}$. O_+ and A_+ are the sets of objects respectively knowledge slices added by the update algorithm: $O_+ = O \setminus O_-$; $A_+ = A \setminus A_-$.

We introduce the following auxiliary theorems:

$$\begin{array}{l|l} \mathbf{O}_- \subseteq \mathbf{O}^* & \mathbf{A}_- \subseteq \mathbf{A}^* \\ \mathbf{O}_+ \subseteq \mathbf{O}^* & \mathbf{A}_+ \subseteq \mathbf{A}^* \\ \mathbf{O}^* \subseteq \mathbf{O} & \mathbf{A}^* \subseteq \mathbf{A} \end{array}$$

Using these auxiliary theorems, it is easy to infer:

$$\begin{array}{l|l} \mathbf{O} = \mathbf{O}_+ \cup \mathbf{O}_- \subseteq \mathbf{O}^* & \mathbf{A} = \mathbf{A}_+ \cup \mathbf{A}_- \subseteq \mathbf{A}^* \\ \mathbf{O} \subseteq \mathbf{O}^* \text{ and } \mathbf{O} \supseteq \mathbf{O}^* & \mathbf{A} \subseteq \mathbf{A}^* \text{ and } \mathbf{A} \supseteq \mathbf{A}^* \\ \implies \mathbf{O} = \mathbf{O}^* & \implies \mathbf{A} = \mathbf{A}^* \end{array}$$

We proved the soundness of the algorithm, given that the auxiliary theorems hold. They are proven as follows:

$\mathbf{O}_- \subseteq \mathbf{O}^*$: assume $\exists o, o \in O_- \wedge o \notin O^*$: o must have had a valid definition $e'_o \in P(DB_{old})$, as by definition of O_- : $O_- \subseteq O_{old}$. As it is not in O^* , there is no valid definition in the current version. Some text modification must have been made yielding in the definition of o to become invalid, and only two cases for this to happen are possible: (1) Its text expression has been modified directly (towards another object definition or completely removed). (2) The definition itself

5 Techniques for the Implementation of DCKA

was not modified but became invalid due to a competing definition added. (3) The definition itself was not modified but became invalid because a dependency object became invalid by the modification..

(1) By the definition of Δ , e_o is in D_o . Line 24 checks whether there currently exists a valid definition of $C_o(e_o)$. As $o \notin O^*$, this is not the case. Because of $e'_o \in P(DB_{old})$ line 39 will evaluate to true and the object will be removed (recursively) and thus o cannot be in $O_- \implies \downarrow$

(2) A concurring definition of the object has been added to the document base. If any object definition e'_o is added to the document: $e'_o \in N_o$. Line 24 checks whether there currently exists a valid definition of $C_o(e_o)$, which is not the case due to the competing definition. Because of $e'_o \in P(DB_{old})$, line 39 will evaluate to true and the object will be removed (recursively) and thus o cannot be in $O_- \implies \downarrow$

(3) At least one of the objects, the (complex) definition of o is based on, has become invalid. Thus e''_o exists, being responsible for that (directly or indirectly) and $e''_o \in \Delta$. As o is in O_{old} , $o'' = C_o(e''_o)$ must have been valid in $P(DB_{old})$. Therefore, line 39 will evaluate to true and e''_o will be deleted recursively. This recursive deletion will also delete o , as it references o'' , potentially indirectly. Thus, o cannot be in $O_- \implies \downarrow$

As $o \in O_- \wedge o \notin O^*$ leads to a contradiction in any case, $O_- \subseteq O^*$ holds.

$A_- \subseteq A^*$: assume $\exists a, a \in A_- \wedge a \notin A^*$: a must have had a valid definition $e_a \in P(DB_{old})$, as $A_- \subseteq A_{old}$. As $a \notin A^*$, there is no valid expression $e_a \in P(DB)$. Some text modification must have been made so that a became invalid and was not compiled to A^* . There are two possibilities for this:

(1) The expression was changed from e_a to e'_a with $a = C(e_a) \neq C(e'_a) = a'$. By the definition of Δ , the expression would be in D_a . The deletion step in line 17 removes all knowledge slices a from A where $e_a \in D_a$ (see line 16) and thus a cannot be in $A_- \implies \downarrow$

(2) The modification caused a definition of some object o to become invalid, which leads to a becoming invalid. Thus e'_o exists, being responsible for that (directly or indirectly), and $e'_o \in \Delta$. As o is invalid, *removeRecursively* will be called. Because a was in A_{old} , $C_o(e'_o)$ must have been valid in $P(DB_{old})$ and line 39 will evaluate to true and e'_o will be deleted recursively. This recursive deletion also adds a to N_a , as it references o' (indirectly). Then a will be deleted in line 17 and thus cannot be in $A_- \implies \downarrow$

As $a \in A_- \wedge a \notin A^*$ is contradictory in any case, $A_- \subseteq A^*$ holds.

$O_+ \subseteq O^*$: Objects in O_+ are only added in line 32. For those a valid definition is asserted in line 30. Hence, they are also contained in O^* : $\implies O_+ \subseteq O^*$

$A_+ \subseteq A^*$: Knowledge slices in A_+ are only added in line 20. For those a valid definition is asserted in line 11. Hence, they are also contained in A^* . $\implies A_+ \subseteq A^*$

$O^* \subseteq O$: For each $o \in O^*$: It either had a valid definition in $P(DB_{old})$ (1), or it has become

valid by the modification of DB_{old} (2).

(1) $o \in O_{old}$ as it was a valid compilation. As $o \in O^*$, o has a valid definition $e_o \in P(DB)$. o will not be deleted directly by line 27, as it has a valid definition (condition in line 24). For any object being called in line 38 an invalid referenced object exists (entails by induction). If so for o , o would not be a valid definition. Therefore, o is not removed and thus is (still) in O .

(2) $o \notin O_{old}$ as o did not have a valid definition in $P(DB_{old})$, and as $o \in O^*$, o has a valid definition $e_o \in P(DB)$. Therefore, some modification $e'_o \in \Delta$ exists, that caused the definition of o becoming valid. Only objects becoming valid can cause other objects to become valid (i.e., an object becoming invalid can never make other objects become valid). Thus, *resolveRecursively* will be called on this e'_o and line 30 will return true (even if $e'_o \in D_o$). $C_o(e'_o)$ will be added in line 32 and in line 36 all referencing objects are recursively checked for validity. As e'_o caused o to become valid, some reference chain to o exists and thus o will be checked and created as all objects in the reference chain have been added to O . $\implies O^* \subseteq O$

$A^* \subseteq A$: For each $a \in A^*$: It either had a valid definition in $P(DB_{old})$ (1), or it has become valid by the modification (2).

(1) $a \in A_{old}$ as it was a valid compilation. As $a \in A^*$, a has a valid definition e_a in $P(DB)$. If $e_a \notin \Delta$ and all objects $o_a \in Ref(C(e_a))$ are validly defined, e_a cannot be added to D_a in line 44, as *removeRecursively* is only called on invalid objects (entails by induction). Therefore, it cannot be deleted in line 17 and thus is contained in A . If $e_a \in \Delta$, the only reason for this can be the removal of the definition in one location and its addition in another (move expression operation). Then $e_a \in N_a$ and $e_a \in D_a$. In this case e_a will be removed by the hazard-filter (and if not, it will be removed and inserted again) and is thus contained in A .

(2) $a \notin A_{old}$ as a did not have a valid definition in $P(DB_{old})$. As $a \in A^*$, a has a valid definition e_a in $P(DB)$. Therefore, some modification $e' \in \Delta$ exists, that caused the definition of a becoming valid. If the knowledge slice expression was modified directly, $e' \in N_a$ and a will be added to A in line 20. If the modification of some object definition e_o caused a to become valid, then e_o is in Δ . Thus, *resolveRecursively* will be called on this e'_o and line 30 will evaluate to true (even if $e'_o \in D_o$). $C_o(e'_o)$ will be added to O in line 32 and in line 36 all referencing objects are recursively checked for validity. As e'_o caused a to become valid, some valid reference chain to a exists and thus e_a will be added to N_a . During the recursive processing of this chain all objects in the chain are added to O . Therefore, a will be inserted into A in line 20, as $Ref(e_a) \subseteq O$ (checked in line 11) holds. $\implies A^* \subseteq A$ \square

5.4.9 Discussion

Incremental source compilation in the context of software engineering is a quite hard task. It needs to be solved specifically for each programming language anew. While looking similar at first glance, the compilation of multimodal knowledge documents shows rather different characteristics. There, one generic algorithm can be employed for terminology resolution for any markup languages to any knowledge repository (that can be mapped to the abstract knowledge model described at Section 5.4.2). This is possible because the problem is less complex than the compilation of general purpose programming languages, considering the interdependencies

of the source parts and the target structure. In contrast to software engineering, the formation (of the valid part) of the knowledge base should be completed even if compile errors are detected. The detection and handling of terminology level errors for closed-world authoring is the main task of the presented algorithm, forming the middle-part of the overall compilation process as illustrated in Figure 5.16. For open-world authoring, this part collapses and a corresponding algorithm would become trivial, only iterating over the new and deleted knowledge slice expressions of the resource delta and calling the create or delete function accordingly.

A considerable risk of incremental compilation in general is its vulnerability to (implementation) errors. If there are cases which are not processed entirely correct, serious problems occur and persist in the authoring environment. If for example the closed-world object set is not correctly updated one time, the system continuously shows to the user a completely incomprehensible behavior with respect to error messages and knowledge base content. It makes further authoring quite impossible. Therefore, the soundness of the implementation of such kind of algorithm is a critical point. Nevertheless, once a sound implementation is achieved it will serve well for any kind of document-centered knowledge acquisition project due to its genericity. Then, only parsers and compile scripts actually instantiating the knowledge slices need to be created for new configurations. Further, the incremental approach performs excellent with respect to response time of the system as the complexity of a knowledge base update is decoupled from the overall size of the knowledge base. Additionally, the algorithm provides a knowledge base diff with respect to the prior version for free. It can be presented to the author as feedback or used for building a "semantic time machine" that allows for long term analysis of the knowledge base development process.

5.5 A Meta-Model for the Declarative Implementation of Markups

For the meta-engineering approach the introduction and implementation of new custom markups is a key task. The implementation at limited development time is critical for the effectiveness of the overall approach. By now two important aspects of the technical task of knowledge document processing, parsing and knowledge base population, have been discussed. In this section, we discuss how new markups can be implemented quickly and easily. Therefore, we make use of the meta-modeling principle by creating a meta-model, which allows for the declarative modeling of a markup language. We also discuss the architecture of a system, which is able to interpret these kind of models, i.e. compile the respective markup within a document-centered authoring environment. Therefore, we employ the techniques introduced in Section 5.3 and Section 5.4. At first however, the problem is summarized once more in its entirety.

5.5.1 The Levels of Multimodal Knowledge Compilation Revisited

In Section 3.1.6.2 we discussed the four levels of knowledge communication. The compilation of multimodal knowledge documents, being a form of communication, also passes through these levels. Figure 5.17 shows the compilations process running through those levels. Each layer represents one level with its name indicated on the right. In the second column the result created by the activities of this layer is stated, together with the kinds of errors that are detected in

this level. The main column on the left describes what kinds of information are created by the activities on this layer and are passed to the next layer.

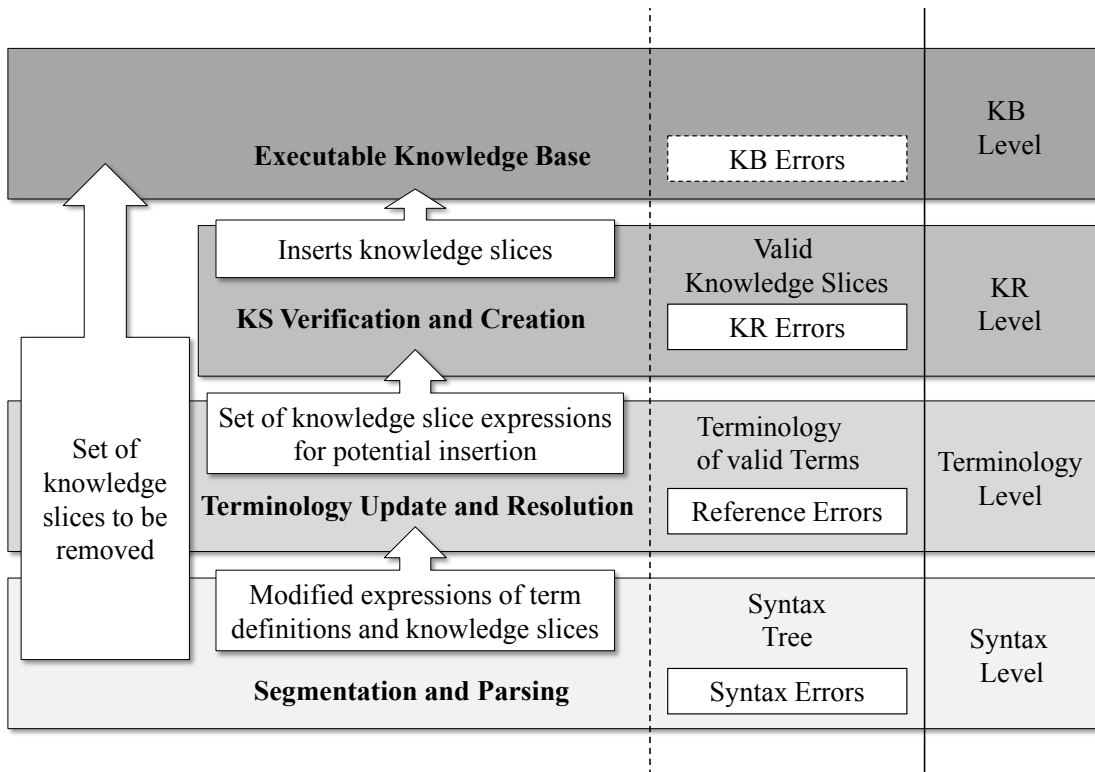


Figure 5.17: A technical view on the different levels of knowledge communication within compilation of multimodal knowledge documents.

From the technical perspective, different software components are relevant at the different stages. In the following those components are briefly introduced:

- **Markup (M):** The markup language including the syntax parser and the instructions to convert the knowledge to the repository.
- **Knowledge Representation Repository (KR):** An implementation of the employed knowledge representation, including a knowledge base repository and the reasoning mechanism. The reasoning mechanism allows for the detection of knowledge representation errors, such as inconsistencies.
- **Knowledge Base (KB):** The knowledge base, i.e. as the content of the knowledge base repository.
- **Terminology (T):** The entire terminology of the domain objects used for forming the knowledge base. This refers to the closed-world set of objects discussed in Section 5.4.

- **Document Base (DB):** The entire content of the document base.
- **Syntax Tree (ST):** The structured representation resulting from the parsing of the document base.
- **Knowledge Base Behavior Tests (KBT):** Tests that test the competency of the knowledge base.

The knowledge representation repository, for example can be reused in arbitrary projects, also being non-document-centered. The knowledge base in contrast is specific to the current project and its application goals. A markup in principle can be reused in different document-centered knowledge acquisition projects, but often also is very project specific.

Table 5.5.1 summarizes the characteristics and dependencies of the different levels. Considering these (in-) dependencies, we discuss the activities of the multi-stage process illustrated in Figure 5.17 in more detail.

At the beginning, the structured representation (syntax tree) of the document(s) is created. For this task it is required to know the markup language and the document base. At that point, it is for example not necessary to know which knowledge representation (repository) or terminology is used.

On the terminology level, the syntax tree created by the syntax level is required. Further, for the incremental terminology resolution and knowledge base update discussed in Section 5.4 access to the terminology is required. For this activity, the information for the reference functions (Ref , Ref_o) are required. Those can easily be obtained by the syntax tree using simple markers on the respective nodes. Then, this level is independent of the detailed nature of the markup language.

The knowledge representation level actually creates the knowledge slices and inserts them into the knowledge base repository, that is, the compile functions $C_{\mathcal{K}}^{M, \mathcal{X}}$ introduced in Section 5.4.3 are called on the syntax tree. Therefore, these compile functions need to know about the markup language to make their way through the syntax tree. If knowledge representation level errors, such as inconsistencies, should be detected, this level also needs access to the existing knowledge base content, as inconsistencies in most cases can only be determined considering the entire knowledge base.

The knowledge base level finally aims to detect errors within the knowledge base behavior. That is, the behavior is compared to the intended behavior by the use of test cases or manual testing. It shows that the knowledge base level is independent of the syntax tree, markup language, and document base. As the activities on this level are independent of those DCKA specific components, the implementation workload for introducing a new markup is zero on this level.

Now, the required efforts on the distinct levels for the implementation of a new markup can be analyzed. It only affects the syntax and knowledge representation level, as depicted in Table 5.5.1. In consequence, for the implementation of a new markup language, reasonable implementation efforts are required only considering the these two level, if the reference functions can be provided conveniently.

5.5 A Meta-Model for the Declarative Implementation of Markups

	Syntax Level	Terminology Level	KR. Level	KB Level
Depending on	ML, DB	T, ST	KR, ML, ST , (KB)	KR, KB, KBT
Independent of ML	-	✓	-	✓

Table 5.1: The dependencies at different levels of multimodal knowledge document processing.

5.5.2 The Knowledge Markup Description Language

In Section 5.3 a method for the implementation of the syntax level activities has been discussed comprehensively: The KDOM schema. The KDOM schema also forms the basis of the declarative language for markup implementation introduced in this section. We extend the graphical language used for the examples for KDOM schema parsing in Section 5.3.7 by several language components to support the compilation process at a wider range.

The following definition of the *knowledge markup description language* is based on the MOF (Meta Facility Object) model, which is a standard method for the definition of visual languages provided by OMG (Object Management Group) [OMG02]. It is a meta-meta model to specify meta-models, which then define visual languages. The main concepts of the meta-meta model are classes, attributes and associations. Further, additional constraints can be added describing the possible relations between those concepts more precisely.

The UML diagram 5.18 illustrates the structure of the knowledge markup description language. The class *Type* plays a central role. The list of sub-types indicates that a hierarchical structure of types is built as already shown in the KDOM schema examples in Section 5.3.7.1. Each instance of the class *Type* is associated with exactly one instance of the class *Finder*. *Finder* is an abstract class providing the operation *findOccurrences()* implementing the parser function. As one generic implementation, the class *RegexFinder* is added. This kind of finder can easily be instantiated in a declarative way by attaching a regular expression as a string. For the description of new markups predefined *Type* and *Finder* instances should be provided in reusable and extensible libraries. The finder library should for example include the 'ALL' finder employed in Section 5.3.7.1. Reusable *Type* instances could be provided for syntactical elements of frequent use, for instance lines, line end comments, or table cells. To a *Type* instance optionally a *CompileScript* instance can be attached. This class also being abstract, requires implementation of the behavior of the operation *insertIntoRepository*. Hence, the knowledge markup description language does not have a entirely declarative nature considering this point.

5.5.3 Semantics

As discussed in Section 5.5.1, for the implementation of a new markup, the activities on the syntax, terminology (marginally), and the knowledge representation level need to be considered.

The semantics of the syntax level has already been discussed in detail in context of KDOM

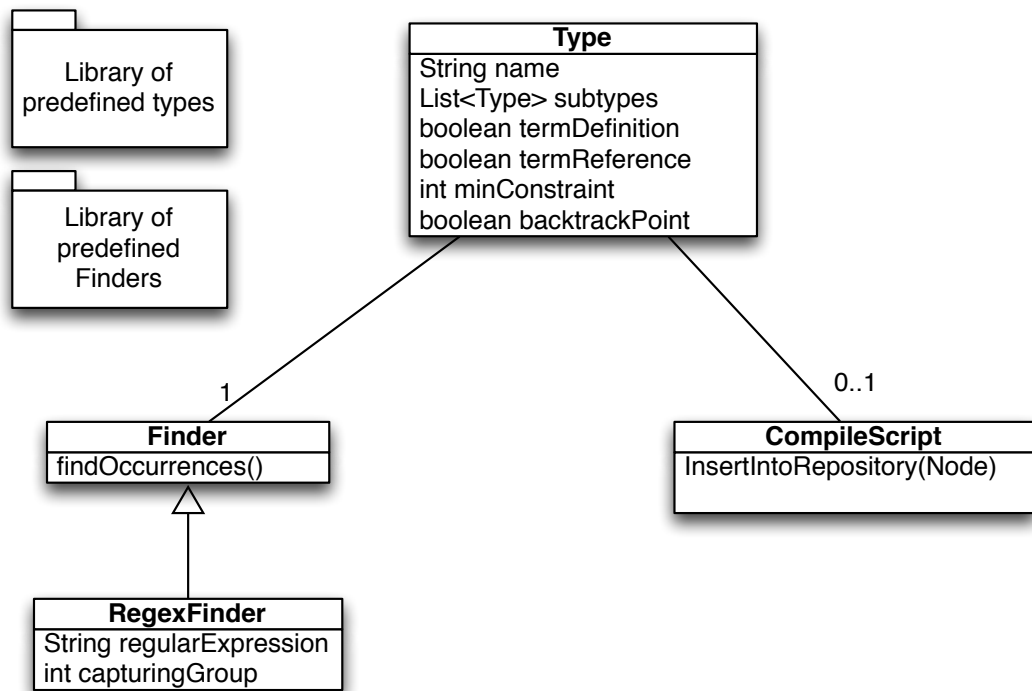


Figure 5.18: The meta-model for the Knowledge Markup Description Language.

schema parsing. Hence, the language elements *Finder*, *subtypes*, *minCardinality*, *backtracking*, as well as the libraries of types and finders deal with the syntax level and are already described in Section 5.3.2.

The terminology level can be handled to a far extent independently of the markup language. The only requirement is the presence of the reference functions (*Ref*, *Ref_o*), denoting which nodes define or reference objects. When the nodes for object references and definitions within a markup expression for a knowledge slice are flagged as such, a generic implementation becomes possible. These reference functions then collect all nodes with types possessing the flag *objectReference* or *objectDefinition*, respectively.

The knowledge representation level is addressed by the concept of the *CompileScript*. A *CompileScript* implements the compile function $C_{\mathcal{K}}^{M, \mathcal{K}}$ for a knowledge slice expression, i.e. inserts the corresponding knowledge into the knowledge base repository. This is done by analyzing the KDOM tree and creating the corresponding elements of the knowledge representation data-structure. The specification of this task can easily be performed by the use of the imperative programming paradigm. Therefore, these scripts should to be defined in a non-declarative way using a common imperative programming language, preferably the one used for the authoring environment core.

5.5.4 Example

In the following, we present a comprehensive example for the use of the knowledge markup description language. It is taken from the domain of (Ancient) History and aims to implement a markup for the formalization of historical time events. The following markup expression shows an example for a time event to be formalized, capturing informations such as name, date, description, and sources. A more detailed description of the markup is given in Section 7.5. The implementation of that markup using the markup description language is shown in Figure 5.19.

```

1 << Lamian War
2 323b-322b
3 => War
4
5 After Alexander's death Greeks
6 revolted against Macedonian rule
7 under the lead of the Athenians...
8
9 SOURCE: Paus:1,25,3-6
10 SOURCE: Diod:18,8-18
11 >>
12

```

After the main type, which is responsible for the segmentation, the *Title* type is defined. It is marked as *ObjectDefinition* indicating for the terminology resolution algorithm that here a new terminology object is defined. It uses the 'LINEFINDER', which we assume is available from the library of predefined finders and splits the content according to line breaks. As there is a min and max cardinality constraint with 1 defined in the subsequent line, at most one line will be allocated by the *Title* type. The *Date* type is implemented in a very similar way. As the first line of the overall markup block is already allocated by the *Title*, the next line will be captured as *Date*. Then the class of the event is handled by the type *EventClass*, which also has a subtype *Class-Reference* actually holding the object reference flag. As *EventClass* only has a max constraint, it is defined as optional. The *Description* type is provided with the 'ALL' finder, which is also assumed to be available from the library of finders. At this point, it is important to note the order of the subtypes of *TimeEventMarkup*. While the *Description* appears in the markup before the source information, it has the number 5, therefore being handled after the sources. In that way, the source content is processed and afterwards all the remaining text is allocated as description. The type *Source* does not have any cardinality constraints. Therefore, any number (including 0) of sources are allowed. The last component is the compile script *TimeEventScript* which is associated to the *TimeEventMarkup* type. An exemplary implementation of that compile script is shown in Listing 5.6 as pseudo-code:

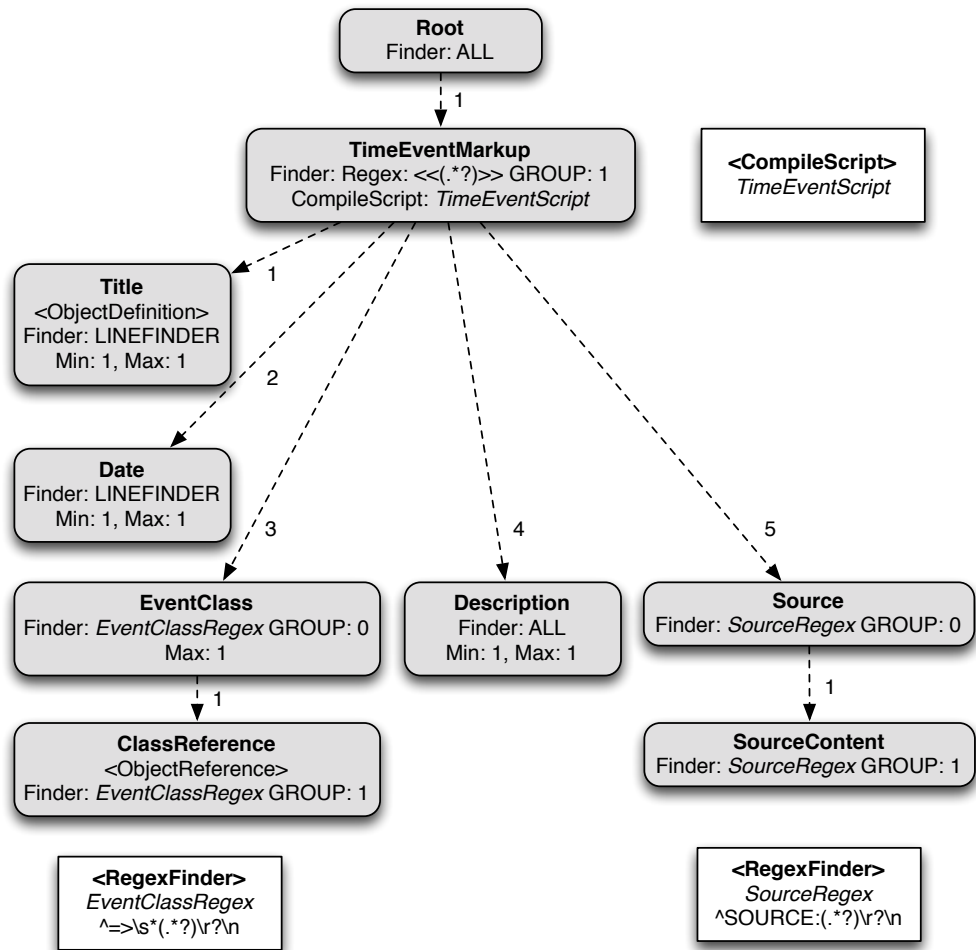


Figure 5.19: Implementation of a markup for formalizing time events in the domain of history.

```

1  TimeEventScript :
2
3  insert (Node<TimeEventMarkup> node)
4      TimeEvent event = new TimeEvent ()
5      event . setName (node . findSuccessorOfType (" Title "). text ())
6      event . setDate (node . findSuccessorOfType (" Date "). text ())
7      Node<ClassReference> classNode = node . findSuccessorOfType (" ClassReference ")
8      if (classNode)
9          event . setClass (classNode . text ())
10
11     List<Node<SourceContent>> sources = node . findAllSuccessorsOfType (" SourceContent ")
12     for (Node<SourceContent> sourceNode : sources)
13         event . addSource (sourceNode . text ())
14
15     KnowledgeBase . addTimeEvent (event , node)

```

Listing 5.6: An exemplary pseudo code implementation of the *TimeEventMarkup* compile script.

The *insert* function will be called by the incremental compilation algorithm. It assembles a *TimeEvent* instance by retrieving the data by navigating the KDOM tree. The function *findSuccessorOfType* is assumed to be provided by the API of the authoring environment core as discussed in Section 5.3.6 to search the corresponding KDOM subtree for nodes of the particular type. When all data has been collected, finally the created event is added to the knowledge base.

The schema shown in Figure 5.19 together with the script of Listing 5.6 forms a complete implementation of the time event markup. Therefore, of course the existence of a corresponding authoring environment core is presumed, which is able to interpret the knowledge markup implementation language using the techniques introduced in this chapter.

5.5.5 Meta-Level Authoring Support

The benefit of the knowledge markup description language can only be taken advantage of in practice if a suitable authoring tool for the language exists. In addition to the editing, the tool also needs to transform the created language constructs into a form which is executable by the target system, i.e., the document-centered authoring environment to be used for the markup. To help the system developer with the non-trivial task of markup implementation, a tool with good usability should be designed. In this section, we briefly discuss the requirements for such kind of tool. After that, a prototype demonstrating the suitability of the language for the implementation of knowledge markups is outlined.

Meta Tool Requirements: An authoring tool for the knowledge markup description language should provide a graphical editor for the visual language. Therefore, the types should be represented as nodes with the edges representing the sub-type hierarchy modeled as a list attribute in the diagram shown in Figure 5.18. The tool should provide convenient support for the definition of regular-expression-based finder components. Also the explicit declaration of those finders with support for their referencing by name and capture groups, as shown in the previous examples, are very helpful. A suitable way to make use of the predefined libraries for types and finders is drag-and-drop. Finders can be attached to types by dragging them from the library onto an existing type node. Predefined types can be dragged into the working area forming new nodes.

5 Techniques for the Implementation of DCKA

For effective markup implementation, a quick-test functionality should be provided. Therefore, the user is able to define one or multiple markup expressions as test objects. By triggering the quick-test function, the current version of the markup schema is applied on the example expressions. In that way, quick development and test cycles can be supported. This however, requires the engine of the target authoring environment to be available within the meta tool.

A Prototype Meta Tool for KnowWE: In the following, we briefly discuss a prototype editor for the document-centered authoring environment KnowWE, which is described in more detail in Chapter 6. The KnowWE meta tool editor allows to define new markup by the use of the knowledge markup implementation language in a similar way as discussed in Section 5.5.2. Figure 5.20 shows the implementation time event markup example from Section 5.5.4 within the KnowWE meta tool. Most parts from the example can be found on that screenshot. The compile script, actually inserting the knowledge into the knowledge base repository (RDF-triples in this example), is not visible in this view.

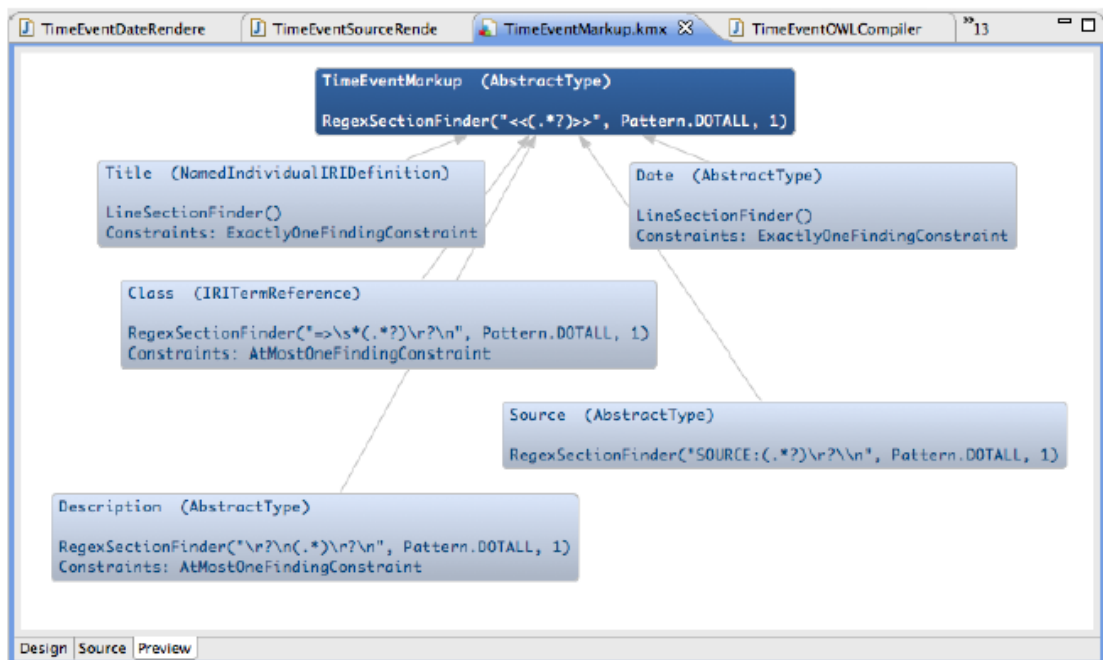


Figure 5.20: The time event markup of the HermesWiki defined in the KnowWE meta tool.

The meta tool, while still being an early prototype, compiles these markup definitions to a KnowWE plugin forming a full-fledged implementation of the markup, which only needs to be copied to the plugin-folder of a KnowWE installation. It can then be used on any document/page in any number or order to populate the knowledge base.

The meta tool is implemented as a plugin of the eclipse framework¹. The tool provides additional support for the user on the non-trivial task of knowledge markup implementation. To allow for quick feedback cycles within the markup development process, a quick test mechanism is included. The developer can specify a test document/wiki page containing examples of the target markup to be implemented. After triggering the quick test mechanism, the current version of the markup definition is compiled and an embedded KnowWE engine is launched, processing the specified test page. The result of the processing is instantly visualized using colors and mouse-overs for the types of the created nodes. Figure 5.21 shows a test page with above markup schema specification launched. The red underline at *War* indicates that the compilation

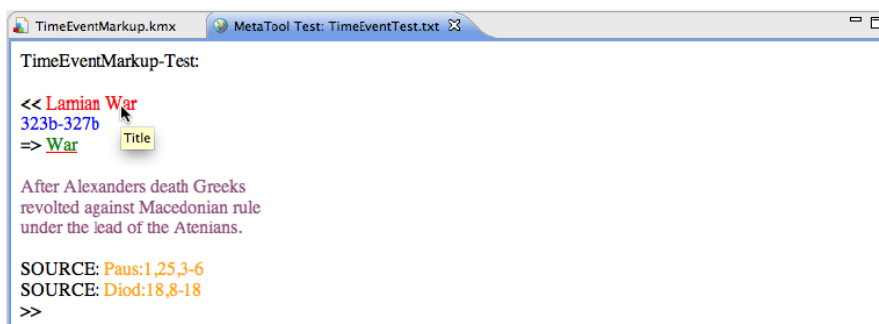


Figure 5.21: The time event markup in the quick test view.

algorithm has not found a proper definition of the term 'War' in the current document base (only consisting of these few lines in this quick-test example). This indicates, that the reference resolution on terminology level also has been performed by the system.

5.5.6 Discussion

In this section, we presented a language, which allows for the implementation of simple knowledge markups at very low implementation costs. That of course requires an authoring environment implementation which is able to interpret the language. The implementation of such a system, incorporating the algorithms and data-structures presented in Section 5.3 and 5.4, is a difficult and tedious task. That task, however, only needs to be conducted once, forming a generic core system that can be used for any kind of document-centered knowledge acquisition project. The (project dependent) definition of markups is then rather simple. This result is an important achievement for supporting the meta-engineering approach introduced in Chapter 4. As this approach proposes the frequent introduction and extension of new markups, a method for the cost efficient implementation of this technical task is essential. The use of the introduced knowledge markup description language for markup implementation is a non-trivial task. It requires some training to get used to the semantics. However, this task is performed by technically skilled knowledge engineers or system developers. Further, this expertise is independent of the

¹<http://www.eclipse.org>

5 Techniques for the Implementation of DCKA

project and domain. Hence, this skill, once acquired, can be employed in every new project alike. Therefore, this activity is not affected by the competency dilemma, which has been introduced as one of the major problems in knowledge engineering in Section 1.2.2.

From the software engineering perspective, a proceeding of this kind is called the *adaptive model* pattern. According to Fowler [Fow10], this pattern is helpful if an alternative computational model is desired. Once the computational model is implemented, it usually is used to interpret instructions in a custom language. This custom language allows to define the intentions more clearly and concise than possible in some existing language. A production rule system is one example for an adaptive model. The rule engine forms an alternative computational model and contains the general rule evaluation mechanism, including the checking of conditions and firing of actions. For an actual rule-based system, the application specific rules are added, often using a DSL for the definition of the rules. In the case of knowledge markup implementation at hand, the computational model is the parsing and (incremental) compilation of knowledge markups. The knowledge markup implementation language allows to write 'programs', which are interpreted by this custom computational model. The main benefit of this strategy is, that all recurring processes are implemented in the computational model, while context-specific information and processes are defined in the input language. In that way, on the one hand the maximum of reuse can be achieved and on the other hand a language that fits to the problem is given.

In the context of the existing work on customized knowledge acquisition tools, an editor for this kind of language can be considered as a meta tool for knowledge acquisition, i.e., a tool that allows to create knowledge acquisition tools [Eri92]. A categorization for knowledge acquisition meta-tools is given by Eriksson et al. [EPG⁺95], distinguishing *method-oriented*, *architecture-oriented*, and *ontology-oriented* meta tools. According to that categorization, the presented tool is an architecture-oriented meta tool.

6 An Authoring Environment for DCKA

In this chapter the document-centered authoring environment *KnowWE* [BRB⁺12] is introduced. At first, a brief historical overview of the system is given. After that, the software architecture is outlined. Further, a summary of the features, that KnowWE provides to support the knowledge engineering process, is given.

6.1 KnowWE: An Overview

In this section a brief overview of the history and the architecture of the system is given. Several techniques discussed in Chapter 5 are implemented KnowWE. For a complete description of the software architecture we refer to the documentation section of the system website.

6.1.1 History

The development of the KnowWE system has been started in late 2007 at the department of Artificial Intelligence and Applied Informatics at the University of Würzburg. It was a reimplementation based on the experiences made with a prototype system [BRP07a] that previously was developed since 2006. The first and major purpose of KnowWE was to provide document-centered development of knowledge systems in d3web¹, which is a framework and reasoning engine for diagnostic problem-solving. In late 2009 the denkbares GmbH² started with development activities on the system, considering feature extensions and quality management. The first quality-assured open source version of KnowWE was release by denkbares GmbH in September 2010 under the name *Hancock*. Since then, twice a year new releases with novel features and improvements are provided by denkbares GmbH.

Today beside research projects, KnowWE also is used in several industrial projects, as for example by Dräger Medical³ [HSM⁺12]. Further applications of KnowWE illustrating the document-centered knowledge acquisition approach are discussed in Chapter 7. Currently, KnowWE is extended to support the development of ontologies in RDFS⁴.

6.1.2 Architecture

In Section 3.3 the advantages of a wiki system as a basis for collaborative document-centered knowledge acquisition have been discussed. Therefore, KnowWE has been built up on an exist-

¹<http://www.d3web.de>

²<http://www.denkbares.com>

³<http://www.draeger.de>

⁴<http://www.w3.org/TR/rdf-schema/>

ing wiki engine, the *JSPWiki*⁵. JSPWiki is an open-source wiki engine implementation written in Java⁶ using the Java Server Pages technology⁷. KnowWE connects to an extension point provided by the JSPWiki architecture in a non-invasive way, allowing for special content processing and rendering. In that way, KnowWE is able to use many important functionalities of the reliable and still improving JSPWiki implementation, such as user management, persistence, versioning, and the basic web interface.

For the management of the document content, i.e., wiki content, the KDOM data-structure introduced in Section 5.3 is employed. Further, the top-down parsing algorithm discussed in Section 5.3.2 has been implemented. It is used for segmentation and parsing of almost all existing markups. Further, the algorithm for incremental compilation presented in Section 5.4 has been implemented for the system.

KnowWE is provided with a plugin framework (JPF⁸), which allows for the simple integration of new features as plugins. In that way, the meta-engineering approach introduced in Chapter 4, which is proposing the introduction of new custom extensions such as markups, can be supported.

6.2 Knowledge Acquisition with KnowWE

Following the document-centered knowledge acquisition paradigm as discussed in Chapter 3, in KnowWE knowledge is formalized by using knowledge markup languages. The markup languages can be used at any place in the wiki articles to create elements of the knowledge base, allowing for interweaving formal and informal knowledge in arbitrary way. Figure 6.1 shows an article taken from an exemplary car fault diagnosis wiki describing the concept *Clogged air filter*. The article contains informal content such as plain text and images (e.g., in the top half of the article) as well as formalized knowledge (rules at the bottom part of the article). The article can easily be edited by use of the general wiki editing interface or by other authoring assistance, which is further discussed in Section 6.2.5. KnowWE provides markup languages for the creation knowledge bases in d3web⁹ and ontologies in RDFS. For the d3web reasoner, markups for decision trees, set-covering models, decision tables, and rules are provided as discussed in Section 3.5.2 and in [BRP07b]. For a comprehensive and up-to-date documentation of all available markups including examples, we refer to the website *www.d3web.de*.

6.2.1 Manual Knowledge Testing and Use

For instant manual testing of the created knowledge base KnowWE provides an embedded interview component, which can be embedded into any wiki article. Figure 6.2 shows the interview interface, which is dynamically generated from the connected knowledge base. It allows the user to answer the input questions and instantly gives feedback of the derived solution concepts.


⁵<http://www.jspwiki.org>

⁶<http://www.java.com/>

⁷<http://www.oracle.com/technetwork/java/javaee/jsp/index.html>

⁸<http://jpf.sourceforge.net/>

⁹<http://d3web.sourceforge.net>



G'day (anonymous guest)
[Log in](#)
[My Prefs](#)

Your trail: [Demo - Master](#), [Demo - CloggedAirFilter](#), [Demo - Continuous Integration](#), [Main](#), [Demo - Main - Car Diagnosis](#)

Quick Navigation

[View](#)
[Attach \(2\)](#)
[Info](#)

[Edit](#)
More... ▼

Clogged air filter

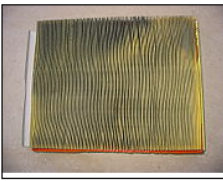
(adapted from Wikipedia)

General

The (combustion) air filter prevents abrasive particulate matter from entering the engine's cylinders, where it would cause mechanical wear and oil contamination.

Most fuel injected vehicles use a pleated paper filter element in the form of a flat panel. This filter is usually placed inside a plastic box connected to the throttle body with an intake tube.

Older vehicles that use carburetors or throttle body fuel injection typically use a cylindrical air filter, usually a few inches high and between 6 and 16 inches in diameter. This is positioned above the carburetor or throttle body, usually in a metal or plastic container which may incorporate ducting to provide cool and/or warm inlet air, and secured with a metal or plastic lid.



clogged air filter

Typical Symptoms

Typical symptoms for a clogged air filter are for example: Driving, unsteady idle speed and weak acceleration, but also problems when starting the car starting problems and an increased fuel consumption (based on average mileage) and the currently measured mileage or abnormal exhaust fumes.

A typical starting problem which is connected to this problem is a barely or not starting engine in combination with a starter that turns over.

A clogged air filter can cause black exhaust fumes which will turn the color of the exhaust pipe to sooty black.

```

IF Driving = unsteady idle speed
THEN Clogged air filter = P4

IF NOT (Driving = unsteady idle speed
OR Driving = weak acceleration)
THEN Clogged air filter = N5

IF (Exhaust fumes = black AND Fuel = unleaded gasoline)
THEN Clogged air filter = P5


IF ((Engine start = does not start
OR Engine start = engine barely starts)
AND Starter = turns over)
THEN Clogged air filter = P4

@package: demo
          
```

Repair Instructions

A clogged air filter needs to be replaced by a new one. Therefore, the air filter housing have to be found. It will be either square (on fuel-injected engines) or round (on older carbureted engines) and about 12 inches (30 cm) in diameter.

After locating the housing the screws or clamps on the top of it have to be



Home

Documentation / FAQ

Demos

- [Car Fault Diagnosis](#)
- [Body-Mass-Index](#)
- [Temperature Progression](#)

Administration

- [All pages](#)
- [Recent changes](#)
- [Plugins](#)
- [Recent changes](#)
- [Left menu](#)

● [Continuous Integration](#)

Documentation article

[attachment](#) [basicMarkup](#) [battery](#) [compile](#) [continuousIntegration](#) [coveringList](#) [expressions](#) [formulas](#) [imageMap](#) [interview](#) [knowledgebase](#) [package](#) [properties](#) [question](#) [quicki](#) [resource](#) [rule](#) [setcovering](#) [solution](#) [tables](#) [testcase](#) [timedb](#) [todo](#) [variables](#) [wikiMarkup](#) [xcl](#)

Tags ([edit](#)): [Demo](#)

KnowWE 20120604_02:29

JSPWiki v2.8.3-svn-19

Figure 6.1: A wiki page from a car fault diagnosis knowledge base in KnowWE.

In the shown example, the entered combination of inputs derived the solution concept *Bad ignition timing* as established. The solutions *Clogged air filter*, *Flat battery*, and *Leaking air intake system* are also suggested as potential solutions, while *Damaged idle speed system* is marked as an excluded solution.

The terminology knowledge forming the basis for this generated interview component for instance can be defined using the dash-tree markup, which already has been discussed in Section 3.5.2.1. An excerpt of the knowledge generating die interview displayed in Figure 6.2 is shown in Listing 6.1.

► **General** ■

▼ **Observations**

Exhaust fumes	<u>black</u> blue invisible unknown
Exhaust pipe color	brown <u>grey</u> light grey sooty black unknown
Exhaust pipe color evaluation	abnormal <u>normal</u> unknown
Fuel	<u>diesel</u> unleaded gasoline unknown
Average mileage /100km	<input type="text"/> <u>unknown</u>
Num. Mileage evaluation	<input type="text"/> unknown
Mileage evaluation	slightly increased normal increased unknown
Real mileage /100km	<input type="text"/> unknown
Engine noises	knocking <u>ringing</u> no /else unknown
Engine start	engine barely starts engine starts <u>does not start</u> unknown
Starter	does not turn over <u>turns over</u> unknown
Driving	<u>insufficient power on partial load</u> insufficient power on full load unsteady idle speed low idle speed delayed take-off <u>weak acceleration</u> no /else unknown

► **Technical Examinations**

Derived Solutions

- Bad ignition timing
- Clogged air filter
- Flat battery
- Leaking air intake system
- Damaged idle speed system
- (*) Exhaust pipe color evaluation = normal

ShowSolutions ▼
 Edit Markup

Figure 6.2: The interview component for manual knowledge base testing.

```

1  - "Real mileage /100km" [num]
2  - Engine noises [oc]
3  -- knocking
4  -- ringing
5  -- "no /else"
6  - Engine start [oc]
7  -- engine barely starts
8  -- engine starts
9  -- does not start
10 - Starter [oc]
11 -- does not turn over
12 -- turns over
13 ...

```

Listing 6.1: Terminology definition for the Car Diagnosis example using dash-tree markup.

For developed ontologies KnowWE provides an inline-query mechanism to summarize the knowledge of the ontology as a dynamic content element. Using a markup, which is based on the SPARQL¹⁰ language, queries can be defined within the wiki pages. Those queries are evaluated on page load on the current version of the developed ontology. The result of the query is displayed in the view of the wiki article at the position of the query.

6.2.2 Automated Testing by Continuous Integration

As a modern knowledge engineering environment, KnowWE supports an agile knowledge engineering approach. Here, knowledge bases are developed in an evolutionary manner, always maintaining an executable and correct version at a certain level of competency. In this context, (automated) testing is very important to ensure successful development cycles in the evolutionary process. We adopted the continuous integration practice known from software engineering into the knowledge engineering tool KnowWE. A continuous integration dashboard in the wiki is used to define a collection of quality tests (for validation and verification). As a special knowledge markup, the dashboard can be configured easily to support tailored quality management for the respective project. Registered automated tests are performed on the current version of the wiki knowledge base and give verbose feedback to the knowledge engineers by status messages on the dashboard, as shown in Figure 6.3.

At any time, the dashboard displays the current state of the wiki knowledge base with respect to quality at one glance. Also the history of builds is listed on the left panel of the dashboard. Older builds can be inspected by clicking on the build number. For the selected build the applied tests are shown in the center of the dashboard. In case of errors, the tests give detailed reports on the errors as well as links for further investigation and debugging of the issue. In Figure 6.3, the top two tests have been passed successfully, while the lower two tests have failed showing more details explaining the actual problem. The tests can be activated by three trigger-modes *onChange*, *onSchedule*, and *onDemand*. In the mode *onChange*, the tests are executed after each modification of a wiki article, that changed the knowledge base. This mode provides the most immediate feedback possible. However, for very time consuming tests this mode can yield inconvenient delays. The mode *onSchedule* executes the tests on a regular basis according

¹⁰<http://www.w3.org/TR/sparql11-query/>

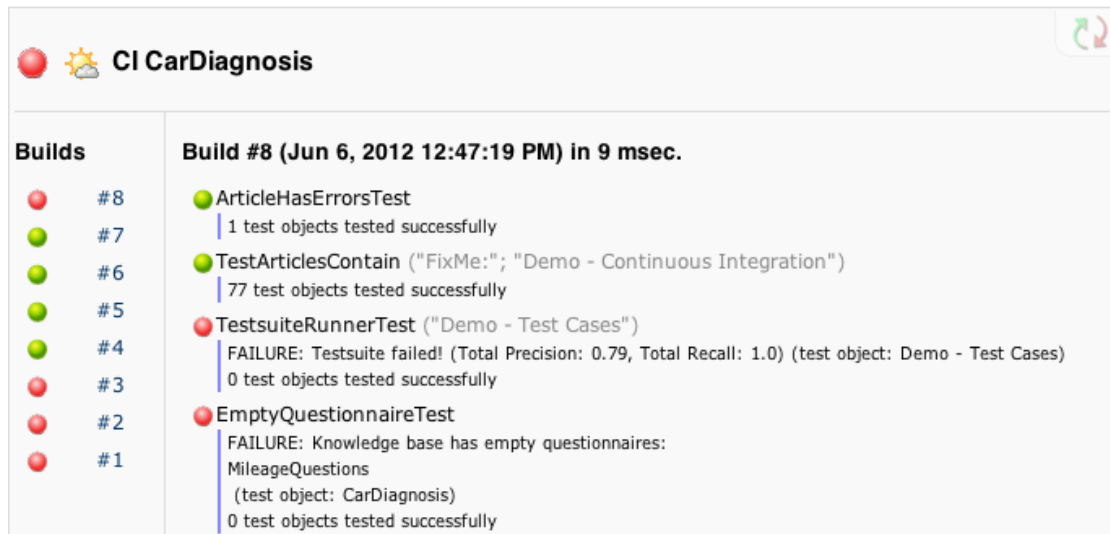



Figure 6.3: The continuous integration dashboard of KnowWE showing messages of the current test runs and the history of the previous development stages.

to a specified schedule, for instance every night. This mode is preferable also for tests with considerable high execution time. Further, in the mode *onDemand* all responsibility for test execution is left to the user, since the user has to explicitly start a continuous integration run. The user has to decide, when the execution is reasonable, which often is an option for tests with high runtime (considering sufficiently experienced users). It is important to note, that the user can define different dashboards, for instance, one for quick tests running *onChange* and another one for executing larger/time-consuming tests *onSchedule*.

Additionally to the dashboard, located on a specific wiki page, KnowWE provides a *CI-Daemon* (daemon for continuous integration), which can be connected to a dashboard. The CI-Daemon is always visible in the KnowWE user interface, basically only showing a colored bubble (green, red, or gray) representing the current state of the connected dashboard. In Figure 6.1 the CI-Daemon is visible as a green bubble on the left of the page below the navigation menu. In this way, the users are always aware of the current quality state not requiring to regularly visit the dashboard article. A very important category of tests for knowledge bases are the competency tests, which can be implemented by (sequential) test cases [Bau11]. Figure 6.4 shows a markup for the definition of sequential test cases in KnowWE. During execution, the test case is performed line-by-line. Equal signs express assignments of input data, added to the current testing session. Expressions containing brackets are expected derivations. The test fails, if the expected derivations do not match the actual ones. That way, input-output behavior of a knowledge base can be covered by automated competency tests, which can be attached to a continuous integration dashboard easily.



```

View Attach (1) Info Edit More...
Test cases
"Leaking air intake system (Demo)" {
  Driving = insufficient power on partial load :
  Leaking air intake system (suggested);
  Driving = unsteady idle speed :
  Clogged air filter (suggested);

  "Check: Air filter." = ok,
  "Average mileage /100km" = 10,
  "Real mileage /100km" = 12,
  Driving = insufficient power on full load :
  Leaking air intake system (established);
}

"Clogged air filter (Demo)" {
  Exhaust pipe color = sooty black,
  Fuel = unleaded gasoline :
  Clogged air filter (suggested);

  Driving = unsteady idle speed,
  Driving = weak acceleration,
  "Check: Ignition timing." = ok :
  Clogged air filter (established);
}
@package: demo

```

Figure 6.4: Markup for the definition of sequential test cases for competency tests.

6.2.3 Debugging

For rule-based knowledge bases in d3web, KnowWE provides a comfortable debugging mode. The debugging mode for rules consists of two components. The first component is a debugging panel, which allows for each term to review all rules setting a value for this term. Also, the current value of the term and the current state of the rules are displayed. This kind of debugging component is also known from other rule-based systems development tools. It however has no direct relation to the document contents actually containing the knowledge. For this purpose a special rendering component for the rule markup is included. To visualize the reasoning mechanism, it highlights the rules markup according the execution state of the rule condition using color codes.

6.2.3.1 The Rule Debugger

The rule debugger component is shown in Figure 6.5 using the car fault diagnosis example again. The surface consists of four components, being strongly interrelated. The debugger is designed to start with the output concepts of the knowledge base, being the solutions. The top line (1) shows the concept, that is currently tracked, is shown as using the breadcrumb pattern [MR02]. Currently, the solution *Clogged Air Filter* is selected, allowing to analyze all rules, which are (potentially) assigning a value to this concept.

In second component (2) all concepts are listed, which appear in a condition of a rule that can assign a value to the selected target concept. Therefore, all these concepts are relevant when

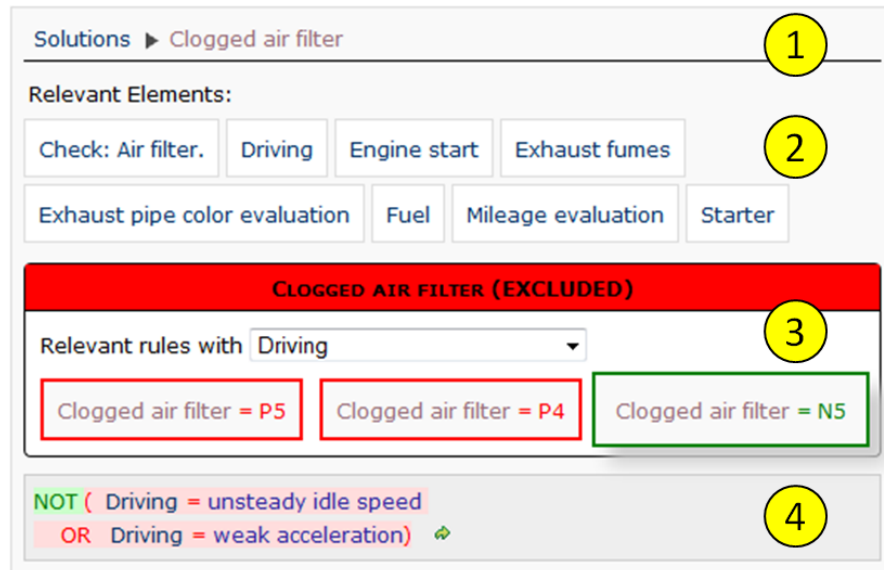


Figure 6.5: The debugging panel for the debugging of rule traces.

debugging this concept. One click on a concept of the list selects it as the active element in the breadcrumb. In that way, the user can recursively navigate through the network of concepts, which is formed by the rule set in the knowledge base.

The next component (3) represents the rules relevant for the target concept. Within the header line of the area the current value of the target concept is shown. Underneath, each rule is represented by a rectangle displaying the value assignment of the respective rule action. The border of the rectangles indicate the state of the rule, being fired (green), cannot fire (red), and cannot be evaluated (gray). The rules can be filtered according the relevant elements for clarity. The default filter setting is 'all', showing all rules.

When clicking on one of these rule heads, the condition of the rule is displayed on the bottom area (4) of the debugger. It uses the similar color coding mechanism as the rule markup component described in the following.

6.2.3.2 The Rule Markup Rendering Component

Figure 6.6 shows an excerpt of the wiki page from Figure 6.1 with the debug rendering mode enabled. Green bars on top and bottom indicate at first glance which rules have been fired. Further, the color coding of the conditions shows the evaluation state in more detail. It is applied recursively on the sub-conditions if the condition is a complex logical expression composed using boolean operators. Operators and comparators are painted in red or green if the condition evaluated false or true respectively. Additionally, the markup expression fragments are highlighted by the respective background color. This color coding allows the user to find out at one glance why some rule is or is not firing.

For interactive debugging, the rendering component additionally provides the possibilities to

manually change (by force) the value of each occurring concept. As shown in Figure 6.6 for concept *Fuel*, which currently has the value *unleaded gasoline*, the user can instantly change or unset the current value of a concept. With this interview component included in the markup presentation, the behavior of the rules on different value configurations can be tried out efficiently.

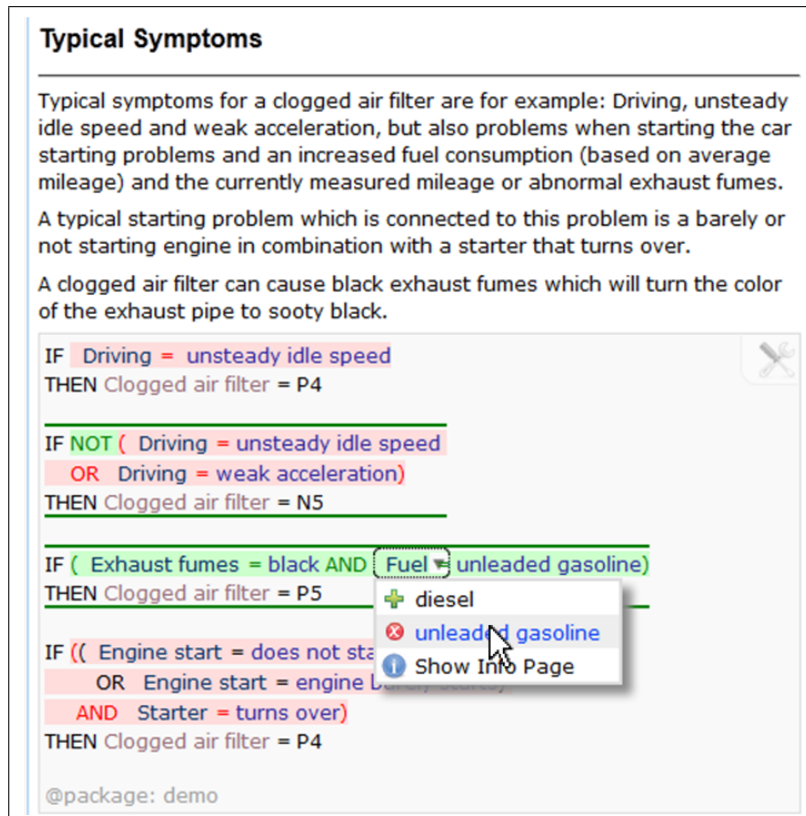


Figure 6.6: The debugging rendering component for the rule markup.

6.2.4 Refactoring

The refactoring of the document base is a fundamental task, necessarily emerging during agile knowledge system development using document-centered knowledge acquisition, especially when employing the meta-engineering approach. In Section 3.2.2.2 we discussed the two categories of refactoring, being document level and knowledge base level refactoring. The technical task for both categories however is quite similar: The structured reorganization of the document contents towards a specific target structure. The KDOM data structure (c.f. Section 5.3) employed in KnowWE is a very helpful basis for this task [RBP09].

For knowledge bases of considerable size automated solutions are needed. KnowWE provides three methods for performing automated refactoring:

- **Term Renaming:** One rather specific refactoring task often required is the renaming of an existing term consistently across the entire document base. A corresponding refactoring tool is integrated in KnowWE shown in Figure 6.8. It allows for each term to specify a new term name and to start the wiki wide replacement of the term name. The renaming mechanism sets up onto the KDOM data structure in the same way as the compilation mechanism creating the knowledge base. In that way, consistent renaming without change of the knowledge base semantics is guaranteed.
- **Replacement Tool:** For general text replacements a corresponding tool exists, which is based on regular expressions. It works in a similar way as known from many text processing tools. The matches for all wiki pages are presented to the user for further selection.
- **Custom Scripts:** Not all kinds of refactorings can be performed by the replacement tool discussed above. For instance, the transformation of existing knowledge into a new markup language can be a complex transformation task. In that case dedicated refactoring scripts, working on the KDOM tree structure and generating the new document contents, need to be written. A script of this kind can be hooked up into the system as a plugin using an extension point of the plugin framework. For the development of the script, the knowledge engineer can use a copy of the document base for testing, before executing the script on the actual head version of the document base.

Refactoring of knowledge bases in general is a complex and challenging task. Except for rather trivial actions, such as the renaming of a term, it usually needs to be performed by a skilled knowledge engineer.

6.2.5 Authoring Support

In addition to the basic wiki editing interface, KnowWE provides different kinds of editing support. Some of them are again implemented as extensible frameworks that allow for the integration of support for new markups.

6.2.5.1 Instant Editing

The system provides *instant edit* functionality that allows in-place editing of a section, i.e. a coherent part of an article, within the view of the wiki page as shown in Figure 6.7. Based on this functionality an edit mode is provided that allows to in-place edit any contents by single clicks. In that way, the switch to the normal source editing interface, which is often inconvenient for larger pages, is obsolete.

6.2.5.2 Table Editing

Typically, the editing of tables is difficult when using the standard text markup for tables. Therefore, KnowWE provides instant editing capabilities for tables in a WYSIWYG style allowing

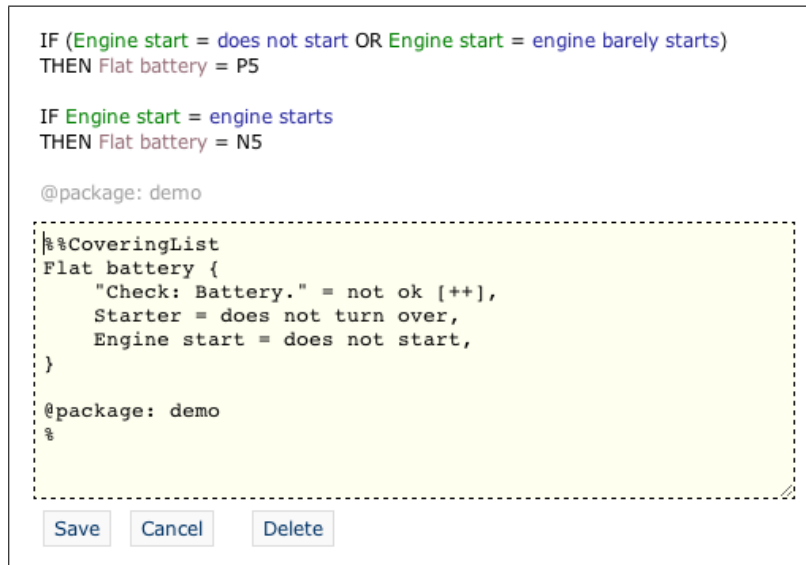


Figure 6.7: Authoring parts of an article using the instant edit feature.

each cell to be edited by one click as shown in Figure 6.9. The table content is stored within the wiki page source in standard wiki markup.

6.2.5.3 Code Completion

Additionally, a code completion mechanism supports the user to create markup sections in the text editing panel. The code completion performs suffix completion of all knowledge base objects that are known to the system. It also support reasonable completion for the assignment of values to valued objects, such as choice questions in d3web.

6.2.5.4 Drag & Drop

In Section 3.1.6.4 we emphasized the importance of authoring assistance for creating textual markups. Beside code completion, drag-and-drop editing was mentioned as a convenient method to prevent exceeding typing workload. In KnowWE a small framework for drag-and-drop editing was integrated. In works in combination with the term browser, which is shown in Figure 7.6. There, the frequently used terms are listed or additional terms can be added by a auto-completed search slot. For any markup, an insertion script can be added as a plugin via a extension point. The insertion script defines, how a drop term triggers an edit operation on the content. For editing, any term can then be dragged into any section, that is provided with an insertion script, triggering the corresponding editing operation.

6.2.5.5 Term Overview

Often, it becomes necessary to obtain an overview of the occurrences and uses of a particular domain concept. Figure 6.8 shows an overview page for the concept *Leaking air intake system*, that is dynamically generated when requested by clicking on the concept name in the wiki. Besides the pure information about the concept, also small refactoring capabilities are available: In the bottom part of the info page, the user can see an overview of the wiki articles, where the concept is used (links yield to the particular occurrences in the wiki). In that way, the user can easily get an overview of the role of a concept within the knowledge base and is then able to navigate quickly to the next document location of interest.

7 Case Studies

In this chapter we describe different case studies of knowledge engineering projects where document-centered knowledge acquisition with meta-engineering has been employed. The projects show quite heterogeneous settings considering domain, number of participants, project duration and knowledge representation. They cover the domains of medicine, history, chemical substance management, and computer device assistance for handicapped people. For each project a brief introduction to the scenario and a short summary of the meta-engineering process is given.

7.1 ESAT: Assisting Technologies for Handicapped Persons

ESAT (Expertensystem für Assistierende Technologien [german]) is an expert system designed to determine an appropriate set of human-computer interaction devices for a person with specific physical handicaps.

7.1.1 Introduction

Interaction with computers has become a natural part of most people's daily live, not only for professional but also private activities like communications, entertainment, or shopping. Human-computer interaction is highly standardized, in general presuming a person's full physical and sensorial capabilities. Many handicapped people are not able to use the common human-computer interaction devices, such as standard keyboard, mouse, and visual screen. Nevertheless, the use of computers for those people is at least as important and beneficial. Therefore, numerous alternative human-computer interaction devices have been designed, which make use of different aspects of physical capabilities available. The determination, which (combination of) devices are suitable for a specific person, however is not trivial. As many devices require a considerable amount of training for efficient use, an approach of extensive evaluation of all potential devices is very time consuming, demanding, and annoying for the target person. For this reason, an extensive study has been made to collect the knowledge about which devices should be recommended for what kind of capabilities being available. By the help of this knowledge for a profile of a person's capabilities a set of suitable devices can be determined.

7.1.2 Application Scenario

Manual application of that knowledge for the determination of suitable devices is still tedious. Therefore, an executable knowledge base is desired to perform this task. With a knowledge-based system of this kind, a detailed profile of the physical capabilities (e.g., visual or motor abilities) for a person is entered and a combination of input and output devices, which together

provide optimal computer interaction, shall be proposed instantly. More details about the application scenario are given by Kreutzer [Kre12].

7.1.3 Knowledge Base Structure

ESAT is a rule-based expert system. For the capabilities of the person a comprehensive set of input attributes with qualified ranges are defined. Every human-computer interaction device can be derived to be part of the solution, which consists of one (or multiple) input devices and one (or multiple) output devices. For the derivation of the solutions several heuristics, that have been established within the theoretical study, are transformed into the rule-based knowledge base.

7.1.4 The Meta-Engineering Process

The actual implementation of a corresponding executable knowledge system has started in spring 2011. Currently, the ESAT knowledge base just has been completed and the system will be launched for a testing phase at the project's initiator (FAB¹). The knowledge base has been implemented mainly by a single person using the document-centered knowledge acquisition approach with the system KnowWE. For knowledge representation on the symbol level production rules in d3web are used. In total, the ESAT knowledge base currently contains 654 rules, distributed topically on numerous wiki documents. Also in this single-user context the possibility of free structuring allows for reasonable and clear distribution of the knowledge to facilitate long-term maintenance.

7.1.4.1 Knowledge Acquisition Architecture

The knowledge acquisition architecture of the ESAT document base can be illustrated by showing the start-up page. All the documents containing ESAT knowledge are available from this starting page, which is shown in Figure 7.1. The documents are organized by categories, starting with the document providing the testing interface for manual knowledge base testing. Underneath, the input attributes, which allow to verbosely describe a clients physical capabilities, are defined. They are partitioned in documents dealing with the sub-categories vision, hearing, motor and haptic abilities, and general skills (e.g., braille). Then the largest category is given, dealing with the assisting devices. Overall, there are about 50 different types of input and output devices, organized in the sub-categories character input devices (e.g., various kinds of keyboards), sensor devices, pointing devices, visual output devices, audio output devices, and haptic output devices. Each device is described on a distinct document, also containing the rules relevant for the derivation of the particular device. The next category refers to the five major heuristics, that have been established within a theoretical study, describing solutions for major categories of handicaps. For each heuristic a description and a set of rules is defined on a distinct document, forming the core of the derivation knowledge. The testing framework for continuous integration discussed in Section 6.2.2 is extensively used to guarantee the safe development process by uncovering undesired side-effects of modifications. That includes the definition of at

¹<http://www.vo-fab.at/>

least one sequential test case for each device and heuristic. The documents containing the test cases and a link to the testing control dashboard are stated at the bottom category.

The documents describing a device all have similar structure, consisting of three parts:

- **General:** A general description (in German "Allgemein") of the device is given, illustrated by images. It discusses the way it is used, including the advantages and disadvantages.
- **Specification:** The technical specification (in German "Spezifikation") is provided presenting detailed technical attributes of the concrete device series, such as scale, weights, or ranges. Further, the contact data for ordering and the pricing information is included.
- **Code:** The third category (in German "Codeblock") contains the formal knowledge that is compiled to the expert system. Usually, it contains the definition of the knowledge base object representing this device along with the derivation knowledge, determining when this device will be part of the solution.

Figure 7.2 shows the document describing a device called head/chin button (in German "Kopf- und Kinn-taster"). On the figure the categories *General* (A) and *Code* (B) are illustrated.

7.1.4.2 Markups

During the development, the meta-engineering process established a few new markups that simplified the definition of the knowledge within the documents, improving conciseness and understandability. In the following, two examples of markups, being introduced for this project, are outlined.

Multi-Rule Markup: The markups created within the design activities of the meta-engineering process aim to simplify the knowledge authoring. Often the invented markups are very specific to the current project. However, it may happen that markups or markup extensions become helpful, which are not domain or project specific but nevertheless not yet existing. That was the case with the multi-rule markup introduced in the context of the ESAT project. It is an extension to the existing rule markup for d3web rules in KnowWE, which is illustrated in Section 6.2.3. The extension allows to define multiple rules with same condition in one expression. While before, for each action a distinct rule expression repeating the condition had to be written, now in a rule markup expression a list of actions, separated by semicolons, can be defined. With the introduction of the markup, existing rules with identical conditions have been aggregated when being located on the same document. This (content level) refactoring left the executable knowledge base unchanged as the compiler in the background creates one d3web rule for each action. Hence, there is no risk to change the operational semantics by this restructuring. The step significantly reduced the amount of rule markup expressions within the overall document base. In Figure 7.13 the page *ESAT_Heuristik_H04* is shown, containing two multi-action rules providing 11 and 15 actions, respectively. The implementation of the markup extension has been contributed to the main KnowWE code base and now is also used in other projects.

7 Case Studies

ESAT_Start

Willkommen, [knowweadmin1](#) (nicht angemeldet) | [Anmeldung](#) | [Meine Einstellungen](#)

Bisher besucht: ESAT_Monitor, ESAT_Start, ESAT_Heuristik_H03, ESAT_Heuristik_H04

[Anzeigen](#) | [Anhänge \(3\)](#) | [Info](#) | [Edt. Mode](#) | [Bearbeiten](#) | [Weiter...](#)

virtual office **FAB Jugend**

ESAT - Expertensystem für Assistierende Technologien (VO-Release)

ESAT ist ein Expertensystem, basierend auf KnowWE, welches für das Virtual Office des FAB entwickelt wurde. Wissenschaftliche Grundlagen bilden die Arbeiten von Frydada de Piotrowski und Kreuzer. ESAT verwendet heuristisches Wissen, um geeignete Assistierende Technologien für Klienten des Virtual Office entsprechend ihrer Fähigkeiten vorzuschlagen.

ESAT Fragebögen und Definition der Wissensbasen **Testing Interface**

- ESAT Fragebogen

ESAT Klientenprofil

Das Klientenprofil enthält die Definition der Fähigkeiten, welche in den Fragebögen abgefragt werden. Diese Fähigkeiten sind gruppiert in Motorik, Sehsinn, Gehörsinn, Haptik und sonstige Fertigkeiten und Vorlieben.

- Motorik
- Sehsinn
- Gehörsinn
- Haptik
- Fertigkeiten und Vorlieben

Definition der Anzeigen-Reihenfolge für den Fragebogen

Assistierende Technologien

Definitionen von Assistierende Technologien (AT), welche aufgrund des Benutzerprofils nach Heuristiken und Regeln dem Klienten zugeordnet werden. Die ATs werden in Technologien zur Zeicheneingabe, Sensoren, Technologien zur Positionseingabe sowie in Technologien zur Ausgabe welche den Sehsinn, den Gehörsinn und die haptische Wahrnehmung ansprechen unterteilt.

- Computerbenutzung nicht möglich

Technologien zur Zeicheneingabe

- Standardtastatur
- Kompakte Tastatur
- Großsegmenttastatur
- Großfeldtastatur
- Minutentastatur
- Half-QUERY-Tastatur
- Einhändige Akkordtastatur
- Brailletastatur
- Tastaturvariationen
- Automatische Spracherkennung
- Bildschirmtastatur
- Einrastfunktion
- Tastatur-Wiederholparameter
- Wortvorhersage
- Optische Texterkennung (OCR)

Sensoren

- Drucktaster
- Daumen- und Griffaster
- Kopf- und Kinntaster
- Swip- und Blasenensor
- Akustik-Sensor
- BIM-Sensor
- Scansoftware und -hardware

Technologien zur Positionseingabe

- Maus
- Trackball
- Touchpad
- Joystick
- Mund- und Kopfjoystick
- Tastenmaus
- Augensteuerung
- Kopfsteuerung
- Touchscreen
- Klick-Einrasten
- Mausklick-Emulator
- Doppelklickgeschwindigkeit
- Mauseizergeschwindigkeit

Technologien zur Ausgabe für den Sehsinn

- Monitor
- Auflösung reduzieren
- Anpassen von Helligkeit und Kontrast
- Mauseinstellungen (Mauspuren, großer Mauszeiger)
- Negativkontrast des Monitors
- Zweiter Monitor
- Bildschirmlupe
- Visuelle oder auditive Rückmeldung

Technologien zur Ausgabe für den Gehörsinn

- Kopfhörer mit ANR
- Frequenzabhängige Lautstärkempassung
- Balanceregung
- Monobetrieb
- Screenreader
- TTS-Software

Technologien zur Ausgabe für die haptische Wahrnehmung

- Braillescheln & Brailldrucker
- Gerät zur taktilen Darstellung von Grafik

Heuristiken

Heuristiken sind Faustregeln und Best Practice, welche für eine schnelle Problemlösung angewandt werden. Die Heuristiken in diesem Programm beruhen auf einer Arbeit von Frydada de Piotrowski. Nicht alle Regeln sind direkt bei den Assistierenden Technologien festgelegt. Teilweise ist es ökonomischer einer bestimmten Konstellation vor Eigenschaften mehrere ATs zuzuordnen, als für jede AT die Konstellation neu zu beschreiben.

- Heuristik H02 - ohne Gesichtsfeldausfall
- Heuristik H03 - mit peripherem Gesichtsfeldausfall
- Heuristik H04 - mit zentralem Gesichtsfeldausfall
- Heuristik H05 - zentraler & peripherer Gesichtsfeldausfall mit Quadrantenausfall
- Heuristik H08 - Aufmerksamkeitsdefizit

Testfälle

- Testfälle für Technologien zur Zeicheneingabe und generelle Computerbenutzung
- Testfälle für Sensoren
- Testfälle für Technologien zur Positionseingabe
- Testfälle für Technologien zur Ausgabe für den Sehsinn
- Testfälle für Technologien zur Ausgabe für den Gehörsinn
- Testfälle für Technologien zur Ausgabe für die haptische Wahrnehmung
- Testfälle für Heuristik H02
- Testfälle für Heuristik H03
- Testfälle für Heuristik H04
- Testfälle für Heuristik H05
- Testfälle für Heuristik H08

- Manuelles Prüfen der Testfälle

● CI-Dashboard (Continuous Integration)

CI-Dashboard für die Prüfung abstrakter Fragen


Figure 7.1: The ESAT starting page with links to all other documents structured as categories.

Anzeigen [Anhänge \(2\)](#) [Info](#) [Edit Mode](#) [Bearbeiten](#) [Weitere...](#)


Kopf- und Kinttaster

Allgemein [Spezifikation](#) [Codeblock](#)

Kopf- und Kinttaster werden meist als Druck- oder Blatttaster angeboten. Die Unterschiede zu anderen Tastern sind, dass Blatttaster auf einem Stativ montiert werden und Drucktaster meist über Klettverschlüsse oder Sicherheitsnadeln verfügen, um sie an der Kleidung zu befestigen. Das hervorstechende Merkmal gegenüber anderen Tastern ist, dass sie meist gepolstert sind, um keine Druckstellen auf der Haut hervorzurufen.



Wobble-Switch der Firma Prentke Romich Deutschland



Die Bedienung eines Wobble-Switch mit dem Kinn (Ingenieurbüro Dr. Seveke)

Zielgruppe

Diese Technologie ist vor allem für Menschen gedacht, welche den Kopf als bevorzugtes Eingabegerät benutzen (mit Krankheitsbildern wie z.B. Tetraplegie). Kleine Unstetigkeiten stellen kein Problem dar, sofern die Geräte in genügendem Abstand ange...

(A)

Anzeigen [Anhänge \(2\)](#) [Info](#) [Edit Mode](#) [Bearbeiten](#) [Weitere...](#)

Kopf- und Kinttaster

Allgemein [Spezifikation](#) [Codeblock](#)

Ergebnis-Objekte

Sensoren Solu
 - kopf_kinntaster Kopf- und Kinttaster
 @package ESATKB1

Link

kopf_kinntaster.link = Wiki.jsp?page=ESAT_Kopf-Kinntaster @package ESATKB1 Prop

Mindestanforderungen für Kopf- und Kinttaster

```
// M23
IF blindness = No AND
(
  bodypart_for_operation = head
  AND eval(gt(head_steadiness,greatly_impaired))
  AND head_motoric = unimpaired
)
THEN kopf_kinntaster = P5

IF kopf_kinntaster = SUGGESTED
OR kopf_kinntaster = ESTABLISHED
THEN sensor_usable = Yes
@package ESATKB1
```

(B)

Figure 7.2: A document of ESAT describing a device called head/chin button ("Kopf- und Kinttaster"). The general description (A) and the formal knowledge (B) is shown.

Anzeigen Anhänge Info Edit Mode Bearbeiten Weitere... ▼

Heuristik H04 - zentraler Gesichtsfeldausfall

Die Heuristik H04 beschreibt die Verwendung von Assistierenden Technologien, wenn beim Klienten ein zentraler Gesichtsfeldausfall vorliegt. Es wird die Größe des zentralen Skotoms mit einbezogen.

Heuristik H04

```

IF visual_field_loss = cent
  AND scotoma_size > 1
  AND scotoma_size < 10
THEN aufloesung_reduzieren = P5;
  tastatur_grossschrift = P5;
  kontrastanpassung = P5;
  monitor20 = P7;
  monitor19 = N5;
  monitor17 = N5;
  monitor15 = N5;
  monitor12 = N5;
  monitor10 = N5;
  monitor08 = N5;
  monitor_usable = Yes

IF visual_field_loss = cent
  AND eval(ge(scotoma_size,10))
  AND eval(lt(scotoma_size,20))
THEN mauszeiger_gross = P5;
  monitor_negativkontrast = P5;
  bildschirmlupe = P5;
  tastatur_grossschrift = P5;
  tastatur_negativkontrast = P5;
  tts_software = P5;
  screenreader = P5;
  monitor20 = P7;
  monitor19 = N5;
  monitor17 = N5;
  monitor15 = N5;
  monitor12 = N5;
  monitor10 = N5;
  monitor08 = N5;
  monitor_usable = Yes
@package ESATKB1

```

Diese Seite (Version-1) wurde zuletzt am 09-Okt-2012 13:04 von Unbekannt geändert. ▲

Figure 7.3: The document describing heuristic H04 about central visual field loss (zentraler Gesichtsfeldausfall) using the multi-rule markup.

Monitor Markup: In ESAT a large number of different monitors are available as display solutions for particular visual capabilities, for instance considering a limited visual field. In the reasoning process for each monitor multiple attributes are required, such as width or area. As d3web only provides limited support for the modeling of complex objects, i.e., objects with multiple slots, these objects are modeled by creating multiple values connected by a naming convention. By using the standard markup for creating d3web knowledge bases the definition of a monitor reads like follows:

```

1 %%question
2   monitor15_a2 [num] <abstract>
3   monitor15_q [num] <abstract>
4   monitor15_width [num] <abstract>
5   monitor15_height [num] <abstract>
6   %
7
8 %%property
9   monitor15_width.init = 305
10  monitor15_height.init = 228
11  monitor15.link = Wiki.jsp?page=ESAT_Monitor
12  %
13
14 %%rule
15  IF KNOWN[monitor15_width] AND KNOWN[monitor15_height]
16  THEN monitor15_a2 = eval(mult(monitor15_width,monitor15_height));
17  %
18
19 %%Solution monitor15 ~ Monitordiagonale 15 Zoll

```

This definition comprises more than a dozen lines, including multiple different keywords, to define one monitor. In principle five d3web valued objects are defined in the lines 2-5 and 19. To allow for a more concise definition of a monitor object a custom markup has been designed and implemented. The definition of the same knowledge object as above using the introduced custom markup reads like follows:

```

1 %%monitor
2   monitor15
3   @name: Monitordiagonale 15 Zoll
4   @link: Wiki.jsp?page=ESAT_Monitor
5   @width: 305
6   @height: 228
7   %

```

Here, the same knowledge can be expressed in a couple of lines reducing redundancy and the amount of keywords used. The five d3web objects (*monitor15_a2*, *monitor15_q*, *monitor15_width*, *monitor15_height*, *monitor15*) are created by the compiler according to the naming convention previously used within the standard markup shown above. The corresponding objects are not only created within the knowledge base but are also introduced to the compiler in a way that they are recognized when being used in other markup expressions. The area calculation also automatically is performed internally. In contrast to the multi-rule markup, this markup is strongly project specific, making it unlikely to be used in any other context.

Considering the markup aspects discussed in Section 4.4.2.2, this markup has a strong frame aspect defining the entire information about one complex object in one expression. Additionally, the markup has a very strong definitional aspect, as it in fact defines multiple variables, which then can be used in markup expressions of other parts of the document base. The calculation of the area is a typical case of an implicit knowledge aspect as the knowledge for the calculation, i.e. formula, is unaccessible from a document author's perspective.

7.1.5 Discussion

The knowledge acquisition activities were carried out during a time of more than two years. The meta-engineering was run in parallel basically emerging during occasional meetings between the main author and the system developer. In these workshops flaws and shortcomings of the current KAA have been analyzed and discussed. For possible improvements cost-benefit estimates have been worked out. With respect to the content level changes caused by modifications of the KAA, in most cases the manual restructuring of the contents has been found practicable, instead of writing refactoring scripts. This includes the introduction of the two markups discussed above. The multi-rule markup was introduced at a very early stage, where only a manageable number of rules of that kind had been inserted. One experience from the introduction of the monitor markup is that a cost-benefit estimation for a markup of this kind is quite difficult, as discussed in Section 4.4.2.1. While the total benefit is hard to be estimated, the implementation was decided as the workload was relatively low.

7.1.6 System Use

For the actual use of the knowledge system, the ESAT wiki is installed at the respective application site. The start the automated assessment process the system provides an interview component similar to the one discussed in Section 6.2.1. There the characteristics of the handicapped person are entered into the system by a care worker. An excerpt of this generated interview dialog is shown in Figure 7.4. Underneath the interview, the results of the consultation process are shown. Each derived solution device is provided with a link to the corresponding wiki page, where the device is described as shown in Figure 7.2. In that way, the care worker can catch up on the proposed device and if appropriate start the procurement process.

▼ ESAT-Fragebogen	
▼ Motorik	
▼ Obere_linke_Extremität	
Anzahl willkürlich beweglicher Finger der oberen linken Extremität?	<input type="text" value="0 - 5"/>
▼ Obere_rechte_Extremität	
Anzahl willkürlich beweglicher Finger der oberen rechten Extremität?	<input type="text" value="0 - 5"/>
▼ Oberes_Extremitäten-Paar	
Horizontale Überlagerung oberer Extremitäten?	<input type="text" value="-500 - 100"/> mm
Koordination von Simultanbewegungen der oberen Extremitäten?	eingeschränkt uneingeschränkt
▼ Kopf	
Kopf kann willkürlich bewegt werden?	Yes No
Stetigkeit des Kopfes?	stark beeinträchtigt gering beeinträchtigt nicht beeinträchtigt
Kopfmotorik?	beeinträchtigt nicht beeinträchtigt
▼ Augen	
Mindestens ein Auge kann willkürlich bewegt werden?	Yes No
Stetigkeit der Augen?	stark beeinträchtigt gering beeinträchtigt nicht beeinträchtigt
Feinmotorik der Augen oder Augenmotilität?	beeinträchtigt nicht beeinträchtigt
▼ Zunge	
Zunge kann willkürlich bewegt werden?	Yes No

Figure 7.4: The interview component for device assessment generated by ESAT.

7.2 WISSASS: Medical Knowledge about Cataract Surgery

The WISSASS project considers the development of an intelligent information system in the medical domain of cataract surgery. The research project is a cooperation of the Karlsruhe Institute of Technology, Germany (KIT) and the denkbares GmbH. It is funded as a ZIM-KOOP² project by the German Federal Ministry of Economics and Technology (BMWi).

7.2.1 Introduction

A cataract is a turbidity of the eye's lens which is appearing quite frequently especially affecting older people. In former days, a significant proportion of the older part of the population was suffering from reduced eyesight due to cataracts. Normally, a cataract can only be treated operatively. While operational treatments were known since the middle ages, these kind of surgeries were unreliable and very risky that time. The modern medical methods applied today however have very high success rates while being rather efficient in general. With about 20 million surgical intrusions per year, cataract surgery is the most widely applied type of operation applied on humans worldwide.

7.2.2 Application Scenario

There are still research efforts going on to improve the existing methods, optimizing success rates and cost effectiveness. Practical experiences have shown that about 90% of the cases can be regarded as ordinary cases, where a standardized treatment is applied, providing an extremely high success rate at relatively low costs. The remaining cases however, show a considerable high complexity, making the treatment process much more demanding. A suitable treatment has to be determined by choosing from a number of surgery methods, incorporating many boundary conditions. These cases, which are demanding even for experienced ophthalmologists, often benefit from new methods evolved recently in the field. The goal of the WISSASS project is to provide an intelligent information system, that assists the physicians, especially in the treatment of these difficult cases. Therefore, a knowledge system is designed that serves the following two use cases:

- **Second Opinion System:** A traditional knowledge-based system is employed routinely to run in parallel with the treatment process of each patient. The anamnesis and examination data of each patient is entered into the knowledge-based system, which checks whether there are deviations from the standard case that need to be considered. If so, the system provides the ophthalmologist hints about special issues that need to be considered for the treatment of the current patient. It further proposes a suitable surgery method if appropriate. The intelligent system also demands additional examination data for the patient, if it is required to exclude the risk of particular complications.
- **Tutoring System:** In an easily accessible intelligent information system, ophthalmologists can look up and study a comprehensive up-to-date body of knowledge about the

²<http://www.zim-bmwi.de/>

domain of cataract surgery. The tutoring system provides intelligent interactive navigation and is illustrated with multi-media content. It also shall provide means to further research about the hints or propositions made by the second opinion system as this only provides a very scarce explanation.

While the latter application scenario allows surgeons to look up particular aspects they are currently interested in, the second opinion system automatically runs in background providing hints on complicated cases.

When a basic version is established, the knowledge system needs to be adapted to the practical requirements of the respective clinic. Therefore, one challenge of the project is to provide a knowledge acquisition concept that allows for simple adaptation and maintenance. The clinic personnel should be enabled to perform minor adaptations on their own. The entire knowledge system is created in German language. Further reading about the project is provided in [RB13].

7.2.3 Knowledge Base Structure

The most important component of the knowledge base is formed by a semantic network. It comprises all important domain concepts of cataract surgery, as for example different kinds of examinations, surgery methods, or potential complications. The network contains about 380 concepts that are organized in a hierarchy. Additionally, they are interconnected by different types of relations, indicating special dependencies (e.g., causal dependencies) during the treatment of a patient.

The material for the tutoring system is organized around these concepts. For each concept illustrative content is presented on a distinct document. The edges of the semantic network provide semantic navigation between related topics of the domain.

A subset of the concepts are modeled as input attributes that allow to represent a patient's profile including the symptoms and examination findings. That profile provides the basis for the inferences of the second opinion system. The recommendations of the second opinion system are derived by a set of rules encoding expert knowledge about cataract treatment.

7.2.4 Knowledge Acquisition Process

The project is run by a small team including a couple of knowledge engineers and two ophthalmologists. While the one ophthalmologist is a young physician at the beginning of his career, the other one is an experienced expert in the field.

7.2.4.1 Seeding of an Initial Knowledge Base

The first knowledge acquisition activities have been conducted by indirect knowledge acquisition. In an initial step the domain concept hierarchies have been defined. For the subsequent step an analogue technique has been employed. The concept hierarchies have been plotted on a large poster. The poster allowed the ophthalmologist to inspect and verify the concept hierarchy in a comfortable visual way. Additionally, the expert defined cross-relations between concepts within the poster using colored pens. Figure 7.5 shows the poster with the drawn relations.

7 Case Studies

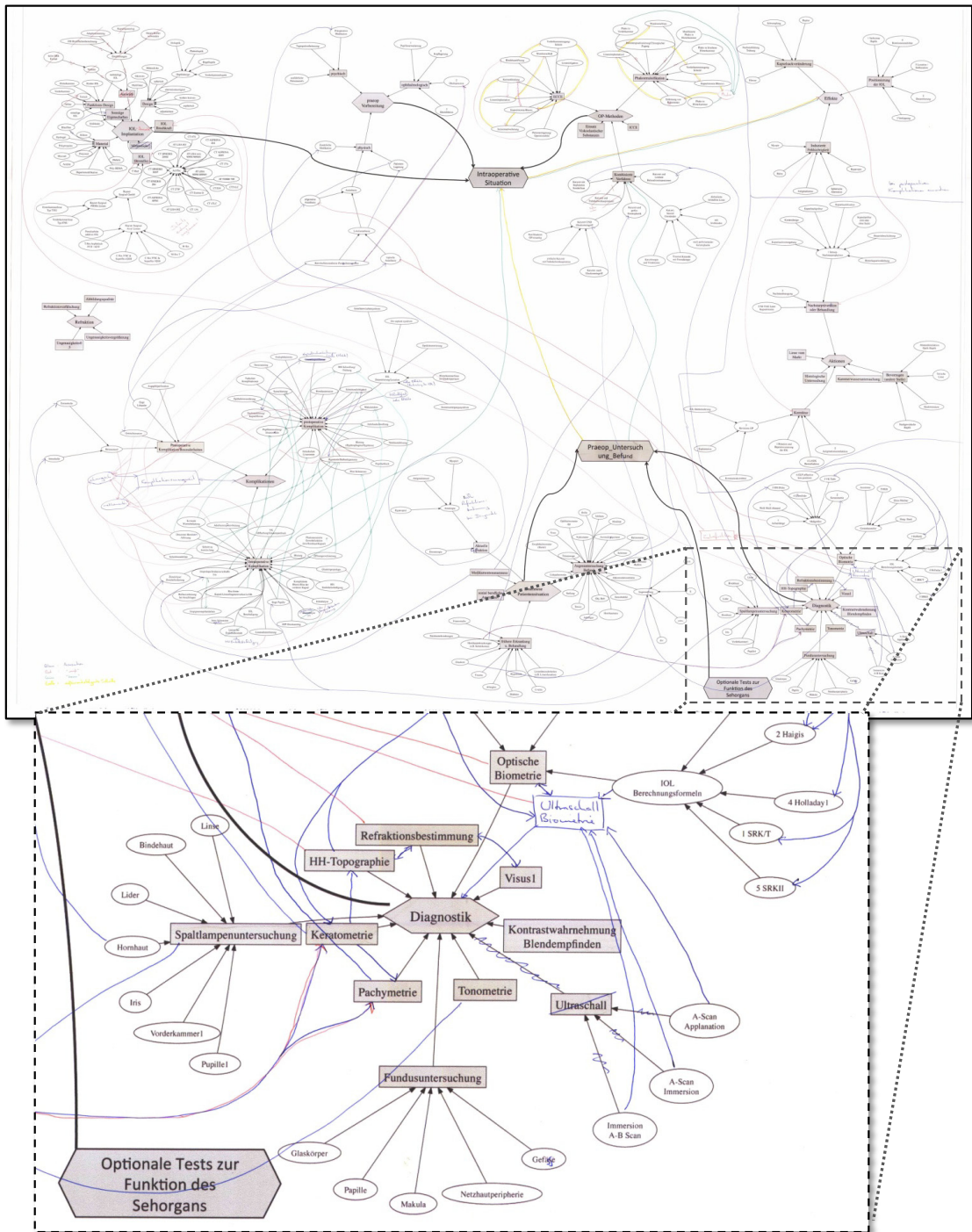


Figure 7.5: The semantic network of the WISSASS project printed on a paper poster.

As an initial version of the illustrative content a text book about cataract surgery has been digitalized. For each domain concept a document has been created, also defining the relations to other concepts. The related contents of the text book are attached to the respective concept documents. In that way, a document base of multimodal knowledge was created, using the system KnowWE introduced in Chapter 5.

7.2.4.2 The Meta-Engineering Process

In the meta-engineering process an adapted version of a KnowWE-based document-centered knowledge acquisition environment is evolved by developing numerous custom extensions. Beside special markups this includes components for navigation, search, visualization, and authoring support. These extensions are designed in close cooperation with the medical experts in joint sessions of discussion and assessment.


Knowledge Acquisition Architecture: In the WISSASS project, the designed knowledge acquisition architecture at the time of writing is as follows: There are two categories of content documents. The first category comprises documents that describe exactly one concept of the semantic network representing the respective aspect of domain knowledge. The second category represents further narrative content that can not be assigned to the description of a particular concept document. These pages should be linked (bidirectionally) to all concepts that are relevant to the topic described.

The structure, which any domain concept of the ontology is/should be described in, and that is illustrated by Figure 7.6, is as follows:

1. A custom concept definition markup defines a new concept of the ontology . (A)
2. The concept label is defined using the custom markup for concept labels. (*optional*) (B)
3. A list of the sub-concepts of the local concept defines the hierarchical structure of the ontology. Introduced by 'Unterkonzepte:', the comma-separated list markup specifies which concepts are sub-concepts of the local concept of this document. (*optional*) (C)
4. Then, further relations of the local concept within the semantic network can be defined. Therefore, the comma-separated list-based markup with the respective keyword are used. (*optional*) (D)
5. Finally, the informal description of the concept is defined using normal wiki syntax. (E)

For the elements 1 to 4 involving formal markups informal comments and/or illustrations are allowed and recommended.

Markup: The markup currently used to define and edit the semantic network is based on comma-separated lists. As illustrated in the document excerpt of Figure 7.6 each concept is described on a distinct page. Figure 7.7 shows the corresponding document source content with



Willkommen, *knowweadmin!* (nicht angemeldet) [Anmeldung](#) [Meine Einstellungen](#)

Untersuchung

[Edit Mode](#) [Bearbeiten](#) [Weitere...](#)

Inhaltsüberblick(Theorie)

Suche I

Benutzte Begriffe:

Augenuntersuchung Befund

- + Ergänzende Untersuchungen
- + Funktionsdiagnostik
 - Kontrastwahrnehmung_
 - Blendempfinden H
- + Orthoptische Untersuchungen
- + Refraktive Untersuchung
- + Spaltlampenuntersuchung
- Tensio
- + Kombinierte Verfahren**
- Kat.extr UND Glaukomeingriff
- Einfache Kat.extr. mit Trabekelwerkaspiration
- Kat.Glaukom OP einseitig
- Kat.extr. nach Glaukomeingriff
- Kat.extr und perfor. Keratoplastik

Bisher besucht:
Untersuchung, Allgemeine Ontologie G

Begriff: Augenuntersuchung Befund A Beg

Label: Augenuntersuchung Befund B Lab

Unterbegriffe: Funktionsdiagnostik, Orthoptische Untersuchungen, Tensio, Ergänzende Untersuchungen, Refraktive Untersuchung, Spaltlampenuntersuchung C Unt

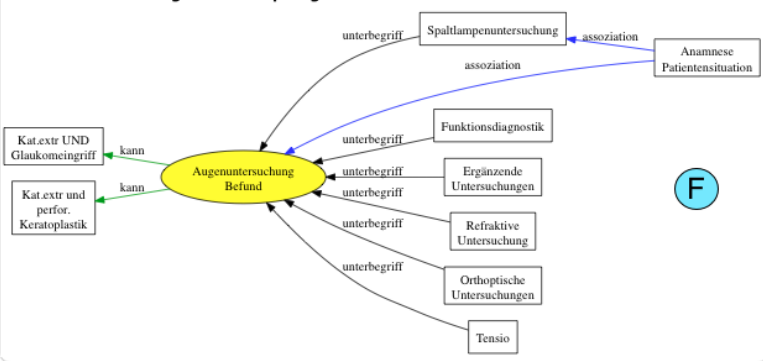
Beziehungen

kann: Kat.extr UND Glaukomeingriff, Kat.extr und perfor. Keratoplastik D Kan

Beschreibung

- Ggf. Quantifizierung von Kontrastwahrnehmung und Blendempfinden.
- Untersuchung der Augenanhangsgebilde (z. B. chronische Blepharitis, Dakryozystitis, Lidfehlstellung). E
- Spaltlampenuntersuchung: Man achte hierbei auf Konjunktivitis, Keratitis, Hornhautdegeneration oder dystrophie, Konfiguration der Vorderkammer, Iritis, Irisanomalien, wie z. B. Irisatrophie und Rubeosis, Typ und Ausmaß der Katarakt, Kapselzustand bei dilatierter Pupille, Linsensitz und Zonulaapparatstatus, maximal mögliche Erweiterung der Pupille.

Übersicht über Begriffsverknüpfungen



F

Diese Seite (Version-39) wurde zuletzt am 18-Jul-2013 13:27 von knowweadmin geändert. ▲

Figure 7.6: The document about the concept *Augenuntersuchung Befund* (in German language).

```

1 Begriff: Augenuntersuchung Befund
2
3 Label: Augenuntersuchung Befund
4
5 Unterbegriffe: Funktionsdiagnostik, Orthoptische Untersuchungen,
6 Tensio, Ergaenzende Untersuchungen, Refraktive Untersuchung,
7 Spaltlampenuntersuchung,
8
9 !! Beziehungen
10
11 kann: Kat.extr UND Glaukomeingriff, Kat.extr und perfor. Keratoplastik
12
13 !! Beschreibung
14
15 * Ggf. Quantifizierung von Kontrastwahrnehmung und Blendungsempfinden.
16
17 * Untersuchung der Augenanhangsgebilde (z. B. chronische Blepharitis,
18   Dakryozystitis, Lidfehlstellung).

```

Figure 7.7: The raw text view of the document about the concept *Augenuntersuchung Befund*.

the list-based markup expressions. In line 1 the concept *Augenuntersuchung Befund* is defined using the keyword 'Begriff:', forming a definitional markup (c.f. Section 4.4.2.2). In the lines 5 to 7 the sub-concepts for that concept are listed. For the predefined relation types of the semantic network corresponding keywords have been designed, such as 'Kann:' and 'Muss:', as shown in the markup example in line 11. There two *kann*-relations of the local concept *Augenuntersuchung Befund* to the concepts *Kat. extr UND Glaukomeingriff* and *Kat. extr und perfor. Keroplastik* respectively are defined. This list-based markup is a good example for a purely relational markup.

The markup expressions from the lines 1, 3, 5 to 7, and 11 are independent, self-contained expressions, considering segmentation and compilation, i.e., there can be other document content in-between. This provides high flexibility as the comma-separated lists can be written anywhere within the document, while always referring to the concept defined on the local page. If none or multiple concepts are defined within the document, an error is shown to the user accordingly. Strictly speaking this markup design violates one of the principle of document-centered knowledge acquisition, which is that reorganization/reordering of the content elements does not change the (semantics of the) knowledge base. In other words, moving any markup expression to another place in any document will result in the identical compiled knowledge base. Moving one of these comma-separated lists to another document will violate this assumption. However while happening rarely anyway, moving a list to another concept resulting in the list referring to the respective local concept still is quite intuitive for the user. Hence, during the design of the markup, we decided to accept this minor flaw in favor of simplicity and conciseness.

Authoring Support: In Figure 7.6 the list-based markup for defining ontology relations is shown (D). The concept browser in the left menu (H) can be used for drag-and-drop editing of the markup lists. Any concept term can be dragged into the main panel of the page and dropped onto a relation list. The concept will automatically be appended to the list, also including the list element separator. The elements of the lists can also be deleted in a very simple way. When the mouse cursor moves over a concept in the list, a little delete icon with an 'x' appears next to the concept name. A click will remove the respective concept from the list (including separator if required). In that way, the lists can be edited in a quick and simple way without the need of typing concept names (assuming they can be found in the concept browser).

The actual creation of new lists still requires typing (or copy-pasting), while not being too demanding. The corresponding keyword for the relation type, e.g., 'kann:' or 'muss:', has to be typed in a new line somewhere within the corresponding document page.

Misspellings within the lists cannot occur using the list editing mechanism described above. Still, as common in document-centered authoring in principle, any content of the documents can also be edited manually using the standard text editing interface. In that way, misspellings can be included within the lists. To prevent problems, the concept identifiers are checked against the set of defined concepts. If no concept with the respective name is found, an error message is displayed and correction recommendations for the concept names with the smallest edit distance are proposed. The correct concept can be chosen from the list and the erroneous element is replaced accordingly by one click.

Navigation & Search There are multiple elements included in the interface shown in Figure 7.6 that help to support the navigation task. For each page that represents a concept of the semantic network, the parent concept is shown at the top of the page (A). The link will lead directly to the page of the parent concept. Further, underneath the document content, an excerpt of the semantic network is shown in a graph-based visualization (F). It contains the concept of the local page at its center and all of its neighbors in the network. All the nodes also serve as direct links to the corresponding page. At the bottom of the left menu (G), the trail is displayed, showing all the pages, that the user has visited recently. The concept browser (H) also plays an important role for the navigation within the document space. Collecting all domain concepts that have been used recently, it provides a comprehensive overview of the related aspects of the domain in a hierarchical style. The search slot (I) combines semantic search within the network and full text search within the document content. The execution of a search leads to a search result page, that displays the network concepts related to the query as well as the full text search results.

Refactoring Due to the design decisions within the meta-engineering process multiple refactoring activities have become necessary. Several of these refactoring tasks are explained in the following. Minor or trivial refactorings, such as the renaming of concepts for example, are omitted in this consideration.

- **Relations to List Markup:** At the beginning of the project the relations between concepts have been defined by a general predefined markup provided by the system. With that

markup, the knowledge defined in line 11 of the above markup example would read as follows:

```
1 > Augenuntersuchung Befund Kann:: Kat. extr UND Glaukomeingriff
2 > Augenuntersuchung Befund Kann:: Kat. extr und perf. Keroplastik
```

The markup expressions of this kind had to be transformed to the introduced list markup. As the relevant relation expressions already were located on the document representing the subject concept (Augenuntersuchung Befund), this refactoring operation can be considered a local refactoring, only depending on the content of the local page. This task had been performed by a special refactoring script that operated on the KnowWE KDOM structure and that was iteratively applied on all documents.

- **Hierarchy Decentralization:** Initially the concept hierarchy has been defined as one large structure on a separate document using a dash-tree markup (c.f. Section 3.5.2.1). With several hundred concepts the hierarchy appeared hard to read and edit. Further, it was completely separated of all the other knowledge existing about the concepts. Therefore, the decision was made to maintain the concept hierarchy in a distributed way. For each concept the list of child concepts simply is enumerated as a comma-separated list on the document describing that concept. In that way, the hierarchy knowledge for a concept is conveniently interwoven with the rest of the knowledge for the respective concept. The corresponding refactoring task also has been implemented by a custom refactoring script traversing the KDOM structure of the centralized hierarchy and creating the respective list expressions on each concept description document. The centralized dash-tree hierarchy has then been deleted.
- **Concept Description Order:** Initially, the structure of the pages describing a concept showed the informal description of the concept on top, followed by the formal knowledge expressions, such as relation lists for instance. This content order proved to be inconvenient when larger descriptions, also including images, were created. As a consequence, in many cases it became necessary to scroll down in the document view to see the formal knowledge expressions. As it was important for the users to have them in sight quickly, a change of the content order was decided: The descriptions have been placed at the bottom by employing a scripted refactoring operation. In that way, the content ordering discussed above has been obtained.

7.2.4.3 Outlook

Above, the current state of the knowledge acquisition architecture and the custom knowledge acquisition tool was described. Up to now experts have been actively involved within the meta-design process. The basic parts of the initial knowledge mostly have been formalized by indirect knowledge acquisition. In the current project phase, a flexible distributed workflow using an online version of the customized authoring environment is conducted. The ophthalmologist reviews the knowledge autonomously and notifies the knowledge engineer by mail when a deficiency was detected. The knowledge engineer then corrects the knowledge base according to the

7 Case Studies

expert's demands. This workflow resembles the strategy of mixed-initiative knowledge acquisition. Within the next project phase we expect the expert to also edit knowledge directly using the customized tool. Based on those experiences, further optimizations of the tool and the KAA will be made, carrying on the meta-engineering process.

7.2.5 System Use

For the actual use of the system, it will be installed at the surgical center. As discussed in Section 7.2.2 the use of the knowledge is twofold. For the tutoring component the system provides access via a web application providing a simplified version of the interface shown in Figure 7.6. It can be accessed by young physicians for manual research in case of need any time.

For the use of the automated decision-support system, the knowledge base needs to be integrated into the workflow of the clinical information system. That is, when the patient data are entered into that clinical information system it is also pushed to the wisass installation. There it is entered into the knowledge base generating the treatment hints which are returned to the clinical information system. When the surgery preparation sheet, which is always read by the surgeon before the surgery, is printed, those hints are added in a special section. In that way, ophthalmologists can benefit from the knowledge base without additional need of actively interacting with a computer system.

7.3 Managing Chemical Safety with KnowSEC

: In the following, the project *KnowSEC* is briefly introduced. It is a knowledge management project at the German Federal Environmental Agency (Umweltbundesamt) supported by the denkbares GmbH. For the management of knowledge about chemical substances an adapted document-centered knowledge management process has been established by the use of a customized version of the system KnowWE.

7.3.1 Introduction

The German Federal Environmental Agency has the difficult task to evaluate the dangerousness of chemical substances newly invented by industry. Due to the large number of novel substances, ranging in thousands every years, it is not possible to perform extensive laboratory examinations for each one. A pre-selection of potentially dangerous substances has to be made based on existing or simple obtainable knowledge. This knowledge needs to be aggregated and reviewed by different domain experts to rate the dangerousness of a substance according to different categories (e.g., toxicity, water persistence). On this basis, a priority list for critical substances, that urgently need to be extensively examined in laboratory test series, is established.

7.3.2 Application Scenario

The decision process sketched above involves many staff members from different departments. The KnowSEC system has been designed to support the complex task of knowledge aggregation and decision making of the distributed team. As a basis for decisions different categories of knowledge are employed. Parts of the domain, that are suitable for formalization, are modeled to executable knowledge bases to perform particular sub-decisions. Other sub-decisions are performed by experts manually by establishing the decision as a *decision memo*, including a verbose justification, in the KnowSEC system. That system does not only capture knowledge about substances but also focuses on process documentation. For this purpose, the system also supports verbose documentation of the distinct steps in the episodic and collaborative decision making process (c.f. [BSBN13]). Figure 7.8 shows the page about the (fictional) substance *kryptonite* summarizing the information about the decision making process about this substance so far.

This application scenario differs from classical knowledge engineering projects, for instance creating a diagnostic knowledge base. The decision process is not fully automated but requires collaboration of a team of experts. Therefore, in addition to the knowledge acquisition task the same system also requires to support the second major task of decision making. This poses significantly higher challenges for the system customization driven by the meta-engineering process.

7.3.3 Knowledge Base Structure

The executable part of the knowledge base consists of 214 questions (user inputs to characterize the investigated substance) grouped by 46 questionnaires, 146 solutions (assessments of the

The screenshot shows the user interface of the KnowSEC system. At the top left is the logo for 'Umwelt Bundes Amt KnowWE'. The main title is 'Kryptonite'. The user is logged in as 'G'day, Joachim Baumeister (not logged in)'. A trail of navigation is shown: 'Your trail: PAK-Team, Teams, Substances worked on by UBA-Teams, Substances for which a RMO-Analysis is or was compiled by UBA, Substance Lists, SVHC-dossier work, Pentacosfluorotri...'. There are navigation buttons for 'View', 'Attach (1)', 'Info', 'Edit Mode', 'Edit', and 'More...'. A search bar is labeled 'Quick Navigation'. On the left, there is a 'Navigation' menu with links like Home, Teams, Substance Lists, etc., and a 'Decision Making' section with a tree view. The main content area is divided into sections: 'Decisions (5)' with a table of decision points and their status, 'Identifier' with a table of chemical identifiers, and 'Memos (2)' with a list of recent updates. A 'Need for more info on tox.' section contains a paragraph about Kryptonite from Wikipedia. An 'About the substance' section contains a paragraph about its fictional origin. An image of Superman is shown on the right. At the bottom right, it says 'This page (revision-2) was last changed on 12-Sep-2013 10:15 by Joachim Baumeister ▲'.

Figure 7.8: The user interface of the KnowSEC system showing a document about the (exemplary) substance kryptonite.

investigated substance), and scoring rules to derive the assessments. At the time of writing the system stores information about more than 11000 substances.

7.3.4 The Meta-Engineering Process

At beginning of the project, knowledge was aggregated using normal (office) documents. While support knowledge was defined in Word documents and mind maps, decision tables have been defined as Excel sheets. That project stage can be considered as the exploration phase of the meta-engineering process (c.f. 4.4). As a reasonable amount of knowledge had been established in well-structured (office) documents, the content was transferred into a document-centered knowledge acquisition environment to support convenient collaborative editing and automated compilation of an executable knowledge base. For this seeding task each office document was converted into an equivalent wiki document. Since the seeding activity, the knowledge was exclusively edited online within the KnowWE-based authoring environment. After the conversion, the development of a compiler for the decision-tables, actually creating executable d3web rules, was one of the first tasks of the implementation activities in the meta-engineering process. Figure 7.9 shows a wiki page with a table of that kind. In the column-header solution predicates about the substance of interest are defined. Question-answer pairs, that are relevant for these solution predicates, are listed in the first row of the table. Each cell entry defines a scoring rule, which is adding the corresponding score to the score account of the solution predicate if the

question-answer pair is fulfilled considering the input data. Up to now, there are more than 1000

The screenshot shows a web browser window with the URL `denkbare.dyndns.org/KnowSEC-TestServer/Wiki.jsp?page=TM%20Relevance%20Status%20internationally`. The page title is "TM Relevance Status internationally". The user is logged in as "G'day, Albrecht Striffler (authenticated)". The page contains a "Scores" table with the following data:

	Substance not regulated internationally	Substance sufficiently regulated internationally	Substance may not be sufficiently regulated internationally	A country outside Europe regulates the substance for environmental reasons
Is this substance internationally regulated? = Yes	N7	P5	P1	-
Is this substance internationally regulated? = No	P1	N1	P4	-
Is this substance on any national environmental regulatory action or procedure outside Europe? = Yes	P3	-	P5	P7
Is this substance on any national environmental regulatory action or procedure outside Europe? = No	N1	-	N2	N7
Listed in the Stockholm POP Convention? = Yes	N7	P7	N7	-
Listed in the Stockholm POP Convention? = No	P1	N1	P3	-
UNKNOWN[Listed in the Stockholm POP Convention?]	P2	-	P4	-
Listed by IPCC as greenhouse-effective substances? = Yes	N7	P7	N7	-
Listed by IPCC as greenhouse-effective substances? = No	P1	N1	P3	-
UNKNOWN[Listed by IPCC as greenhouse-effective substances?]	P2	-	P4	-
Listed by Montreal Protocol as ozone-depleting substances? = Yes	N7	P4	N3	-
Listed by Montreal Protocol as ozone-depleting substances? = No	P1	N1	P3	-
UNKNOWN[Listed by Montreal Protocol as ozone-depleting substances?]	P2	-	P4	-

Figure 7.9: A score table generating decision rules for the automated rating of substances.

rules defined in KnowSEC using that table representation. Beside that markup for tables, also a custom markup for the representation of decision memos has been designed. During the knowledge engineering activities the demand for specialized authoring components for the table-based representation of rules and also for the editing of memos emerged. In consequence, form-based editors have been introduced as an alternative to the textual editing interface. The custom extensions have been developed during a time period of more than 2 years. They have been designed during the meetings of the project leaders and experts from Federal Environmental Agency and

a knowledge engineer and a system developer from the denkbares GmbH, which took place on a regular basis.

7.3.5 System Use

In Section 4.4 we described the meta-engineering process as a second process running in parallel to the actual knowledge acquisition process. Within the application scenario of KnowSEC in addition to these two processes the decision making can be considered as a third process. This third process starts some time after the knowledge acquisition process, as a certain amount of knowledge is required to support the decision making. For the decision making the experts create memos within the system which represent and document a decision. Beside the decision fact itself, e.g., whether the substance is bio-accumulating or not, it also contains the following information: Who made this decision; when this decision was made; on what information the decision is grounded.

During the practical use of the KnowSEC system, knowledge acquisition and decision making runs in parallel as new knowledge (possibly about newly invented substances) needs to be integrated continuously. After about 2 years of knowledge acquisition and meta-design, recently the decision making on the system has begun as a pilot phase. To make previous work available within the system, existing information on substance assessment has been imported into the system. With this data decision memos have been generated but also the formalized knowledge bases have been executed by filling the input questions from existing data bases. Together, the system currently contains more than 42,000 module decisions. More information about the KnowSEC project are presented by Baumeister et al. [BSBN13].

7.4 Maintenance Knowledge for Special Purpose Machines

7.4.1 Introduction

Today's modern agricultural vehicles are machines of very high technical complexity. Despite of a high quality manufacturing process, technical defects can occur due to the intensive use of the harvesting machines in practice. A breakdown is very costly and disturbing, especially during the harvest time. Therefore, providing timely and effective maintenance service and support is an important issue. The precise capturing and reporting of the problem symptoms forms the first step within the maintenance chain. For manufacturers operating internationally, this is even more challenging as the communication between the machine operator or technician on site and the service center at the manufacturer's facility often has to cross a language barrier.

7.4.2 Application Scenario

In this section, we report about a knowledge engineering project, carried out by a world-wide operating manufacturer of harvesting machines, developing a structured representation of the symptoms as a formal ontology. Operational problems reported from machine owners, often located in foreign countries, in natural language are in many cases hard to comprehend by the experts at the service center. The aim of the described project is to allow for the precise and unambiguous description of observed symptoms for all products in a language independent and uniform manner. Therefore, an ontology is defined that provides the required terms for symptoms, functions and parts in a structured and language independent way. Additionally, the ontology connects symptoms with repair actions, helping to give hints for solutions instantly.

In the end, the ontology enables to provide the customers and technicians a tool which supports the creation of unambiguous failure reports, which can then automatically be transformed to different languages.

7.4.3 Knowledge Base Structure

The content of the developed ontology can be summarized as follows: The model of a machine basically is described by a component hierarchy, a function hierarchy, and a catalogue of parts. Furthermore, the ontology defines the two different categories of symptoms: the *malfunctions* and the *damages*. A malfunction is a symptom that is associated to a function, which is observed to be not working properly under certain conditions. There, the faulty component potentially is unknown. For damages on the other hand the problem can be clearly localized to a faulty component of the machine. The ontology also comprises a catalogue of repair actions. A symptom can be associated to one or more repair actions, that are likely to solve the problem.

7.4.4 Knowledge Acquisition

For the development of the ontology the document-centered knowledge acquisition approach has been chosen. As authoring environment the system KnowWE, introduced in Chapter 6, is employed. The major part of the knowledge has been imported from information sources, that already had been created by the domain experts. Many markup expressions could be generated

7 Case Studies

automatically from knowledge already existing in structured formats. Currently, the knowledge base is consisting of 20 wiki pages, containing 94 malfunctions, 104 damages, and 33 repair actions with labels for three languages for each knowledge base object. Figure ?? shows the entry page of the wiki containing the abstract classes *Damage* and *Malfunction*.

Initially, a standard markup, that is provided by the system, was used to define the ontology concepts and their interrelations. The following markup expression defines the symptom *ah045* and its relations. In line 1 the concept itself is defined as an instance of the class *Damage* (symptom). Subsequently, a set of relations for this symptom is stated in the lines 2 to 8, including a label and a comment. Furthermore, the damage is connected with a failure code in line 6 and a component identifier in line 7.

```
1 %%Individual ah045 @type: Damage
2 %%Relation
3 ah045 rdfs:label 'Verschmutzung des Filters in der Harnstoffpumpe.'@de
4 ah045 rdfs:comment 'ah045'
5 ah045 hasLocation B_1165320
6 ah045 hasFailureCode T0003
7 ah045 hasFailurePart P_0019403910
8 %
```

For all of the damage symptoms, similar expressions are defined within the wiki pages. In this example only a German label for the concept is defined. Additional labels for other languages can easily be added to support the automated translation later-on. Applying the meta-engineering approach as proposed in Chapter 4, a new markup has been defined to simplify reading and editing of knowledge. Using the new custom markup, the knowledge describing the damage *ah045* reads as follows:

```
1 %%Damage
2 ah045: 'Verschmutzung des Filters in der Harnstoffpumpe.'@de
3 @location: B_1165320
4 @part:      P_0019403910
5 @code:      T0003
6 %
```

This domain specific markup combines the definitional aspect and the relational aspect, i.e. it defines a new concept and its relations in one coherent markup expression. In that way, the knowledge is expressed more concise, improving readability. The comment, which is missing in this new markup, is generated automatically in background, containing the same term as the concept identifier, which is intended in the project.

The introduction of the new markup leads to a significant improvement of the content comprehensibility. For instance, the total amount of markup expression lines was reduced by more than 200 lines.

7.4.5 System Use

The developed ontology is used to create a web-based service support system which is available world wide to machine operators. When a system fault is observed the operator instantly can access the web application to report the problem. For this purpose the page provides a single input field which provides semantic autocompletion using the operators native language. The intelligent search slot assists the user to select a system component and a damage for instance. During the selection process, the operator can see short descriptions of the respective concepts in his language. When this encoded problem report is submitted, the professional service personnel can translate the problem report automatically and unambiguously into a language of choice. In that way, effective service support actions can be taken instantly.

7.5 HermesWiki: E-Learning in Ancient Greek History

7.5.1 Introduction

The *HermesWiki* [RLB⁺10] is an e-Learning platform in the domain of Ancient Greek History. It is developed in cooperation with the Department of Ancient History of the University of Würzburg. The aim of the project is to support students in exam preparation by providing a web-based learning platform with quality assured, relevant material. The entire project is created in German language. Basically, there are four different types of content entities contained in the system:

1. Medium sized essays, each describing an important topic of the domain as plain text.
2. Important events with date information, a brief plain text description and (historical) source references.
3. Descriptions of important domain concepts (e.g., persons, cities, islands).
4. Historical sources (German translations).

The goal of the knowledge engineering process was the enhancement from a standard content management system (allowing for reading, browsing and plain text search) to a semantically enriched platform providing augmented visualization of the content, interactive features, and semantic navigation and search methods, based on a formalized model of the knowledge.

7.5.2 Knowledge Acquisition Architecture

At first, we introduce the current KAA of the HermesWiki platform before the different phases of the project's meta-engineering process are discussed:

- **Support Knowledge:** The HermesWiki gives an overview of all the important concepts of the domain, such as persons, cities and peoples. Each so called glossary concept is briefly described on a distinct page. However, the most important content parts are the essays, each covering some important aspect of ancient history by a coherent description.
- **Arrangement of Formal Knowledge:** The HermesWiki ontology is entirely defined within the document base. General terms and relations of classes and properties, e.g., the class hierarchies, are defined on a few centralized pages containing the vocabulary definitions. Instances and their interrelations however, are widely distributed over the wiki, being strongly interwoven with the support knowledge according to the domain context. Considering the glossary concepts, the general attributes, such as birth and death dates of persons or coordinates for locations, are defined on the corresponding wiki page. The time events, forming the most important entities of the formal knowledge base, technically can be defined on distinct documents or inline anywhere within the source text of some document (e.g., essay). While it is reasonable to have own pages for very important events, we also perceived the inline definition in context as practical for further events. However, an event defined inline can be easily extracted to a new page by a refactoring operation later.

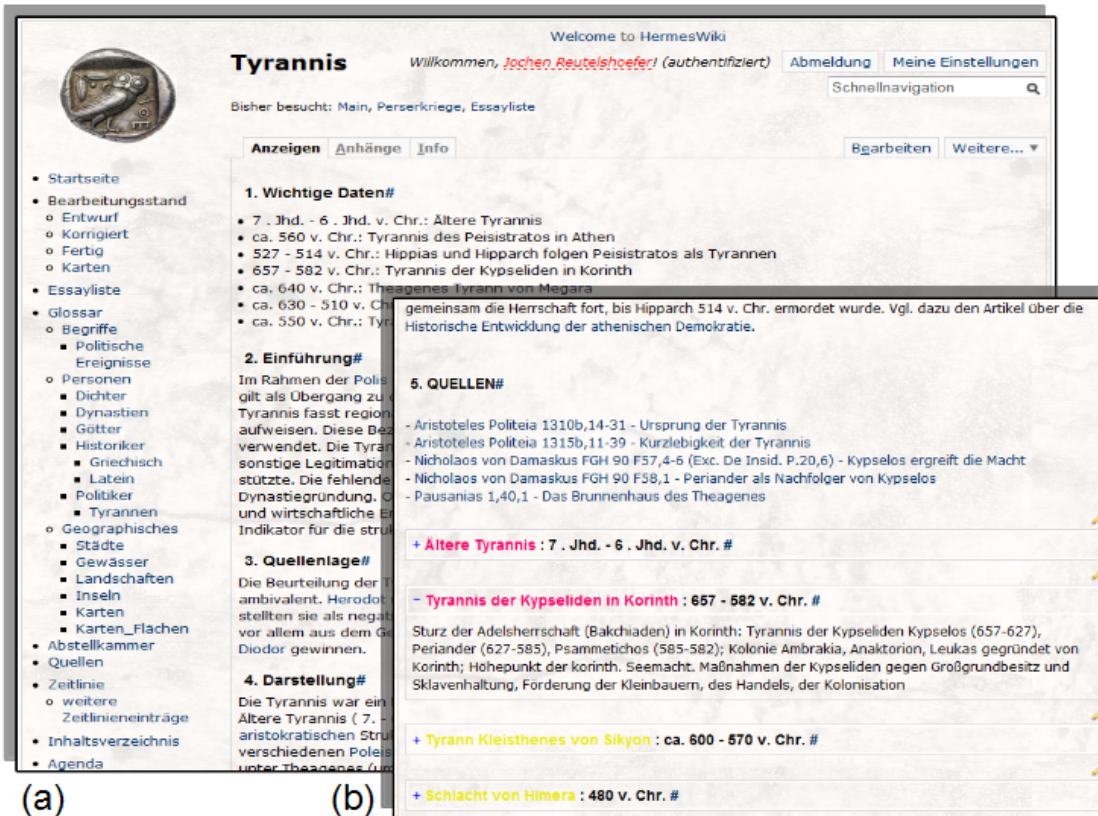


Figure 7.10: Screenshots from the HermesWiki system.

- Syntactical Structure:** The syntactical shape of the class hierarchy discussed above is a dash-tree (c.f. Section 3.5.2.1), that proved to be practical for concise representation and quick editing abilities. Any term being a dash-tree child, i.e., follows with an incremented number of dashes, is defined as a subclass of its parent. Another important (customized) formalization aspect of the HermesWiki KAA is the markup for the inline definition of time events. Figure 7.11 shows a markup example for the time event *Lamian War*. The markup is translated and added to the ontology repository. As first information entity the title of the event (*Lamian War*) is given, followed by the importance rating defining its relevance for student exams. In the next line, the time-stamp of the event is notated, also including annotations for different degrees of uncertainty. Then, introduced by “=>” an optional class membership definition can be added. Further, the body of the markup follows, consisting of a (free-text) description of the actual event. The markup concludes with an (optional) list of historical sources where the event is mentioned, explicitly marked by the keyword “SOURCE:” as the first word of a new line (from *Diodor* and *Pausanias* in this example). Considering the markup aspects discussed in Section 4.4.2.2, this markup has a strong frame aspect defining the entire information about one complex in one expression. With respect to the slot identification, it is a mixture of order convention (title,

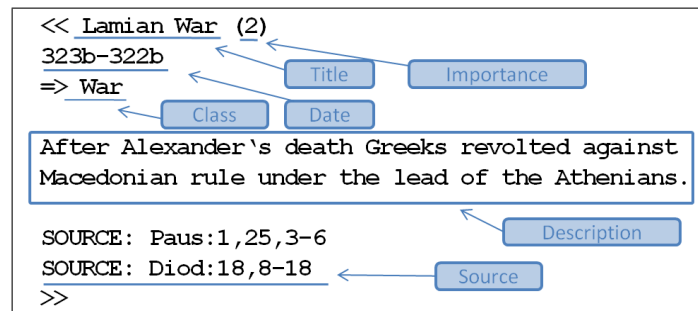


Figure 7.11: The markup to formalize time events 'inline' in HermesWiki

importance, date) and explicit slot naming (assignment of class and sources). Any content in between is considered as the description of the event, which also follows the convention style.

7.5.3 The Meta-Engineering Process

In the following, we sketch the progress of the phases of the meta-engineering process:

- **Exploration:** At the beginning of the project different ways of structuring the content entities (essays, events, concept-descriptions, sources) were discussed. It became obvious, that the domain concepts (e.g., cities) should be described independently of the essays referring to them. Describing the concepts in a very general way on distinct articles allows references from different contexts/essays. As opposed to this, specific events should not necessarily require an own page, but flexible definition 'inline' within a document (i.e., essay) proved to be appropriate.
- **Design:** In the design phase, at first a simple markup for defining time events, similar to the one shown in Figure 7.11, was specified in a workshop together with the historians. The ability to specify a class membership was not contained in the first version of the markup, used for months. As these class memberships then showed to be necessary and otherwise needed to be inserted using the general `rdf-turtle-syntax`³ separately, the time event markup was extended accordingly.
For efficient navigation and search a taxonomy of domain concepts was required. To allow for simple and quick modification of the class hierarchy for the domain experts, we decided for a dash-tree markup for defining classes and subclass relations.
- **Implementation:** In general, the web application is frequently updated when features are added or improved. The time event markup was implemented in multiple stages. First, the events only have been recognized and highlighted in the page view. In the next step its translation to the RDF-store was carried out to make the knowledge available for automated processing. Later, the extension for defining the class membership has been introduced.

³<http://www.w3.org/TeamSubmission/turtle/>

- **Knowledge Acquisition:** One advantage of the meta-engineering process is the possibility to start knowledge acquisition at an intermediate stage when the development of the tool is not yet finished. Large parts of the content (i.e., essays, concept descriptions) could be evolved independently of the current implementation state. While the design of the KAA is developed in close cooperation of knowledge engineers and domain experts, large parts of the knowledge acquisition and formalization in this project is in general performed by the domain experts autonomously. However, axioms and entities describing the terminology of the ontology are developed in close cooperation.

7.5.4 System Use

It is not a single major use case that the created domain ontology is employed for. The possibilities how the knowledge can serve the e-Learning scenario are manifold. For instance, the ontology can be used to improve search and navigation within the content. Two interesting use cases, valuable for learning activities, are introduced in the following. More details about the HermesWiki ontology use cases can be found in [RLB⁺10].

7.5.4.1 Generated Geographic CV

The ontology contains lots of information about events and persons, also including information about what person has been involved in some event. Further, the events are provided with date information and a geographic location. All this information can be put together to generate a life story of a particular person. Therefore, the important events, that the person was involved with, are selected. For these events the geographic locations can be obtained from the ontology. With this information a map can be generated showing the different events by markers, providing more information on demand. Figure 7.12 shows the life story of Alexander the Great. This map, automatically generated by the ontology content, aggregates lots of related domain information, making it easily accessible to students.

7.5.4.2 Automated Quiz Sessions

Another use case to support learning activities are generated quiz sessions. Each fact in the ontology can be used to generate a question. For one part of the fact (person, location, date) multiple distractors, i.e., wrong but quite plausible answer options, need to be generated. In this way, a multiple choice quiz can be created, which is able to provide instant feedback to the learner. Figure 7.12 shows the HermesWiki quiz asking dates for time events.

7 Case Studies

Alexander Willkommen! (unbekannter Gast) Anmeldung Meine Einstellungen

Schnellnavigation

Bisher besucht: Inhaltsverzeichnis, Agenda, Sandbox, Problem, Schablone, TextFormattingRules, Recent Changes, Verwaltung, Alexander, Alexanderzug

Anzeigen Anhänge Info Bearbeiten Weitere...

Alexander III. (auch Alexander der Große) lebte 356-323 v. Chr. und errichtete durch mehrere Feldzüge seit 336 ein riesiges Reich Alexanderzug und Alexanderreich.

Politiker

333 v. Chr - Schlacht bei Issos mehr...

+ Ereignisse für "Alexander":

- Startseite
- Bearbeitungsstand
 - Entwurf
 - Korrigiert
 - Fertig
 - Karten
- Essayliste
- Glossar
 - Begriffe
 - Politische Ereignisse
 - Personen
 - Dichter
 - Dynastien
 - Götter
 - Historiker
 - Griechisch
 - Latein
 - Politiker
 - Tyrannen
 - Geographisches
 - Städte
 - Gewässer
 - Landschaften
 - Inseln
 - Karten
 - Karten_Flächen
- Abstellkammer
- Quellen
- Zeitleine

Figure 7.12: The "life story" of Alexander the Great on a map, generated from the ontology data.

HermesQuiz

End of Spartan Naval Supremacy

Autumn 394 B.C. ✓

206 B.C.

Summer 393 B.C.

Begin of the Persian Empire ?

395 B.C.

ca. 248 B.C.

ca. 552 B.C.

Stop quiz 3 / 4 = **75%**

Figure 7.13: A quiz, generated from the time event data in the ontology, is asking dates of historical events. The students get instant feedback about his choices and scores.

8 Conclusion

This chapter provides a conclusion of the presented work. At first, a summary of the distinct chapters is given. Then, an outlook of interesting research questions considering this approach is presented. The work closes with a discussion about the role of this work within the landscape of knowledge acquisition research.

8.1 Summary

8.1.1 Introduction

As an introduction to the topic of this work, a short summary of the past 30 years of knowledge acquisition research was given. Beside a chronological overview, the most pressing research questions of the last decades, and the progresses made with respect to those, have been discussed.

Then, the general problem of knowledge engineering, the so-called 'knowledge acquisition bottleneck' was introduced, describing the problematic nature of formalizing domain knowledge for building intelligent systems. In manual knowledge acquisition the problem is strongly related to the 'competency dilemma', describing the unbalanced distribution of competency between domain specialists and knowledge engineers, considering domain expertise and knowledge engineering expertise. There are two major strategies of (manual) knowledge acquisition, the direct and the indirect knowledge acquisition, each having its strengths and shortcomings. The 'mixed-initiative knowledge acquisition', based on active participation, is a flexible combination of both strategies. To create the required socio-technical conditions for effective mixed-initiative knowledge acquisition, we propose to adequately support the social aspect that is inherent to knowledge engineering. We propose to establish a social process that allows for the exchange of expertise along the two dimensions domain knowledge and knowledge engineering skills on the one hand. On the other hand the process shall lead to the specification of a project-specific custom-tailored knowledge acquisition tool environment. To support the social process, a suitable cognitive environment for learning is required, providing domain descriptions and allowing for participation at low technical barriers.

Further, the knowledge level perspective for intelligent systems was introduced, which is located on top of the symbol level at the computer systems stack. It provides an additional layer of abstraction allowing for the discussion of knowledge engineering issues independently of the symbol level. In that way, the knowledge acquisition task can be considered as an act of communication between a human agent and a knowledge-based system agent in a particular environment, i.e. the employed knowledge acquisition environment. This communication is based on the possibilities of access to the knowledge in the (knowledge acquisition) environment. Allowing effective knowledge transfer in a knowledge acquisition environment therefore requires designing suitable means of access—for human and computer agents. The use of documents

8 Conclusion

for knowledge acquisition provides a wide range of possibilities for knowledge transfer as well as for supporting the social process of knowledge acquisition. The contribution of this work is a comprehensive discussion about taking advantage of these possibilities for enabling effective knowledge engineering.

8.1.2 Approaches for Knowledge Base Authoring

In Chapter 2 the different existing knowledge formalization approaches are discussed. The most important family of user interfaces is the graphical user interfaces, also being predominant in knowledge acquisition until today. There the major categories are tools with form-based interfaces, graphical representation languages, and tabular knowledge acquisition. Also, domain specific languages and electronic documents are possible means for knowledge acquisition. Those two elements play a fundamental role for the knowledge engineering approach presented in this work. One important aspect of distinction between graphical user interfaces and the use of document is the persistence structure of the content. Graphical user interfaces generate the content view "on-the-fly" from the formal knowledge base content, entirely determining the presentation of the content elements, e.g. their order or positions. In document-centered knowledge acquisition these structural characteristics of the content elements are defined by the user by creating the document structure in a reasonable and memorable way.

8.1.3 Document-Centered Knowledge Acquisition

The document-centered knowledge acquisition approach proposes the use of a particular user interaction paradigm—the authoring of (electronic) documents with knowledge formalization by the use of (knowledge) markup languages. Knowledge documents containing markup expressions are forming 'multimodal knowledge' containing content elements of the following categories: 'Domain Description', 'Modeling Rationale', 'Markup Expressions', and 'Organizational Information'. Documents containing these kinds of content elements are forming the 'Document Space', which is a graph of content elements connected by relations within the documents, e.g. successor, inclusion, or links. The document space provides many possibilities to structure the content according to the users' mental model of the domain. According to insights from cognitive psychology the relevant knowledge is situated in the so-called semantic memory. The semantic memory is a graph where domain concepts are forming the nodes while weighted edges define the semantic relatedness of two connected concepts. To simplify the alignment of a knowledge system's content to the expert's mental model, the document space provides means to adapt the content structure towards (a subset of) the semantic memory graph.

The document-centered approach also supports the social process of knowledge acquisition, requiring exchange of expertise along the dimensions knowledge engineering and domain knowledge. Considering the dimension of domain expertise, cycles of externalization and internalization of knowledge by the participants can lead to knowledge convergence, considering the document base and the participants' knowledge. On the knowledge engineering dimension, informal learning can be performed during the knowledge acquisition activities, supported by training examples. For the authoring of multimodal knowledge, a human-computer interaction model based on interactive alignment is proposed, supported by techniques like autocompletion and

special purpose editors.

When compared to GUI-based interfaces, the document-centered approach has several advantages. It allows for basic contributions at very low technical barriers for the user and provides large freedom of structuring. Additionally, the approach allows for incremental formalization and example-based authoring. Also, the documents allow for a convenient quality management process by versioning and continuous integration. There are however also considerable challenges to be overcome for successful document-centered knowledge acquisition, such as authoring assistance and solutions for navigation and search within the documents. Further, the tasks of content refactoring, redundancy detection, and knowledge base debugging need to be addressed.

A category of tools which is related to DCKA is semantic wikis. As a semantic wiki allows for simple editing of documents (wiki pages) in a collaborative way, they provide a reasonable basis for creating a document-centered knowledge acquisition environment.

The document-centered knowledge acquisition approach can be applied for a wide-range of knowledge acquisition scenarios, as for instance the creation of ontologies, diagnostic knowledge systems, data analysis background knowledge, or case-based e-Learning systems.

DCKA in several aspects strongly differs from the heavy-weight methodology CommonKADS, which is strictly applying indirect knowledge acquisition. Nevertheless, the approaches can be combined in a reasonable way, for instance by using DCKA in the implementation phase of CommonKADS.

8.1.4 A Meta-Engineering Approach for DCKA

One major aspect for lowering the barriers for widespread participation on the knowledge acquisition process is the use of customized knowledge acquisition tools. The systems design dilemma, describes the problem to anticipate all use-time aspects of a system at design-time. That problem is in particular relevant for knowledge acquisition tools, considering the competency dilemma complicating the specification process even more. Therefore, instead of the design of a custom tool a priori, we propose to drive an ongoing customization process in parallel to the actual knowledge acquisition process, the 'meta-engineering process'. This continuous design process aims to adapt the content representation towards the specialists' mental model of the domain and allows for comprehensible representation of the domain knowledge. The 'knowledge acquisition architecture' (KAA) describes the currently aspired document-space structure. Beside the possibilities for structuring the document space, the choice of the employed markup language constitutes a degree of freedom with high potential for customization. The meta-engineering process coordinates the design and implementation of project specific custom-tailored markups that capture the required knowledge in a simple and comprehensive way. For this purpose, there are a number of guidelines for markup design that can be referred to. A markup design prototype can easily be assessed by using it for a small part of the knowledge base (, while not being processed by the system). In that way, the design process allows for the quick and cheap evaluation of all candidate designs for determining the best solution. During the implementation process of the new markup, it can already be employed within the documents. The corresponding markup expressions will be recognized by the system after the next software update, when the corresponding implementation work has been finished. The adapta-

8 Conclusion

tion of the document space towards the specialist's mental model and the increasing experiences with document-centered knowledge acquisition are forming a co-adaptive process, leading to increased knowledge engineering capabilities of the specialists over time. The major technical challenges arising from the meta-engineering process are the frequent need for refactoring of the document content and the implementation of new markups or the extension of existing ones. Hence, effective technical support of these tasks is important.

8.1.5 Techniques for the Implementation of DCKA

The task of compiling knowledge markups within documents into an executable knowledge base repository at first glance resembles the compilation task of general purpose programming languages in classical software engineering. A thorough analysis of the task (c.f. Section 5.1) however reveals that the requirements at several points show significant differences, as for example the frequent introduction of new languages or language components and the simplicity of the markup languages. It turns out that standard techniques in some cases are either not applicable or not practicable. Therefore, we present methods, that are adapted to the conditions of document-centered knowledge acquisition.

The 'KDOM data-structure' is proposed for the management of multimodal knowledge. It is well suited to support refactoring on the document and knowledge base level. To build up the KDOM data-structure for the documents, a top-down parsing mechanism with multiple extensions can be employed. The process of translating the markup expressions to the executable knowledge base repository should perform reference resolution for objects in markup expressions to prevent misspelling or miss-use of terminology objects by the user. We introduce an algorithm that performs reference resolution and knowledge base update in an incremental way. That algorithm makes the computation time for knowledge compilation independent of the overall knowledge/document base size.

A major contribution of Chapter 4 is a holistic approach for the timely and cost-effective implementation of new markups as demanded by the meta-engineering approach. For this purpose, a language is designed, which allows for the (semi-) declarative implementation of knowledge markups. The mechanism, which interprets this language is based on the top-down parsing and reference resolution algorithms introduced before. The chapter can serve as a guideline for the implementation of a document-centered authoring system core, which meets the requirements posed by the meta-engineering approach.

8.1.6 KnowWE - An Authoring Environment for DCKA

The KnowWE system is an implementation of a document-centered authoring environment, which is designed to support the meta-engineering approach. It is based on an open source wiki engine, i.e., each wiki page is considered as one document. KnowWE uses the KDOM data-structure and parsing algorithm discussed in Chapter 5. In that way, a good basis for refactoring and for the integration of new markups is provided. The highly extensible architecture provides optimal conditions for project specific customizations. It provides frameworks for syntax checking including reference resolution, code-completion, drag-and-drop editing, continuous integration testing, and refactoring.

The system has proven to be suitable for document-centered knowledge acquisition with meta-engineering in a number of academic and industrial knowledge engineering projects. The tool, being written in Java, is distributed as open source and maintained by the denkbares GmbH.

8.1.7 Case Studies

The meta-engineering approach for document-centered knowledge acquisition has been applied to five real-world case studies from different domains: 7.1: A rule-based expert system for the determination of suitable computer-interaction devices for handicapped people; 7.2: A decision-support and information system to support ophthalmologists in cataract surgery. 7.3: A decision-support and documentation system for the assessment of potentially dangerous chemical substances. 7.4: An intelligent system to support the precise, multi-lingual, and computer-interpretable acquisition of malfunction problems in the technical service support and maintenance. 7.5: An interactive ontology-based e-Learning platform for students in the domain of Ancient History.

A wide range of different knowledge representations have been employed in these different projects. In each project, the meta-engineering process has led to beneficial improvements for the overall knowledge engineering project. Therefore, the case studies show that the meta-engineering approach is applicable and helpful in a wide range of project scenarios, independent of the domain and employed knowledge representation.

8.2 Outlook

This work provides a comprehensive introduction of the basics of the novel approach—document-centered knowledge acquisition with markup languages. However, considering the broadness of the approach, there are still interesting research questions to be explored. Also, some new ideas are just being arising recently during the application of the approach in practice. In the following, some of these ideas for improving and extending the approach in the future are outlined.

8.2.1 A Catalogue of Markup Design Guidelines

The design of markups that facilitate the capture of the knowledge in the current project is one of the most important tasks within the meta-engineering approach. The design of markups is a creative activity and decisions can have major implications on the long term. Hence, reliable guidelines for that task have great value. In Section 4.4.2.1 we presented a collection of markup design principles, which have either been taken from DSL literature or that have been derived from our experiences from the cases studies. However, extensive long term experiences with meta-engineering projects could not be made yet. One important task for the future would be to collect more experiences from as many projects as possible and evolve a catalogue of design principles. An example set of designed markups, discussing their advantages and disadvantages, should be included.

8.2.2 Combination with Heavy-weight Knowledge Acquisition Approaches

In Section 3.6 the CommonKADS methodology is briefly introduced. The correlation and differences of CommonKADS and DCKA are discussed. While the heavy weight methodology CommonKADS proposes comprehensive top-down specification and indirect knowledge acquisition, the document-centered approach employs direct (mixed-initiative) knowledge acquisition, embedded in an agile development process. Despite the differences in the methodological orientation of the approaches, promising possibilities for the combination are outlined. This is in accordance to the recent trends that can be observed in software development. Often, neither traditional nor agile development is performed in its pure form, but combinations, mixing up the respective methods, are pursued. However, the combination of DCKA and CommonKADS has not yet been explored in practice. The application of the combined approach in real-world knowledge engineering projects could provide interesting and valuable results.

8.2.3 Formal Definition of the KAA

In Section 4.3.2 the knowledge acquisition architecture is introduced. It specifies how the document space should be developed and in that way serves as a guideline for the user on contributions. It is only a weak guideline, still allowing full freedom for contributions of arbitrary kinds. This however implies, that the contributions need to be reviewed with respect to their compliance with the KAA. These reviews constitute tedious manual work to be performed by experienced participants. An approach to alleviate this problem is to establish a mechanism that (at least to some extent) is able check the compliance of documents with the current KAA automatically.

One requirement for this task is, that the KAA is somehow defined in a formal way to be recognized and processed by the system. The structure of particular documents needs to be specified as precisely as possible. This implies to define what kinds of content elements, i.e. the elements of which category in what order, are expected. For this formal specification, a kind of constraint language could be employed, defining for particular categories of documents particular orders of content categories, possibly including the expected size of the respective content elements.

Beside the KAA, also the structure of the actual documents needs to be recognized by the system in the required level of detail. A priori, a document-centered authoring environment as discussed in this work only distinguishes the content category markup expression from the rest of the content. It however can not distinguish modeling description from domain description or organizational knowledge. Enabling this distinction would be an important step towards checking compliance to a formal KAA specification. This can be achieved by markup languages, similar as for example *Markdown*¹.

Assumed the specified KAA constraints can be checked automatically, violations can and should be used to create hints and todo messages to indicate the (temporary) insufficiency. While becoming possible then, we strongly recommend not to strictly force for contributions the compliance to the KAA as this contradicts and destroys the principle of free and easy contributions at low barriers. These notification messages should be displayed on a centralized dashboard or/and be sent by mail to the responsible knowledge gardener (c.f. Section 4.4.4). In that way,

¹<http://daringfireball.net/projects/markdown/>

adherence to the KAA could be ensured at lower manual efforts and degeneration of the content can be prevented more easily. Establishing a reasonable constraint or description language to formalize the KAA in the first place constitutes a considerable research question.

8.2.4 User Interface and Deployment

In this work an approach for the development of knowledge bases for intelligent systems is introduced. It focuses on the formation of the knowledge base by an agile knowledge acquisition process, however not regarding its deployment into the intended productive setting. In classical knowledge-based systems the user interface for the end user also plays an important role and needs to be incorporated within the deployment task. The design and implementation of this user interface has not been discussed in this work. Its inclusion into the overall knowledge engineering process however could be quite beneficial and poses an aspect for further research. Then not only the correct behavior of the knowledge system, but also its practicability and usability from a end user perspective can be tested in short cycles. Some work in this direction is provided by Freiberg et al. [FSP12].

8.2.5 Learning Material

One central aspect of the approach is the social process of learning. The development of basic knowledge engineering expertise by the domain specialists is assumed to be divided in activities of formal and informal learning.

In the first place, the basic usage of the document-centered authoring environment, e.g., considering contribution to the domain descriptions, is trained in a formal style at the beginning of the project. For that purpose, learning material, possibly for conducting a kind of interactive workshop, could be provided. With an established best practice for this kind of beginners lesson, the starting phase of the project could be performed very efficiently.

Second, even more important, is the support of the informal learning activities, that are performed throughout the entire project progression. These learning activities arise spontaneously during work typically without a knowledge engineer being present (in the first place). Hence, learning material is required, which is suited to support as far as possible independent learning. The provision of fully functional and consistent examples was proposed to be a valuable strategy in this context (c.f. example-based learning, Section 3.1.5.3). On the other hand, the material should also indicate when the domain specialists should contact a knowledge engineer to support the current learning task. Informal learning does not necessarily imply to be performed by the learner alone in a completely independent way.

A collection of suitable learning material, considering both aspects, would be of great value for the document-centered knowledge acquisition approach. The content and style of the material needs to be derived from experiences of projects in various domains with participants of different backgrounds. Its didactic preparation should primarily be oriented to the perspective of the indented target audience, being domain specialists without knowledge engineering expertise. Therefore, the material should probably not (entirely) be created by a knowledge engineer, but be established by participation of domain specialists, which are just gaining experiences in DCKA.

8.3 Discussion

We can look back today to a long history of knowledge acquisition research. Despite constant progress in foundational methods, the development of a knowledge system is still far from being a routine activity. While the nature of the challenge of knowledge acquisition remained unchanged, the world has changed greatly within the last twenty years. The way, how information technology plays a continuously increasing role in all kinds of private and professional activities, provides significantly different conditions considering the socio-technical aspects of knowledge acquisition.

Collaborative knowledge acquisition by a distributed team has been studied for several years now as its potential has been recognized [SG96] already before the necessary technology spread out. At the time of the millennium, Schilstra and Spronck [SS01] for instance proposed a collaborative knowledge acquisition approach. While the term at that time not yet had been widely established, one clearly can classify it as an *agile* knowledge engineering methodology, that had been proposed. Many of the general ideas of that approach, including for example the active participation of domain specialists, very much comply to the way the problem is addressed in this work. We claim however, that the use of documents provides the best way to support that kind of collaboration. Remarkably, the use of documents as a basis for collaborative knowledge engineering has not been explored systematically and extensively by now. The possibilities for flexible adaptation towards the involved persons and the subject domain, which is focus of this work, provides advantages that can hardly be achieved by other methods. The results, that we could observe until now in our projects, indicate that the approach is well suited to exploit the potential provided by a group of professionals to successfully conduct a knowledge engineering project.

As a part of mixed-initiative knowledge acquisition, the approach incorporates the strategy of direct knowledge acquisition, which is discussed controversially within the research community. The key question for this topic is, whether direct knowledge acquisition activities performed by domain experts can achieve a quantity and quality making it in the end worthwhile for the overall project. Skeptics argue that domain specialists usually never are capable to fully understand the semantics of a complex knowledge representation language and therefore *cannot* perform the knowledge modeling task properly. To provide our opinion for this questioning, we would like to employ an analogy from software development as discussed by Decker [Dec98]: How many programmers (be it C++, Java, Prolog, or any other) fully understand the semantics of their language in its entirety? Well, when being honest, one has to confirm that many do not. Nevertheless, they successfully create many useful things with these languages. We claim that this also can hold for direct—or rather mixed-initiative—knowledge acquisition. If the conditions are adapted towards the persons needs, ruling out unnecessary difficulties for cognitively accessing the knowledge, effective contributions become possible. According to the experiences made, we are confident that the document-centered knowledge acquisition approach with meta-engineering can help to create these conditions.

Abbreviations

BNF	Backus Naur Form
CI	Continuous Integration
DB	Document Base
DCKA	Document-Centered Knowledge Acquisition
DSL	Domain Specific Language
GUI	Graphical User Interface
IDE	Integrated Development Environment
KA	Knowledge Acquisition
KAA	Knowledge Acquisition Architecture
KB	Knowledge Base
KDOM	Knowledge Document Object Model
KE	Knowledge Engineering
MIKA	Mixed-Initiative Knowledge Acquisition
OWL	Web Ontology Language
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema

Bibliography

- [ADRW00] Robert K. Atkinson, Sharon J. Derry, Alexander Renkl, and Donald Wortham. Learning from examples: Instructional principles from the worked examples research. *Review of Educational Research*, 70:181–214, 2000.
- [AE10] Heather L. Ainsworth and Sarah E. Eaton. *Formal, Non-Formal and Informal Learning in the Sciences*. Onate Press, 2010.
- [AGT89] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code Generation Using Tree Matching and Dynamic Programming. *ACM Trans. Program. Lang. Syst.*, 11(4):491–516, October 1989.
- [ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.
- [BA04] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley, Boston, 2004.
- [Bas06] H. Bast. Type Less, Find More: Fast Autocompletion Search with a Succinct Index. In *29th conference on research and development in informtion retrieval (SIGIR'06)*, pages 364–371, 2006.
- [Bau04] Joachim Baumeister. *Agile Development of Diagnostic Knowledge Systems*. IOS Press, AKA, DISKI 284, 2004.
- [Bau11] Joachim Baumeister. Advanced empirical testing. *Knowledge-Based Systems*, 24(1):83–94, 2011.
- [Bec00] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.
- [Ben83] Herbert D. Benington. Production of large computer programs. *IEEE Ann. Hist. Comput.*, 5(4):350–361, October 1983.
- [BF10] Joachim Baumeister and Martina Freiberg. Knowledge visualization for evaluation tasks. *Knowledge and Information Systems*, submitted Jan 15, 2010.
- [BGS⁺11] Michel Buffa, Fabien Gandon, Peter Sander, Catherine Faron, and Guillaume Ereteo. SweetWiki: A semantic wiki. *Web Semantics: Science, Services and Agents on the World Wide Web*, 6(1), 2011.

BIBLIOGRAPHY

- [BKKZ92] Reinhard Budde, Karlheinz Kautz, Karin Kuhlenkamp, and Heinz Züllinghoven. *Prototyping: An Approach to Evolutionary System Development*. Springer, Berlin, 1992.
- [BKMZ84] Reinhard Budde, Karin Kuhlenkamp, Lars Mathiassen, and Heinz Züllinghoven. *Approaches to Prototyping*. Springer, Berlin, 1984.
- [BKS07] Joachim Baumeister, Thomas Kleemann, and Dietmar Seipel. Towards the Verification of Ontologies with Rules. In *FLAIRS'07: Proceedings of the 20th International Florida Artificial Intelligence Research Society Conference*, pages 524–529. AAAI Press, 2007.
- [BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, May 2001.
- [Boe88] Barry W. Boehm. A Spiral Model of Software Development and Enhancement. *Computer*, 21(5):61–72, May 1988.
- [BR11] Joachim Baumeister and Jochen Reutelshoefer. Developing Knowledge Systems with Continuous Integration. In *i-KNOW 2011: 11th International Conference on Knowledge Management and Knowledge Technologies*, Graz, Austria, 2011. ACM ICPS.
- [BRB⁺12] Joachim Baumeister, Jochen Reutelshoefer, Volker Belli, Albrecht Striffler, Reinhard Hatko, and Markus Friedrich. KnowWE - A Wiki for Knowledge Base Development. In *The 8th Workshop on Knowledge Engineering and Software Engineering (KESE2012)*, 2012.
- [Bro09] Paul Browne. *JBoss Drools Business Rules*. Packt Publishing, 2009.
- [BRP07a] Joachim Baumeister, Jochen Reutelshoefer, and Frank Puppe. KnowWE: Community-based Knowledge Capture with Knowledge Wikis. In *K-CAP '07: Proceedings of the 4th international conference on Knowledge capture*, pages 189–190, New York, NY, USA, 2007. ACM.
- [BRP07b] Joachim Baumeister, Jochen Reutelshoefer, and Frank Puppe. Markups for Knowledge Wikis. In *SAAKM'07: Proceedings of the Semantic Authoring, Annotation and Knowledge Markup Workshop*, pages 7–14, Whistler, Canada, 2007.
- [BS84] B.G. Buchanan and E.H. Shortliffe. *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Addison-Wesley, 1984.
- [BS06] Joachim Baumeister and Dietmar Seipel. Verification and Refactoring of Ontologies With Rules. In *EKAW'06: Proceedings of the 15th International Conference on Knowledge Engineering and Knowledge Management*, pages 82–95, Berlin, 2006. Springer.

- [BSBN13] Joachim Baumeister, Albrecht Striffler, Marc Brandt, and Michael Neumann. Towards Continuous Knowledge Representations in Episodic and Collaborative Decision Making. In *KESE9: 9th Workshop on Knowledge Engineering and Software Engineering*, 2013.
- [BSP04] Joachim Baumeister, Dietmar Seipel, and Frank Puppe. Refactoring methods for knowledge bases. In *EKAW'04: Engineering Knowledge in the Age of the Semantic Web: 14th International Conference, LNAI 3257*, pages 157–171, Berlin, 2004. Springer.
- [CJhH02] Murray Campbell, A. Joseph Hoane Jr., and Feng hsiung Hsu. Deep Blue. *Artificial Intelligence*, 134(1-2):57–83, 2002.
- [CK07] Ulrike Cress and Joachim Kimmerle. A theoretical framework of collaborative knowledge building with wikis: a systemic and cognitive perspective. In *Proceedings of the 8th international conference on Computer supported collaborative learning, CSCL'07*, pages 156–164. International Society of the Learning Sciences, 2007.
- [CK08] Ulrike Cress and Joachim Kimmerle. A systemic and cognitive view on collaborative knowledge building with wikis. *International Journal of Computer-Supported Collaborative Learning*, 3(2):105–122, June 2008.
- [CL75] Allan M. Collins and Elizabeth F. Loftus. A Spreading-Activation Theory of Semantic Processing. *Psychological Review*, 82(6):407 – 428, 1975.
- [CM08] Michel Chein and Marie-Laure Mugnier. *Graph-based Knowledge Representation: Computational Foundations of Conceptual Graphs*. Springer, London, 2008.
- [Coc02] Alistair Cockburn. *Agile Software Development*. Addison-Wesley, 2002.
- [CQ95] Allan M. Collins and M. Ross Quillian. Computation & Intelligence. chapter Retrieval Time from Semantic Memory, pages 191–201. American Association for Artificial Intelligence, Menlo Park, CA, USA, 1995.
- [Dec98] Stefan Decker. On Domain-Specific Declarative Knowledge Representation and Database Languages. In Alexander Borgida, Vinay K. Chaudhri, and Martin Staudt, editors, *KRDB*, volume 10 of *CEUR Workshop Proceedings*, pages 9.1–9.7. CEUR-WS.org, 1998.
- [DFGR12] Chiara Di Francescomarino, Chiara Ghidini, and Marco Rospocher. Evaluating wiki-enhanced ontology authoring. In *Proceedings of the 18th international conference on Knowledge Engineering and Knowledge Management, EKAW'12*, pages 292–301, Berlin, Heidelberg, 2012. Springer-Verlag.
- [Dij86] Edsger W. Dijkstra. On a cultural gap. *The Mathematical Intelligencer*, 8(1):48 – 52, 1986.

BIBLIOGRAPHY

- [DSW⁺00] A. J. Duineveld, R. Stoter, M. R. Weiden, B. Kenepa, and V. R. Benjamins. Wondertools? a comparative study of ontological engineering tools. *International Journal of Human-Computer Studies*, 52(6):1111–1133(23), June 2000.
- [Ebb85] H. Ebbinghaus. *Über das Gedächtnis: Untersuchungen zur experimentellen Psychologie*. Duncker & Humblot, 1885.
- [EEMT87] Larry J. Eshelman, Damien Ehret, John P. McDermott, and Ming Tan. MOLE: A Tenacious Knowledge-Acquisition Tool. *International Journal of Man-Machine Studies*, 26(1):41–54, 1987.
- [EM93] Henrik Eriksson and Mark Musen. Metatools for Knowledge Acquisition. *IEEE Softw.*, 10:23–29, May 1993.
- [EPG⁺95] Henrik Eriksson, Angel R. Puerta, John H. Gennari, Thomas E. Rothenuh, Samson W. Tu, and Mark A. Musen. Custom-Tailored Development Tools for Knowledge-Based Systems. Technical report, Stanford University School of Medicine, 1995.
- [Eri92] Henrik Eriksson. Metatool support for custom-tailored, domain-oriented knowledge acquisition. *Knowledge Acquisition*, 4(4):445 – 476, 1992.
- [ESA05] N. Ernst, M. A. Storey, and P. Allen. Cognitive support for ontology modeling. *International Journal of Human-Computer Studies*, May 2005.
- [FBCC⁺10] David Ferrucci, Eric Brown, Jennifer Chu-Carroll, James Fan, David Gondek, Aditya A. Kalyanpur, Adam Lally, J. William Murdock, Eric Nyberg, John Prager, Nico Schlaefel, and Chris Welty. Building Watson: An Overview of the DeepQA Project. *AI Magazine*, 31(3), 2010.
- [Fei77] E. A. Feigenbaum. The Art of Artificial Intelligence: Themes and Case Studies of Knowledge Engineering. In *Proc. of the 5th IJCAI*, pages 1014–1029, Cambridge, MA, 1977.
- [FG06] Gerhard Fischer and Elisa Giaccardi. Meta-design: A Framework for the Future of End-User Development. In Henry Lieberman, Fabio Paternò, and Volker Wulf, editors, *End User Development*, volume 9 of *Human-Computer Interaction Series*, chapter 19, pages 427–457. Springer Netherlands, Dordrecht, 2006.
- [FGY⁺04] G. Fischer, E. Giaccardi, Y. Ye, A. G. Sutcliffe, and N. Mehandjiev. Meta-Design: A Manifesto for End-User Development. *Commun. ACM*, 47(9):33–37, September 2004.
- [Fin13] E. O. Finkenbinder. The Curve of Forgetting. *The American Journal of Psychology*, 24(1):pp. 8–32, 1913.
- [FMPS10] Alexander Felfernig, Monika Mandl, Anton Pum, and Monika Schubert. Empirical Knowledge Engineering: Cognitive Aspects in the Development of

- Constraint-Based Recommenders. In Nicolas Garcia-Pedrajas, Francisco Herrera, Colin Fyfe, JoseManuel Benitez, and Moonis Ali, editors, *Trends in Applied Intelligent Systems*, volume 6096 of *Lecture Notes in Computer Science*, pages 631–640. Springer Berlin Heidelberg, 2010.
- [For82] Charles Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligences*, 19(1):17–37, 1982.
- [Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.
- [Fow10] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.
- [Fre79] Gottlob Frege. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Verlag von Louis Nebert, Halle, 1879.
- [Fri02] Jeffrey E. F. Friedl. *Mastering Regular Expressions*. O’Reilly & Associates, Inc., Sebastopol, CA, USA, 2 edition, 2002.
- [FS00] Gerhard Fischer and Eric Scharff. Meta-Design: Design for Designers. In *Proceedings of the 3rd conference on Designing interactive systems: processes, practices, methods, and techniques*, DIS ’00, pages 396–405, New York, NY, USA, 2000. ACM.
- [FSA⁺94] Otto K. Ferstl, Elmar J. Sinz, Michael Amberg, Udo Hagemann, and Carsten Malischewski. Tool-Based Business Process Modeling Using the SOM Approach. In *GI Jahrestagung 1994*, pages 430–436, 1994.
- [FSP12] Martina Freiberg, Albrecht Striffler, and Frank Puppe. Extensible Prototyping for Pragmatic Engineering of Knowledge-based Systems. *Expert Syst. Appl.*, 39(11):10177–10190, September 2012.
- [FvHH⁺01] D. Fensel, F. van Harmelen, I. Horrocks, D. McGuinness, and P. F. Patel-Schneider. OIL: An Ontology Infrastructure for the Semantic Web. *IEEE Intelligent Systems*, 16(2):38–45, 2001.
- [GGM95] D. A. Giuse, N. B. Giuse, and R. A. Miller. Evaluation of long-term maintenance of a large medical knowledge base. *American Medical Informatics Association*, 2(5):297–306, 1995.
- [GLR09] Jana Giceva, Christoph Lange, and Florian Rabe. Integrating Web Services into Active Mathematical Documents. In Jacques Carette, Lucas Dixon, Claudio S. Coen, and Stephen M. Watt, editors, *Intelligent Computer Mathematics*, volume 5625 of *Lecture Notes in Computer Science*, pages 279–293. Springer Berlin Heidelberg, 2009.

BIBLIOGRAPHY

- [GMF⁺03] John H. Gennari, Mark A. Musen, Ray W. Ferguson, William E. Grosso, Monica Crubezy, Henrik Eriksson, Natalya F. Noy, and Samson W. Tu. The Evolution of Protégé: An Environment for Knowledge-based Systems Development. *Int. J. Hum.-Comput. Stud.*, 58(1):89–123, 2003.
- [GMSZ08] Blaise Genest, Anca Muscholl, Olivier Serre, and Marc Zeitoun. Tree Pattern Rewriting Systems. In Sung-Deok Cha, Jin-Young Choi, Moonzoo Kim, Insup Lee, and Mahesh Viswanathan, editors, *ATVA*, volume 5311 of *Lecture Notes in Computer Science*, pages 332–346. Springer, 2008.
- [GPS93] Ute Gappa, Frank Puppe, and Stefan Schewe. Graphical knowledge acquisition for medical diagnostic expert systems. *Artificial Intelligence in Medicine*, 5(3):185 – 211, 1993.
- [GS97] Brian R. Gaines and Mildred L. G. Shaw. Knowledge acquisition, modelling and inference through the World Wide Web. *Int. J. Hum.-Comput. Stud.*, 46(6):729–759, June 1997.
- [GS99] Brian R. Gaines and Mildred L. G. Shaw. Embedding formal knowledge models in active documents. *Commun. ACM*, 42(1):57–64, January 1999.
- [GT97] Yolanda Gil and Marcelo Tallis. A Script-Based Approach to Modifying Knowledge Bases. In *AAAI/IAAI'97: Proceedings of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference*, pages 377–383, 1997.
- [HA08] Josef Hofer-Alfeis. Knowledge management solutions for the leaving expert issue. *J. Knowledge Management*, 12(4):44–54, 2008.
- [HBBP12] Reinhard Hatko, Joachim Baumeister, Volker Belli, and Frank Puppe. Diaflux: A Graphical Language for Computer-Interpretable Guidelines. In David Riano, Annette ten Teije, and Silvia Miksch, editors, *Knowledge Representation for Health-Care*, volume 6924 of *Lecture Notes in Computer Science*, pages 94–107. Springer, 2012.
- [HBM⁺04] Matthias Hüttig, Georg Buscher, Thomas Menzel, Wolfgang Scheppach, Frank Puppe, and Hans-Peter Buscher. A Diagnostic Expert System for Structured Reports, Quality Assessment, and Training of Residents in Sonography. *Medizinische Klinik*, 3:117–22, 2004.
- [HDG⁺06] Matthew Horridge, Nick Drummond, John Goodwin, Alan Rector, Robert Stevens, and Hai H Wang. The Manchester OWL Syntax. In Bernardo Cuenca Grau, Pascal Hitzler, Connor Shankey, and Evan Wallace, editors, *Proceedings of OWL: Experiences and Directions (OWLED'06)*, Athens, Georgia, USA, 2006.
- [Hen88] James Hendler. *Expert Systems: The User Interface*. Human/Computer Interaction. Ablex Publishing, Norwood, NJ, 1988.

- [Hig01] Jim Highsmith. The Great Methodologies Debate: Part 1. *Cutter IT Journal*, 2001.
- [Hil03] Ernest Friedman Hill. *Jess in Action: Java Rule-Based Systems*. Manning Publications Co., Greenwich, CT, USA, 2003.
- [HJ02] Clyde W. Holsapple and K. D. Joshi. A collaborative approach to ontology design. *Commun. ACM*, 45(2):42–47, February 2002.
- [HM00] Eva Heinrich and Hermann A. Maurer. Active Documents: Concept, Implementation and Applications. *J. UCS*, 6(12):1197–1202, 2000.
- [Hor02] Ian Horrocks. DAML+OIL: a description logic for the semantic web. *Bull. of the IEEE Computer Society Technical Committee on Data Engineering*, 25(1):4–9, March 2002.
- [HPSvH03] Ian Horrocks, Peter F. Patel-Schneider, and Frank van Harmelen. From *S^HI²* and RDF to OWL: The making of a web ontology language. *J. of Web Semantics*, 1(1):7–26, 2003.
- [HR04] Rosco Hill and Joe Rideout. Automatic Method Completion. *Automated Software Engineering, International Conference on*, 0:228–235, 2004.
- [HRWL83] Frederick Hayes-Roth, Donald A. Waterman, and Douglas B. Lenat. *Building expert systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.
- [HSE11] Philipp Heim, Thomas Schlegel, and Thomas Ertl. A Model for Human-Computer Interaction in the Semantic Web. In *Proceedings of the 7th International Conference on Semantic Systems, I-Semantics '11*, pages 150–158, New York, NY, USA, 2011. ACM.
- [HSM⁺12] Reinhard Hatko, Dirk Schädler, Stefan Mersmann, Joachim Baumeister, Norbert Weiler, and Frank Puppe. Implementing an Automated Ventilation Guideline Using the Semantic Wiki KnowWE. In Annette ten Teije, Johanna Völker, Siegfried Handschuh, Heiner Stuckenschmidt, Mathieu d’Aquin, Andriy Nikolov, Nathalie Aussenac-Gilles, and Nathalie Hernandez, editors, *EKAW*, volume 7603 of *Lecture Notes in Computer Science*, pages 363–372. Springer, 2012.
- [HvHtT05] Zhisheng Huang, Frank van Harmelen, and Annette ten Teije. Reasoning with Inconsistent Ontologies. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI*, pages 454–459. Professional Book Center, 2005.
- [HWM09] Sangmok Han, David R. Wallace, and Robert C. Miller. Code Completion from Abbreviated Input. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, ASE '09*, pages 332–343, Washington, DC, USA, 2009. IEEE Computer Society.

BIBLIOGRAPHY

- [JGD04] Jelena Jovanović, Dragan Gašević, and Vladan Devedić. A GUI for Jess. *Expert Syst. Appl.*, 26(4):625–637, May 2004.
- [Joh75] S. C. Johnson. YACC—yet another compiler-compiler. Technical Report CS-32, AT & T Bell Laboratories, Murray Hill, N.J., 1975.
- [Jon88] Sara Jones. Graphical interfaces for knowledge engineering: an overview of relevant literature. *The Knowledge Engineering Review*, 3(03):221–247, 1988.
- [KBD⁺89] G. Klinker, C. Boyd, D. Dong, J. Maiman, J. McDermott, and R. Schnelbach. Building expert systems with KNACK. *Knowledge Acquisition*, 1(3):299–320, 1989.
- [Kel55] G.A. Kelly. *The Psychology of Personal Constructs*. Number Bd. 2 in The Psychology of Personal Constructs. Norton, 1955.
- [KFK99] G. J. Kuperman, J. M. Fiskio, and A. Karson. A process to maintain the quality of a computerized knowledge base. *Proc AMIA Symp*, pages 87–91, 1999.
- [KFNM04] Holger Knublauch, Ray W. Ferguson, Natalya F. Noy, and Mark A. Musen. The Protégé OWL Plugin: An Open Development Environment for Semantic Web Applications. In Sheila A. McIlraith, Dimitris Plexousakis, and Frank van Harmelen, editors, *The Semantic Web – ISWC 2004*, volume 3298 of *Lecture Notes in Computer Science*, pages 229–243. Springer Berlin / Heidelberg, 2004.
- [KKG⁺07] Dierk Koenig, Andrew Glover, Paul King, Guillaume Laforge, and Jon Skeet. *Groovy in Action*. Manning Publications Co., Greenwich, CT, USA, 2007.
- [KKP⁺09] Gabor Karsai, Holger Krahn, Class Pinkernell, Bernhard Rumpe, Martin Schneider, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM09)*, 2009.
- [Kli08] Pavel Klinov. Pronto: A Non-monotonic Probabilistic Description Logic Reasoner. In Sean Bechhofer, Manfred Hauswirth, Jörg Hoffmann, and Manolis Koubarakis, editors, *The Semantic Web: Research and Applications, 5th European Semantic Web Conference, ESWC 2008, Tenerife, Canary Islands, Spain, June 1-5, 2008, Proceedings*, volume 5021 of *Lecture Notes in Computer Science*, pages 822–826. Springer, 2008.
- [Klo92] J. W. Klop. Handbook of Logic in Computer Science (vol. 2). chapter Term Rewriting Systems, pages 1–116. Oxford University Press, Inc., New York, NY, USA, 1992.
- [KM06] Mik Kersten and Gail C. Murphy. Using Task Context to Improve Programmer Productivity. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, SIGSOFT '06/FSE-14*, pages 1–11, New York, NY, USA, 2006. ACM.

- [KNM85] Gary S. Kahn, Steven J. Nowlan, and John P. McDermott. MORE: An Intelligent Knowledge Acquisition Tool. In Aravind K. Joshi, editor, *IJCAI*, pages 581–584. Morgan Kaufmann, 1985.
- [Kno50] M.S. Knowles. *Informal Adult Education: A Guide for Administrators, Leaders, and Teachers*. Association Press, 1950.
- [Knu64] Donald E. Knuth. backus normal form vs. Backus Naur form. *Communications of the ACM*, 7(12):735–736, 1964.
- [Knu02] Holger Knublauch. *An Agile Development Methodology for Knowledge-Based Systems Including a Java Framework for Knowledge Modeling and Appropriate Tool Support*. PhD thesis, University of Ulm, 2002.
- [Kre12] S. Kreutzer. *Ein Expertensystem zur Unterstützung körperbehinderter Menschen*. Diplomica, 2012.
- [KSV07] Markus Krötzsch, Sebastian Schaffert, and Denny Vrandečić. Reasoning in Semantic Wikis. In Grigoris Antoniou, Uwe Aßmann, Cristina Baroglio, Stefan Decker, Nicola Henze, Paula-Lavinia Patranjan, and Robert Tolksdorf, editors, *Reasoning Web*, volume 4636 of *Lecture Notes in Computer Science*, pages 310–329. Springer, 2007.
- [Kuh08] Tobias Kuhn. AceWiki: Collaborative Ontology Management in Controlled Natural Language. In Christoph Lange, Sebastian Schaffert, Hala Skaf-Molli, and Max Völkel, editors, *Proceedings of the 3rd Semantic Wiki Workshop*, volume 360. CEUR Workshop Proceedings, 2008.
- [KV03] K. Kotis and G. Vouros. Human Centered Ontology Management with HCONE. In *ODS 2003 Workshop on Ontologies and Distributed Systems, Int. Joint Conf. in Artificial Intelligence (IJCAI-03)*. CEUR Workshop Proc.(CEUR-WS.org), 08/2003 2003.
- [KVV06] Markus Krötzsch, Denny Vrandečić, and Max Völkel. Semantic MediaWiki. In Isabel F. Cruz, Stefan Decker, Dean Allemang, Chris Preist, Daniel Schwabe, Peter Mika, Michael Uschold, and Lora Aroyo, editors, *International Semantic Web Conference*, volume 4273 of *Lecture Notes in Computer Science*, pages 935–942. Springer, 2006.
- [Lan08] Christoph Lange. SWiM - A Semantic Wiki for Mathematical Knowledge Management. In Sean Bechhofer, Manfred Hauswirth, Jörg Hoffmann, and Manolis Koubarakis, editors, *ESWC*, volume 5021 of *Lecture Notes in Computer Science*, pages 832–837. Springer, 2008.
- [LAY87] R. LAYTON. Expert Systems Technology: Just a Flash in the Pan? In *Proceedings of the twentieth International Symposium on the Application of Computers and Mathematics in the Mineral Industries*, page 189. SAIMM, 1987.

BIBLIOGRAPHY

- [LC01] Bo Leuf and Ward Cunningham. *The Wiki Way: Quick Collaboration on the Web*. Addison-Wesley, New York, 2001.
- [LFL05] Elizabeth Da Lio, Lucia Fraboni, and Tommaso Leo. Twiki-based facilitation in a newly formed academic community of practice. In *Proceedings of the 2005 international symposium on Wikis, WikiSym '05*, pages 85–111, New York, NY, USA, 2005. ACM.
- [LHB03] William Lidwell, Kritina Holden, and Jill Butler. *Universal Principles of Design*. Rockport Publishers, October 2003.
- [Mac11] Monica G. Maceli. Bridging the design time – use time divide: towards a future of designing in use. In *Proceedings of the 8th ACM conference on Creativity and Cognition, C&C '11*, pages 461–462, New York, NY, USA, 2011. ACM.
- [Mae06] John Maeda. *The Laws of Simplicity (Simplicity: Design, Technology, Business, Life)*. The MIT Press, August 2006.
- [Mar95] Phillip Martin. Knowledge Acquisition Using Documents, Conceptual Graphs and a Semantically Structured Dictionary. In *KAW 1995, Proceedings of the 9th International Knowledge Acquisition for Knowledge-Based Systems Workshop*, pages 1–19, 1995.
- [Mar09] R. Martin. *Clean Code: Handbook of agile software craftsmanship*. Prent. Hall, 2009.
- [MB03] Martin Molina and Gemma Blasco. Using Electronic Documents for Knowledge Acquisition and Model Maintenance. In Vasile Palade, RobertJ. Howlett, and Lakhmi Jain, editors, *Knowledge-Based Intelligent Information and Engineering Systems*, volume 2774 of *Lecture Notes in Computer Science*, pages 1357–1364. Springer Berlin Heidelberg, 2003.
- [MBH⁺12] Kivanc Muslu, Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Improving IDE recommendations by considering global implications of existing recommendations. In *Proceedings of the 2012 International Conference on Software Engineering, ICSE 2012*, pages 1349–1352, Piscataway, NJ, USA, 2012. IEEE Press.
- [McC78] John McCarthy. History of LISP. *SIGPLAN Not.*, 13(8):217–223, August 1978.
- [Mcr04] K. Mcrae. *Semantic memory: Some insights from feature-based connectionist attractor networks*, volume 45, pages 41–82. Elsevier, January 2004.
- [MCW⁺86] Mark A. Musen, David M. Combs, Joan D. Walton, Edward H. Shortliffe, and Lawrence M. Fagan. OPAL: Toward the Computer-Aided Design of Oncology Advice Systems. In *Computer Application in Medical Care*, pages 43–52, 1986.
- [Mer04] D Merrit. Best Practices for Rule-Based Application Development. *Microsoft Architects JOURNAL*, 1, 2004.

- [MHL07] Yue Ma, Pascal Hitzler, and Zuoquan Lin. Algorithms for Paraconsistent Reasoning with OWL. In *Proceedings of the 4th European conference on The Semantic Web: Research and Applications, ESWC '07*, pages 399–413, Berlin, Heidelberg, 2007. Springer-Verlag.
- [MHM12] Gregory L. Murphy, James A. Hampton, and Goran S. Milovanovic. Semantic memory redux: An experimental test of hierarchical category representation. *Journal of Memory and Language*, September 2012.
- [Min75] Marvin Minsky. Minsky's frame system theory. In *Proceedings of the 1975 workshop on Theoretical issues in natural language processing, TINLAP '75*, pages 104–116, Stroudsburg, PA, USA, 1975. Association for Computational Linguistics.
- [Mit03] Ruslan Mitkov. *The Oxford Handbook of Computational Linguistics (Oxford Handbooks in Linguistics S.)*. Oxford University Press, 2003.
- [MJ82] D. D. McCracken and M. A. Jackson. Life cycle concept considered harmful. *Software Engineering Notes*, 7(2):29–32, 1982.
- [MNTMQ96] Robert Milne, Charlie Nicol, Louise Trave-Massuyès, and Joseba Quevedo. TIGER: Knowledge Based Gas Turbine Condition Monitoring. *AI Communications*, 9(3):92–108, 1996.
- [MR02] Carlos A. Maldonado and Marc L. Resnick. Do Common User Interface Design Patterns Improve Navigation? *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, 46(14):1315–1319, 2002.
- [MS99] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. The MIT Press, Cambridge, Massachusetts, 1999.
- [Mul90] Peter W. Mullarkey. An Experiment in Direct Knowledge Acquisition. In *Proceedings of the eighth National conference on Artificial intelligence - Volume 1, AAAI'90*, pages 498–504. AAAI Press, 1990.
- [Mus88] M. A. Musen. Conceptual Models of Interactive Knowledge-Acquisition Tools. In J. Boose, B. Gaines, and M. Linster, editors, *Proc. of the European Knowledge Acquisition Workshop (EKAW'88)*, pages 26–1 – 26–15. Gesellschaft für Mathematik und Datenverarbeitung mbH, Sankt Augustin, Germany, 1988.
- [Mus89a] M. A. Musen. *Automated Generation of Model-Based Knowledge-Acquisition Tools*. Pitman Publishing London, 1989.
- [Mus89b] M. A. Musen. An editor for the conceptual models of interactive knowledge-acquisition tools. *Int. J. Man-Mach. Stud.*, 31:673–698, December 1989.
- [Mus89c] Mark A. Musen. Automated Support for Building and Extending Expert Models. *Mach. Learn.*, 4(3-4):347–375, December 1989.

BIBLIOGRAPHY

- [Mus13] Mark A. Musen. The knowledge acquisition workshops: A remarkable convergence of ideas. *Int. J. Hum.-Comput. Stud.*, 71(2):195–199, 2013.
- [New80] Allen Newell. Physical Symbol Systems. *Cognitive Science*, 4:135–183, 1980.
- [New82] A. Newell. The Knowledge Level. *Artificial Intelligence*, 28(2), 1982.
- [NL05] Grzegorz J. Nalepa and Antoni Ligeza. A graphical tabular model for rule-based logic programming and verification. *Systems Science*, 31(2):89–95, 2005.
- [NLK11] Grzegorz J. Nalepa, Antoni Ligeza, and Krzysztof Kaczor. Formalization and modeling of rules using the xtt2 method. *International Journal on Artificial Intelligence Tools*, 20(06):1107–1125, 2011.
- [OMG02] OMG. Meta Object Facility Specification. Specification Version 1.4, Object Management Group, 2002.
- [Par07] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.
- [PETM92] Angel R. Puerta, John W. Egar, Samson W. Tu, and Mark A. Musen. A Multiple-Method Knowledge-Acquisition Shell for the Automatic Generation of Knowledge-Acquisition Tools. *Knowledge Acquisition*, 4(2):171 – 196, 1992.
- [PG92] Frank Puppe and Ute Gappa. Towards Knowledge Acquisition by Experts. In Fevzi Belli and Franz Radermacher, editors, *Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, volume 604 of *Lecture Notes in Computer Science*, pages 546–555. Springer Berlin / Heidelberg, 1992. 10.1007/BFb0025008.
- [PG04] Martin J. Pickering and Simon Garrod. Toward a mechanistic psychology of dialogue. *Behavioral and Brain Sciences*, 27:169–225, 2004.
- [PGPB96] Frank Puppe, Ute Gappa, Karsten Poeck, and Stefan Bamberger. *Wissensbasierte Diagnose- und Informationssysteme. Mit Anwendung des Experten-Shell-Baukasten D3*. Springer, Berlin, 1996.
- [Pia77] Jean Piaget. *The Development of Thought: Equilibration of Cognitive Structures*. Viking Press, 1st edition, November 1977.
- [PoNR56] United States. Navy Mathematical Computing Advisory Panel and United States. Office of Naval Research. *Symposium on Advanced Programming Methods for Digital Computers: Washington, D.C., June 28, 29, 1956*. ONR symposium report. Office of Naval Research, Department of the Navy, 1956.
- [Pup90] Frank Puppe. *Problemlösungsmethoden in Expertensystemen*. Springer Verlag, 1990.

- [Pup93] Frank Puppe. Set-Covering Classification. In *Systematic Introduction to Expert Systems*, pages 156–169. Springer Berlin Heidelberg, 1993.
- [Pup00] Frank Puppe. Knowledge Formalization Patterns. In *PKAW 2000: Proceedings of the Pacific Rim Knowledge Acquisition Workshop*, Sydney, Australia, 2000.
- [Qui68] Ross Quillian. Semantic Memory. In *Semantic Information Processing*, pages 216–270. MIT Press, 1968.
- [RB13] Jochen Reutelshoefer and Joachim Baumeister. Supporting Direct Knowledge Acquisition by Customized Tools: A Case Study in the Domain of Cataract Surgery. In *Proceedings of the LWA-2013 (Special Track on Knowledge Management)*, 2013.
- [RBFP12] Jochen Reutelshoefer, Joachim Baumeister, Georg Fette, and Frank Puppe. A Document-centered Authoring Approach for Ontology Engineering. In *FGWM'12: Proceedings of German Workshop of Knowledge and Experience Management (at LWA'12)*, 2012.
- [RBP09] Jochen Reutelshoefer, Joachim Baumeister, and Frank Puppe. A Data Structure for the Refactoring of Multimodal Knowledge. In *KESE'09: 5th Workshop on Knowledge Engineering and Software Engineering (CEUR proceedings 486)*, Paderborn, 2009.
- [RHT⁺06] Stanislaus Reimer, Alexander Hörnlein, Hans-Peter Tony, Doris Kraemer, Stephan Oberück, Christian Betz, Frank Puppe, and Christian Kneitz. Assessment of a Case-Based Training System (d3web.Train) in Rheumatology. *Rheumatol Int*, 26(10):942–8, 2006.
- [RLB⁺10] Jochen Reutelshoefer, Florian Lemmerich, Joachim Baumeister, Jorit Wintjes, and Lorenz Haas. Taking OWL to Athens – Semantic Web Technology takes Ancient Greek History to Students. In *ESWC'10: Proceedings of the 7th Extended Semantic Web Conference*. Springer, 2010.
- [RLHB09] Jochen Reutelshoefer, Florian Lemmerich, Fabian Haupt, and Joachim Baumeister. An Extensible Semantic Wiki Architecture. In *SemWiki'09: Fourth Workshop on Semantic Wikis – The Semantic Wiki Web (CEUR proceedings 464)*, 2009.
- [RN03] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2 edition, 2003.
- [RP87] Stephen Regoczei and Edwin P. O. Plantinga. Creating the Domain of Discourse: Ontology and Inventory. *International Journal of Man-Machine Studies*, 27(3):235–250, 1987.
- [RSLP11] Jochen Reutelshoefer, Albrecht Striffler, Florian Lemmerich, and Frank Puppe. Incremental Compilation of Knowledge Documents for Markup-based Closed-World Authoring. In *K-CAP '11: Proceedings of the sixth international conference on Knowledge Capture*. ACM, 2011.

BIBLIOGRAPHY

- [RSS09] Barbara Russo, Marco Scotto, Alberto Sillitti, and Giancarlo Succi. *Agile Technologies in Open Source Development*. Information Science Reference - Imprint of: IGI Publishing, Hershey, PA, 2009.
- [SAA⁺01] Guus Schreiber, Hans Akkermans, Anjo Anjewierden, Robert de Hoog, Nigel Shadbolt, Walter Van de Velde, and Bob Wielinga. *Knowledge Engineering and Management - The CommonKADS Methodology*. MIT Press, 2 edition, 2001.
- [SB01] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.
- [SBBK09] Sebastian Schaffert, Francois Bry, Joachim Baumeister, and Malte Kiesel. *Semantische Wikis*, pages 245–258. Springer, 2009.
- [SBGB88] M. L. G. Shaw, J. M. Bradshaw, B. R. Gaines, and J. H. Boose. Rapid Prototyping Techniques for Expert Systems. In *Conference on Artificial Intelligence Applications*, 1988.
- [SC01] M. Stokes and MOKA Consortium. *Managing Engineering Knowledge: MOKA: Methodology for Knowledge Based Engineering Applications*. Professional Engineering Publishing, 2001.
- [SEA⁺02] York Sure, Michael Erdmann, Jürgen Angele, Steffen Staab, Rudi Studer, and Dirk Wenke. OntoEdit: Collaborative ontology development for the Semantic Web. In *ISWC'02: International Semantic Web Conference*, pages 221–235, 2002.
- [SEG⁺09] Sebastian Schaffert, Julia Eder, Szaby Grünwald, Thomas Kurz, and Mihai Radulescu. KiWi – a platform for semantic social software (demonstration). In *ESWC'09: Proceedings of the 6th European Semantic Web Conference, The Semantic Web: Research and Applications*, pages 888–892, Heraklion, Greece, June 2009. Springer.
- [SG93] Mildred L. G. Shaw and Brian R. Gaines. Personal Construct Psychology Foundations for Knowledge Acquisition and Representation. In Nathalie Aussenac-Gilles, Guy A. Boy, Brian R. Gaines, Jean-Gabriel Ganascia, Yves Kodratoff, and Marc Linster, editors, *EKAW*, volume 723 of *Lecture Notes in Computer Science*, pages 256–276. Springer, 1993.
- [SG96] Mildred L. G. Shaw and Brian R. Gaines. WebGrid: Knowledge elicitation and modeling on the Web. In *WebNet*. AACE, 1996.
- [Sha91] Nigel Shadbolt. Facts, Fantasies and Frameworks: The Design of a Knowledge Acquisition Workbench. In Franz Schmalhofer, Gerhard Strube, and Thomas Wetter, editors, *Contemporary Knowledge Engineering and Cognition*, volume 622 of *Lecture Notes in Computer Science*, pages 39–58. Springer, 1991.

- [SP09] Ben Shneiderman and Catherine Plaisant. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Pearson Addison-Wesley, 5. edition, 2009.
- [SS01] Klaas Schilstra and Pieter Spronck. Towards Continuous Knowledge Engineering. In Ann Macintosh, Mike Moulton, and Frans Coenen, editors, *Applications and Innovations in Intelligent Systems VIII*, pages 49–62. Springer London, 2001.
- [SWB93] Guus Schreiber, Bob Wielinga, and Joost Breuker. *KADS - A Principled Approach to Knowledge-Based System Development*. Academic Press, 1993.
- [TMD⁺06] Sebastian Thrun, Mike Montemerlo, Hendrik Dahlkamp, David Stavens, Andrei Aron, James Diebel, Philip Fong, John Gale, Morgan Halpenny, Gabriel Hoffmann, Kenny Lau, Celia Oakley, Mark Palatucci, Vaughan Pratt, Pascal Stang, Sven Strohband, Cedric Dupont, Lars-Erik Jendrossek, Christian Koelen, Charles Markey, Carlo Rummel, Joe van Niekerk, Eric Jensen, Philippe Alessandrini, Gary Bradski, Bob Davies, Scott Ettinger, Adrian Kaehler, Ara Nefian, and Pamela Mahoney. Stanley: The robot that won the DARPA Grand Challenge: Research Articles. *J. Robot. Syst.*, 23(9):661–692, September 2006.
- [Tuk77] John W. Tukey. *Exploratory Data Analysis*. Addison-Wesley, 1977.
- [Tul72] Endel Tulving. Episodic and Semantic Memory. In Endel Tulving and W Donaldson, editors, *Organization of memory*, pages 381–403. Academic Press, New York, 1972.
- [VC99] Anca I. Vermesan and Frans Coenen, editors. *Validation and Verification of Knowledge Based Systems - Theory, Tools and Practice, Collected papers from EUROAV '99, 5th European Symposium on Validation and Verification of Knowledge Based Systems, June 9-11, 1999, Oslo, Norway*. Kluwer, 1999.
- [vHLP07] Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter. *Handbook of Knowledge Representation*. Elsevier Science, San Diego, USA, 2007.
- [VKBKs07] Max Van Kleek, Michael Bernstein, David R. Karger, and mc schraefel. GUI — Phooey!: The Case for Text Input. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology*, UIST '07, pages 193–202, New York, NY, USA, 2007. ACM.
- [vKV03] Mark van den Brand, Paul Klint, and Jurgen Vinju. Term rewriting with Traversal functions. *ACM Trans. Software Engineering and Methodology*, 12:152–190, 2003.
- [VKV⁺06] Max Völkel, Markus Krötzsch, Denny Vrandečić, Heiko Haller, and Rudi Studer. Semantic Wikipedia. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 585–594, New York, NY, USA, 2006. ACM Press.

BIBLIOGRAPHY

- [Voo94] Ellen M. Voorhees. Query Expansion using Lexical-Semantic Relations. In *Proceedings of the 17th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '94, pages 61–69, New York, NY, USA, 1994. Springer-Verlag New York, Inc.
- [VPST05] Denny Vrandečić, H. Sofia Pinto, York Sure, and Christoph Tempich. The DILIGENT Knowledge Processes. *Journal of Knowledge Management*, 9(5):85–96, October 2005.
- [VWD04] Fernanda B. Viégas, Martin Wattenberg, and Kushal Dave. Studying Cooperation and Conflict between Authors with history flow Visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, pages 575–582, New York, NY, USA, 2004. ACM.
- [Wag06] Christian Wagner. Breaking the Knowledge Acquisition Bottleneck through Conversational Knowledge Management. *Information Resources Management Journal*, 19(1):70–83, 2006.
- [WF86] T. Winograd and F. Flores. *Understanding Computers and Cognition: A New Foundation for Design*. Language and Being. Ablex Publishing Corporation, 1986.
- [WF99] Ian H. Witten and Eibe Frank. *Data Mining*. Morgan Kaufmann Publisher, 1999.
- [WG98] Tim A. Wagner and Susan L. Graham. Efficient and Flexible Incremental Parsing. *ACM Transactions on Programming Languages and Systems*, 20:980–1013, 1998.
- [Wil04] David Wile. Lessons learned from real DSL experiments. *Sci. Comput. Program.*, 51(3):265–290, 2004.
- [WKJ⁺01] Patrik Werle, Fredrik Kilander, Martin Jonsson, Peter Lönnqvist, and Carl Gustaf Jansson. A Ubiquitous Service Environment with Active Documents for Teamwork Support. In Gregory D. Abowd, Barry Brumitt, and Steven Shafer, editors, *Ubicomp 2001: Ubiquitous Computing*, volume 2201 of *Lecture Notes in Computer Science*, pages 139–155. Springer Berlin Heidelberg, 2001.
- [WMS02] Michael Wright, Mary Marlino, and Tamara Sumner. Meta-Design of a Community Digital Library. *D-Lib Magazine*, 8(5), 2002.
- [WR95] Thomas C. Wooten and Thomas H. Rowley. Using anthropological interview strategies to enhance knowledge acquisition. *Expert Systems with Applications*, 9(4):469 – 482, 1995. Expert systems in accounting, auditing, and finance.
- [WSG92] J.Brian Woodward, M.L.G. Shaw, and B.R. Gaines. The Cognitive Basis of Knowledge Engineering. In Franz Schmalhofer, Gerhard Strube, and Thomas Wetter, editors, *Contemporary Knowledge Engineering and Cognition*, volume 622 of *Lecture Notes in Computer Science*, pages 194–221. Springer Berlin Heidelberg, 1992.

BIBLIOGRAPHY

- [Wyg89] Robert M. Wygant. CLIPS — A powerful development and delivery expert system tool. *Computers & Industrial Engineering*, 17(1-4):546 – 549, 1989.
- [Zav97] Pamela Zave. Classification of Research Efforts in Requirements Engineering. *ACM Comput. Surv.*, 29(4):315–321, December 1997.