Approximate Assertional Reasoning Over Expressive Ontologies

Zur Erlangung des akademischen Grades eines
Doktors der Wirtschaftswissenschaften

(Dr. rer. pol.)

von der Fakultät für
Wirtschaftswissenschaften
des Karlsruher Instituts für Technologie (KIT)

genehmigte

DISSERTATION

von

Dipl.-Inform. Tuvshintur Tserendorj

Tag der mündlichen Prüfung: 27. Januar 2010

Referent: Prof. Dr. Rudi Studer

Korreferent: Prof. Dr. Detlef Seese

Prüfer: Prof. Dr. Karl-Heinz Waldmann

Karlsruhe 2010

Abstract

In the Semantic Web, ontologies provide the required vocabulary for a meaningful and machine-understandable interpretation of data. Nowadays, expressive ontologies are usually written in the W3C standard language called the Web Ontology Language (OWL). In order to leverage the full power of OWL for emerging Semantic Web applications, ontology reasoning holds a key position by allowing access to implicit knowledge in ontologies. Analyzing the application domain of state-of-the-art OWL reasoning techniques, an important issue to be considered is the problem of scalable assertional reasoning over expressive ontologies with large terminological as well as assertional knowledge – a computationally intractable problem.

In this thesis, we address this issue by means of logical approximation. Subsequently, we provide approximate reasoning methods for scalable assertional reasoning whose computational properties can be established in a well-understood way, namely in terms of soundness and completeness, and whose quality can be analyzed in terms of statistical measurements, namely recall and precision. The basic idea of these approximate reasoning methods is to speed up reasoning by trading off the quality of reasoning results against increased speed.

The various aspects of this thesis span the fields of knowledge compilation, query answering, and resource-bounded reasoning. In the context of knowledge compilation, exploiting the fact that data complexity is polynomial for non-disjunctive datalog, a knowledge compilation technique with different useful approximations has been created, which allows tractable assertional reasoning. The logical properties, such as soundness and completeness, of these approximations have been investigated and a comprehensive evaluation using well-known benchmark ontologies has been conducted.

Regarding query answering, a fast approximate reasoning system for instance retrieval has been developed. A central aspect of this approach is the approximate semantics for computing the approximate extensions of a complex concept expression. Based on this semantics, several algorithms have been developed that allow for scalable instance retrieval. In the context of resource-bounded reasoning, a combination of approximate reasoning algorithms has been examined by providing solid formal foundations for the assessment and comparison of approximate reasoning algorithms.

Finally, an approximate ontology reasoning framework has been devised, which enables scalable assertional reasoning over expressive ontologies by loosely integrating the approximate reasoning solutions developed in this work.

Acknowledgements

Foremost, I would like to express my gratitude to my advisor, Prof. Dr. Rudi Studer for giving me the opportunity to work in a very fruitful and stimulating working atmosphere and to do this research. I am also extremely grateful for his commitment to review this thesis despite his busy schedule.

I would also like to thank Prof. Dr. Detlef Seese without hesitation, accepted the request to serve on the examination committee as second reviewer and provided detailed and valuable technical feedback. Many thanks to Prof. Dr. Karl-Heinz Waldmann and Prof. Dr. Christian Hipp who served on the examination committee as examiner and chairman, respectively.

The research reported herein would not have been possible without my supervisors Prof. Dr. Pascal Hitzler and Dr. Stephan Grimm. I warmly thank Pascal for continuous support over the past years, and Stephan for extensive discussions, insightful comments, and ideas. Hitzler's tolerance combined with brilliance has been a major yardstick for me.

Thanks to the "Semantic Karlsruhe Team" that consists of all my colleagues at the Institute AIFB, the FZI research center and at Ontoprise GmbH, for providing a very fruitful and stimulating working atmosphere – even after work. In particular I thankfully mention Andreas Abecker, Sebastian Rudolph, Joachim Kleb, Markus Krötzsch, Jürgen Bock, Andreas Walter, Jens Wissmann, Mark Hefke and Hans-Jörg Happel for valuable comments and ideas.

My parents living in Mongolia receive my deepest gratitude and love for the many years of support and especially for always believing in me. Last, but not least, I thank my wife Tumee who gave me love and support for realizing this work. I also thank her for accepting the lost time on all the weekends that we could have spent together while I was working on this thesis. And my little son who always reminds me that there is something else to live for...

Karlsruhe, January 2010
Tuvshintur Tserendorj

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter we position our work and outline the contents of the thesis. We explain our motivation, which is to develop approximate solutions for query answering over expressive ontologies with large terminological and assertional knowledges. We list the main results of the thesis.

1.1 Motivation

As a worldwide network of information exchange and business transactions, the World Wide Web (WWW) has drastically changed the availability of electronically accessible information. Today, with growing contents in various languages and fields of knowledge, the WWW contains enormous amounts of data which are consumed by billions of people. However, the web data are not structured in such a way that machines can understand and process them automatically. As a result, locating and accessing information or resources on the Web is very difficult, search engines do not understand the context of the data they are indexing and storing, search results are often unsatisfactory, and require further human efforts. Thus, in the long run, it is extremely difficult to make use of data on the Web without *explicit meaning*.

The Semantic Web is a vision for the future of the Web in which data or information is given explicit meaning. This idea refers to metadata making it easier for machines to automatically process and integrate information available on the Web. In contrast to other initiatives towards metadata, the Semantic Web provides a declarative way to present metadata by means of ontologies. With the use of expressive ontologies and sophisticated reasoning solutions, intelligent search could be performed. Search engines become more effective than they are now, and users can find the precise information they are searching for. For this purpose, the World Wide Web Consortium (W3C) has standardized the Web Ontology Language (OWL).

The logical foundation of OWL is formed by a subset of first-order logic called Description Logic (DL) – a family of knowledge representation. Due to its nature of decidability, Description Logics (DLs) have proved useful in a wide range of applications in computer science regarding knowledge representation formalism.

In order to leverage the full power of OWL for emerging Semantic Web applications, ontology reasoning holds a key position by allowing access to implicit knowledge in ontologies. Ontology reasoning for OWL is supported by DL reasoning services, for which there exist two main reasoning approaches for expressive ontologies. Tableaux-based methods [SSS91] implemented in tools such as Pellet [SPG$^+$07] and Racer [HM01a, HM01b] have been shown to be efficient for complex terminological reasoning. Though expressive DLs are supported, they are limited with respect to their support for large assertional knowledges. In contrast, the reasoning techniques based on reduction to disjunctive datalog as implemented in KAON2 [MS06, Mot06] scale well for large assertional knowledge, while at the same time, they support the rich language fragment DL $\mathcal{SHIQ}$.

Besides these two main reasoning approaches, a common approach to achieving scalability is to consider lightweight language fragments of the OWL language and build tractable reasoning procedures. Examples of these approaches are the EL family of languages [BBL05], DL-Lite [CDL$^+$07b], DLP [GHVD03, Vol04] and OWL-Prime [KOM05]. Supporting lightweight language fragments limited for practical applications, such approaches run counter to standardization efforts, namely the recent proposal to increase the expressive power of the OWL language[1]. Based on the observation of application domains of existing approaches to ontology reasoning and also the research challenge for the DL reasoning field identified in [Hor05], a crucial issue, which remains to be investigated, is the problem of scalable assertional reasoning over expressive ontologies with large terminological and assertional knowledge. From a theoretical point of view, we know that it is impossible to find any tractable algorithm for reasoning over expressive ontologies due to the underlying high computational complexities [Tob01]. From the practical point of view, query answering of the complexities will result in unacceptable performance in large-scale, realistic semantic web applications.

Thus, in emerging time-critical applications of expressive ontologies, non-classical reasoning solutions, like approximate reasoning trading expressive power for efficiency, will become more and more important. Approximate reasoning allows applications of expressive ontologies to deal with their underlying high computational complexities in a controlled and well-understood manner. It is based on controlled alterations of inferences in order to achieve lower reasoning complexities at the expense of unsoundness or incompleteness, but it also allows arriving at correctness estimates or at algorithms which subsequently correct initially given answers if more time is available. In particular, approximate reasoning algorithms can be tractable although the underlying language is not.

[1] `http://www.w3.org/2007/OWL/wiki/OWL_Working_Group` [accessed 2009-04-03]

1.2 Research Questions and Goals

This thesis addresses the scalability problem of assertional reasoning over expressive ontologies with large terminological as well as assertional knowledge. Hence, the overall goal is to provide a better insight into the *effect of approximation* to expressive ontology reasoning and to realize an approximate ontology reasoning framework that enables efficient query answering over existing DL systems of one or more orders of magnitude.

Towards this end, we have analyzed why this is currently problematic, how the inherent computational complexities of expressive ontology languages can be effectively handled by using approximation. This analysis results in a number of hypotheses explaining where practical problems lie. These hypotheses are then used to derive concrete approximate reasoning methods. The following hypothesis captures the main research question of this thesis.

Main Hypothesis: *Using approximation, scalable reasoning over expressive ontologies can be achieved in a controlled and well-understood way.*

To increase the scalability of reasoning over expressive ontologies with large terminological and assertional knowledge, one can use other techniques like numerical approximation and ontology modularization[2]. These approaches however do not directly address the underlying complexities of ontology reasoning problems. Towards achieving scalable ontology reasoning, we are concerned with approximate reasoning solutions which allow us to deal with the high computational complexities and whose characteristics are known in terms of soundness and completeness.

In order to support our main hypothesis, we investigate in this thesis how approximation can be applied in order to realize scalable ontology reasoning. We split our main hypothesis into three subordinate hypotheses, while for each of them an approach how to support the hypothesis is given.

Hypothesis 1: *Scalable reasoning in large and expressive knowledge bases of expressive (intractable) ontology languages can be achieved by compiling them into less expressive ones.*

Compiling expressive knowledge bases into less expressive knowledge bases is well known as *knowledge compilation* method [SK91] that has been successfully used in various symbolic Artificial Intelligence (AI) systems. The key motivation behind knowledge compilation is to push as much of the computational overhead into an off-line phase, and thus improving the behavior of an inference method at runtime.

The first hypothesis postulates that knowledge compilation techniques can be applied to large knowledge bases of computationally intractable ontology languages.

[2] The notion of Ontology Modularization refers to a methodological principle for building ontologies in a modular manner.

Hypothesis 2: *Scalable reasoning in large and expressive knowledge bases of intractable ontology languages can be also achieved by query approximation.*

Instance retrieval is one of the most important reasoning services in many practical application systems based on DLs, however, it is still one of the major bottlenecks in reasoning over expressive ontologies, in particular, if the number of instances as well as the ontology structure becomes large and complex.

The second hypothesis postulates that approximation can be applied to improve reasoning performance for complex query concepts. Based on the idea of breaking complex concept queries down to less complex queries, we are going to support this hypothesis by designing algorithms for instance retrieval of complex query concepts.

Hypothesis 3: *Development of approximate reasoning methods needs well-defined methodological guidelines and tool support to measure correctness and performance.*

To obtain an insight into the practical applicability of approximate methods and to compare and contrast them to competing complete and sound reasoning systems, comprehensive evaluations including performance and a degree of their soundness and completeness are required. This kind of evaluations should be an integral part of developing approximate reasoning methods. However, the evaluation of approximate reasoning methods is considered to be a difficult task, in fact, it is not a priori clear how this should be best done.

To address the issues related to the evaluation of approximate reasoning methods, the difficulty of evaluating approximate reasoning systems requires a methodology to advise developers of approximate reasoning systems and help them in designing and executing benchmarks.

1.3 Overall Approach

To achieve our overall goal stated in the previous section, we focus on the reduction-based reasoning system, such as KAON2 [MS06, Mot06], and develop various approximate methods to improve efficiency in query answering.

The reason for this choice is that KAON2 has been shown to be an efficient reasoner when reasoning over expressive ontologies with large data sets [MS06, Mot06]. It employs sophisticated algorithms to translate OWL DL ontologies into disjunctive datalog programs. Theoretical investigations of this reasoning technique have revealed that data complexity[3] is lower than the complexity of tableaux based algorithms [Tob01]. Tableaux based algorithms are inherently limited by the need to build and maintain a model of the whole ontology including all of the instance data [Hor05, Tob01].

[3] The complexity of answering queries against a fixed ontology and set of instance data.

As indicated by our research questions, in this thesis, various forms of approximation will be analyzed which result in a comprehensive approximate ontology reasoning framework. In the following, we describe this framework in more detail and discuss its extensions and interplays.

1.3.1 Knowledge Compilation Method

Our approximation method indicated in the first hypothesis is designed to optimize assertional reasoning (query answering) with the KAON2 ontology reasoning system[4]. Reasoning with this system proceeds in two main stages. First, the given expressive ontology to be queried is transformed into a (disjunctive) datalog program which may contain disjunctive rules. We call this process TBox-transformation. Second, the resulting program (knowledge base) is used to answer queries. This query answering process is performed using sophisticated deductive database techniques specifically designed to reason with large data sets.

In fact, disjunctive rules in the translated knowledge bases lead to a dramatic increase of the computational costs – an increase from polynomial complexity to NP-hardness, *i.e.,* a problem which probably requires exponential time to be solved. Thus, assertional reasoning remains NP-hard, and thus intractable. To lower this high computational complexity and achieve polynomial data complexity, knowledge compilation can be applied by treating disjunctive rules as if they were non-disjunctive ones.

In order to support the first hypothesis, we develop a knowledge compilation method with various approximations for replacing disjunctive rules and thereby show that efficiency of assertional reasoning over expressive ontologies can be improved.

1.3.2 Query Approximation

The idea of our query approximation method – to support the second hypothesis – deploys the fact that instance retrieval for atomic concepts requires significantly less effort than that for complex ones. We closely look at the complexity results on DLs. The worst case time complexity of tableaux algorithms for the most basic reasoning problem to which instance retrieval is reduced is NExpTime [Tob00]. The worst case time complexity of query answering with the KAON2 approach including TBox-transformation is ExpTime [Mot06]. In case the TBox-transformation is performed in an offline phase, the worst case time complexity of query answering with KAON2 is NP; we deal with lowering this complexity in the method supporting the first hypothesis. In case of complex queries, we cannot push the TBox-translation into an offline phase, because KAON2 needs to perform it repeatly [Mot06].

With the development of an approximate instance retrieval method for complex queries, we are concerned with the question, how to lower the worst case time complexity of instance retrieval for complex queries. In this regard, we introduce the

[4] `www.semanticweb.kaon2.org` [accessed 2009-06-21]

novel notion of the approximate extension[5] of DL concepts. Using this notion, we present an approximate semantics for DL complex concepts based on the model-theoretic semantics of DLs. Based on the resulting approximate semantics, we are going to design and implement an approximate instance retrieval method and show that a significant performance improvement in instance retrieval can be achieved.

1.3.3 Anytime Reasoning and Combination

In many cases, a satisfying answer, which falls within the range of error tolerance and which is available at a certain point in time, is preferred to correct and best answer requiring arbitrary long time. Anytime algorithms are algorithms designed to conform to this practical requirement. They gradually improve the quality of their results, as computation time increases, and end with providing the whole answer when complete computation is required. To this end, we are developing anytime algorithms by combining our approximate methods. Such a combination can be realized either by combining only the approximate algorithms or by combining them with a sound and complete reasoner. Thereby, our approximate framework includes anytime reasoners that generates suboptimal answers at a certain point in time and acts a complete and sound reasoner when time allowed.

1.3.4 Methodology on Benchmarking Approximate Reasoning

Evaluation of approximate reasoning methods is considered to be a difficult task and different from that of complete and sound DL reasoners. The difficulty is that we do not only want to measure execution time, but we also need to determine empirically to what extent the evaluated algorithms are sound and complete in terms of precision and recall. In order to do this, one needs to create and test a suitable and large enough sample of queries. However, it is not a priori clear what kind of queries should be considered. Even when there are enough criteria to generate test queries, our experience shows that it is non-trivial to conduct the measurements. That is because we have to test a DL reasoner with reasoning-intensive queries in order to determine precision and recall for each query. Furthermore, it often involves several iterative rounds in order to attain predictable, useful conclusions. Interpretation of benchmarking data is also extraordinarily difficult.

In order to support the third hypothesis, we are going to develop an automated benchmarking framework.

[5] Set of individuals belonging to a DL concept.

1.4 Contributions

After having outlined scalable ontology reasoning techniques and discussed theories for supporting them, we determine the scope, highlight the scientific contribution, and introduce the structure of this thesis.

With the comprehensive treatment of the question what can be done about the difficulty of scalable reasoning over expressive ontologies, the main contribution of this work is the design and realization of an approximate ontology reasoning framework. Within this comprehensive framework a number of individual contributions were made.

A knowledge compilation technique with different useful approximations was conceived. Soundness and completeness of these approximations were shown. A prototypical proof-of-concept implementation was realized and evaluated on well-known benchmarking ontologies.

In the context of query answering, a novel notion for approximating DL concepts was introduced. Using this notion, an approximate semantics for DL concepts was defined. Based on the resulting approximate semantics, an approximate instance retrieval reasoner specially designed for querying complex concepts was conceived, implemented and evaluated.

In the context of benchmarking, methodological guidelines were developed to advise the testers of approximate ontology reasoning systems. Furthermore, supporting these guidelines, the first benchmarking framework for approximate reasoning systems in the Semantic Web was designed and implemented. The evaluation of this framework showed its suitability for end users.

In the context of anytime reasoning, we devised a novel anytime reasoning framework for instance retrieval that works on top of our approximate reasoners as well as existing ontology reasoners. The framework is integrated into the Protégé[6] ontology engineering environment.

In a further development, our approximate solutions were integrated into a unified approximate ontology reasoning framework that can be used either programatically via both popular ontology management APIs such as KAON2 and OWL API or standalone. Finally, it was systematically analyzed and demonstrated what can be done using approximation for coping with the high computational complexities of assertional reasoning problems.

1.5 Thesis outline

Before we move on in the thesis, this section gives the reader a brief overview of the entire work. The present work consists of four main parts, bracketed by twelve chapters and enhanced by two appendices and a bibliography.

[6] `http://protege.stanford.edu/` [accessed 2009-07-25]

Part I – Foundations — The first part lays the foundations for the thesis, presenting preliminaries in the fields of Description Logics, ontologies and the Semantic Web. Chapter 2 introduces the notion of ontologies and their role in the Semantic Web vision. It provides an overview on the ontology language OWL. Chapter 3 introduces Description Logics formally. Chapter 4 presents elementary fundamentals of logic programming, relational databases and deductive databases.

Part II – Approximate Reasoning in the Semantic Web — The second part concerns various forms of approximate reasoning methods for query answering as the main theme of the thesis. Chapter 5 provides a foundation of approximate reasoning which shall clarify the essentials of approximate reasoning. It contains a discussion of benchmarking issues relevant to the evaluation of approximate reasoning methods and motivates the need for methodological guidance and tool support.

Chapter 6 presents the concept of the knowledge compilation method SCREECH which allows for various Horn transformations of disjunctive datalog. It describes details of the several variants of the SCREECH approach as well as their soundness and completeness characteristics. Finally, we report on a comprehensive experimental analysis conducted with well-known benchmarking ontologies. The chapter contains an overview of the general concept of addressing the scalability of reasoning in the context of knowledge compilation.

Chapter 7 presents an approximate method for instance retrieval of complex concept queries. Moreover, it introduces the notion of approximate concept extension and presents several approximate algorithms to compute approximate extension of complex concept queries. It also contains a discussion of several optimizations for these algorithms. Finally, a comprehensive empirical analysis of the algorithms reporting positive results will be given.

Chapter 8 presents several combinations of the approximate reasoning methods introduced in Chapters 6 and 7 which constitute to the development of anytime reasoning algorithms.

Part III – Implementations and Applications — This part presents the implementation of the methods introduced in Chapters 6 through 8 and their applications. It contains a discussion of the application of approximate reasoning techniques.

Chapter 9 presents the implementation of the SCREECH knowledge compilation method, the approximate instance retrieval method, the composed anytime reasoning algorithms, and the benchmarking framework. Furthermore, it presents the approximate ontology reasoning framework in which all the components above are loosely integrated.

Chapter 10 discusses how the approximate ontology reasoning framework is used in the THESEUS[7] research project.

[7] THESEUS is a research project initiated by the Federal Ministry of Economy and Technology, http://theseus-programm.de.

Part IV – Finale — In the final part, we conclude this thesis and discuss the results achieved. Furthermore, for future work, we sketch the possible research directions and extensions of the approximate ontology reasoning framework.

Part V – Appendices — The first appendix gives a full description of the data structures used to describe the approximate reasoning algorithms developed in this work. The second appendix presents the detailed experimental results.

1.6 Publications

Most of the works presented in this thesis have been previously published in proceedings of international conferences and workshops. This work is the result of fruitful cooperations in the German research project THESEUS.

In the following list, we provide references to relevant publications for individual chapters of this thesis.

The work forming the foundations for the assessment and comparison of approximate reasoning algorithms presented in Chapter 5 was published in [RTH08].

The knowledge compilation method SCREECH described in Chapter 6 and its approximation strategies were published in [TRKH08]. The results of the experiments conducted with SCREECH were published in [RTH08, TRKH08].

The AQA approach described in Chapter 7 is under submission in the Semantic Web community and made available in the technical report [TGH08].

The concept of the composed anytime reasoning algorithms from Chapters 5 was published in [RTH08], and their implementation and integration into the reasoning broker system HERAKLES presented in Chapter 10 were published in [BTX+09b, BTX+09a].

Part I

Foundations

Chapter 2

The Semantic Web and Ontologies

Ontologies, as used in information systems, are conceptual yet computational models of a domain of interest that build on techniques of knowledge representation. They play a key role in the Semantic Web, where they support the meaningful annotation of web content and resources.

This chapter gives a brief introduction to ontologies and the Semantic Web to lay the ground for approximate reasoning over expressive ontologies. Section 2.1 presents the vision of the Semantic Web and the notion of ontologies. Section 2.2 provides an overview on the ontology language OWL. A summary of other ontology formalisms is given in Section 2.3.

2.1 The Semantic Web Vision

The Web can be seen as a success story, both in terms of the amount of available information and the number of people using it. The Web's success is owed to the simplicity of the underlying structures and protocols that ensure easy access to all kinds of resources, i.e. it is characterized by easy access to a huge amount of information. The Internet helps its users to deal with documents, though unfortunately not with the accompanying information. As a result of the continuous growth of the Internet, the search for, location, organization, and maintenance of information and knowledge (not documents) has become difficult and complex. Web pages are developed and designed by people for people, and the computer, as a machine, can only present information understandable to humans. On the other hand, existing technologies used by common search engines are usually incapable of serving the expectations of the users, delivering mismatched, irrelevant, or insufficient results, or are unable to deliver the hoped-for responses to more complex queries.

In contrast, the Semantic Web is an evolving extension of the current Web in which Web content can be expressed not only in natural language, but also in a form that can be read and "understood" by software agents, thus permitting them to find, share, and integrate information more easily. The term Semantic Web, which was coined by the inventor of the web, Sir Tim Berners-Lee, in [BLHL01], stands for the idea of a future Web which aims to increase machine support for the interpretation and integration of information on the Web. The World Wide Web Consortium (W3C), which is the standardization body responsible for the Web, defines the term Semantic Web in its ″Semantic Web Activity Statement“ [BLHL01] as follows:

> *The Semantic Web is an extension of the current web in which information is given a well-defined meaning, better enabling computers and people to work in cooperation.*

To achieve the vision of the Semantic Web, networked resources, *e.g.*, websites or web services, are annotated by structured and machine-understandable metadata, which are assigned a well-defined meaning and are interpreted by means of ontologies.

Different Views on Ontologies

The term ″Ontology“ (Greek. onto = *being*, logos = *to reason*) was first use by Aristotle to describe ″the science of being *qua* being“ [Ari08]. *Categories of being* explain and classify everything what exists. In the following, we will introduce the main definitions of the term ″ontology“ whose meanings depend on the domain in which the ontology is to be applied. We also highlight the usage of ontology in the context of our research within Computer Science.

Philosophical Roots

According to [Flo03], ontology as a branch of philosophy is the science which defines the kinds and structures of objects, properties, processes, relations, etc. in every area of reality. According to the definition in the “Encyclopedia Britannica”[1], an ontology is the theory or study of being as such; it is an area of philosophy that deals with the nature and organization of reality. Traditionally, issues on existence and the state of being were answered by metaphysics, a discipline which goes back to Aristotle and refers to fourteen treatises dealing with what he called “first philosophy”. Philosophical ontology looks for the description (not explanation) of the terms of a classification of entities in the universe and can also be called descriptive or realist ontology. Philosophical ontology describes the categories available inside a given domain of interest and can be unitized into a formal ontology, which is a formal theory of non domain-specific entities, attributes, items, or domains, and a material ontology, also called regional ontology, which is concerned with domain-specific terms, concepts etc. Generally, the philosopher-ontologist attempts to establish the truth about reality by finding an answer to the question of existence.

[1] `http://original.britannica.com` [accessed 2009-6-27]

Ontologies in Computer Science

The term ontology has been introduced to Computer Science as a means to formalize the kinds of things that can be talked about in a system or a context. In the world of information systems, an ontology is a software or formal language artefact designed with a specific set of uses and computational environments in mind. In this context, ontologies aim at capturing domain knowledge in a generic way and provide a commonly agreed understanding of a domain, which may be reused and shared across applications and groups [CJB99]. Ontology is (very largely) qualitative and deals with relations, including the relations between entities belonging to distinct domains of science, as well as between such entities and the entities recognized by common sense. An ontology is specified by a specific client within a specific context and in relation to specific practical needs and resources. With applications in fields such as knowledge management, information retrieval, natural language processing, information integration, and the Semantic Web, ontologies are part of a new approach to building intelligent information systems [Fen03]. They are intended to provide knowledge engineers with reusable pieces of declarative knowledge, which can be – together with problem-solving methods and reasoning services – easily assembled into high quality and cost-effective systems [NFF^{+}91]. A definition of ontology that is much referenced in the literature is: ″An ontology is an explicit specification of a conceptualization“ [Gru93].

A conceptualization refers to the way knowledge is represented. It is encoded in an abstract manner using concepts and relations between concepts. Abstractness refers to the fact that ontologies try to cover as many situations as possible, instead of focusing on particular individuals [Gua98]. An *explicit specification* refers to the fact that the concepts and the constraints on their use are explicitly defined in an ontology and thus accessible for machines. This definition is extended by requiring a ″formal specification“ and a ″shared conceptualization“ [Bor06]. In this context, *formality* refers to the type of knowledge representation language used for specifying the ontology. This language has to provide formal semantics in a sense that the domain knowledge can be interpreted by machines in an unambiguous and well-defined way.

In addition, the vocabulary formally defined by this language should represent a consensus between the members of a community. By committing to such a common ontology, community members (or more precisely their software agents) can make assertions or ask queries that are understood by the other members. Finally, an ontology always covers knowledge about a certain ″domain of interest“. Therefore, many applications use a set of ontology modules that model different aspects of the application.

In recent years, ontologies became an important technology for knowledge sharing in distributed, heterogeneous environments, particularly in the context of the Semantic Web. Relying on the defintions above, Studer et al. [SBF98] give the following definition which is predominantly used within the Semantic Web community.

Definition 2.1 (Ontology). *An ontology is a formal explicit specification of a shared conceptualization of a domain of interest.*

Ontologies are targeted to provide the means to formally specify a commonly agreed upon understanding of a domain of interest in terms of concepts, relationships, and axioms, as well as a common vocabulary of an area, and to define – with different levels of formality – the meaning of the terms and of the relations between them [Gru93]. While, according to [Hep08], in computer science, researchers assume that they can define the conceptual entities in ontologies mainly by formal means, in information systems, researchers discussing ontologies are more concerned with understanding conceptual elements and their relationships, and often specify their ontologies using only informal means.

One of the central ideas behind ontologies is the possibility of reusing existing ontologies and thus reducing the modeling effort. However, it has turned out that different types of ontologies are more suitable for reuse than others. In this context, the generality of the ontologies is important: general ontologies can be reused in many different contexts, whereas very specific ontologies are rarely reused. Thus, the following categorization of ontologies can be applied [Gua97]:

Generic Ontology (core or upper ontology): Generic ontologies describe general concepts, such as object, event, action, etc. that may be present or occur in many (or even all) different domains and applications. Therefore, these concepts are independent from a concrete usage scenario and can be shared by a large community of users. Generic ontologies are also often called *foundational*, *top-level* or *upper level ontologies*. Since they are easily reused, it is worth devoting effort into building philosophically sound and highly axiomatized top-level ontologies, which unambiguously describe the vocabulary. A prominent example of generic ontologies is DOLCE[2] [MBG$^+$02].

Domain ontology: A domain ontology models a specific domain, or part of the world. It represents the particular meanings of terms as they apply to that domain. For example, the word "golf" has many different meanings. An ontology about the domain of automobiles would model the "kind of car" meaning of the word, an ontology about the domain of sports would model the "kind of game" meaning to the word, while an ontology about the domain of geography would model the "geographical location" meanings. An example of domain ontologies is the "Wine ontology"[3] which is about the most appropriate combination of wine and meals.

Application Ontology: These ontologies contain all the necessary knowledge for modeling a particular domain (usually a combination of domain and method ontologies that provide terms specific to a particular problem-solving method).

[2] `http://www.loa-cnr.it/DOLCE.html` [accessed 2009-5-11]

[3] `http://www.w3.org/TR/owl-guide/wine.rdf` [accessed 2009-07-22]

Usually they can be rarely reused for other application contexts (see *e.g.,* [SBF98]). An example of application ontologies is the CCO ontology[4] which presents the cell cycle knowledge by extending existing ontologies.

Representational ontology: These ontologies do not commit to any particular domain. Such ontologies provide representational entities without stating what specifically should be represented. A well-known representational ontology is the *frame ontology*, which defines concepts such as frames, slots, and slot constraints that allow the expression of knowledge in an object-oriented or frame-based way [Gru93].

Task Ontology: A task ontology provides terms specific to particular tasks (*e.g.,* hypotheses belong to the diagnosis task ontology).

Note that other categorization dimensions have been proposed in literature. For example, ontologies can be classified according to the level of formality or with respect to the ontology language used.

2.2 Ontology Languages for the Semantic Web

Ontologies are a pillar of the Semantic Web. They are used to capture background knowledge about some domain of interest by providing relevant concepts and relations. Moreover, their role is to provide intensional knowledge in a machine processable way, and thus enable automatic aggregation and the proactive use of web resources. This section introduces the ontology languages that are used for representing and querying knowledge within the Semantic Web. such as the Resource Description Framework (RDF), its extension RDFS and the Web Ontology Language (OWL).

2.2.1 Resource Description Framework (RDF)

RDF [KC04, MM04] allows for the description of resources and how they relate to each other. RDF specifies a data model for publishing metadata as well as data on the Web and utilizes XML as serialization syntax for data transmission. It provides a consistent, standardized way of describing and querying Web resources, from text pages and graphics to audio files and video clips. It is a model and syntax for annotating web resources designed for the exchange of information over the Web. RDF provides (with the other standards like RDFS and OWL) syntactic interoperability between applications on the Web as well as a base layer for building the Semantic Web.

The underlying structure of any expression in RDF is a collection of triples, each consisting of a subject, a predicate and an object. A set of such triples is called an RDF

[4] `http://www.cellcycleontology.org/` [accessed 2009-11-29]

graph. The assertion of an RDF triple says that some relationship, indicated by the predicate, holds between the things denoted by subject and object of the triple. The assertion of an RDF graph amounts to asserting all the triples in it, so the meaning of an RDF graph is the conjunction (logical AND) of the statements corresponding to all the triples it contains. Ontologies written in RDF are not in the focus of this thesis, so the interested readers might refer to for a formal account of the meaning of RDF graphs [Hay04, MM04].

The RDF vocabulary description language (RDF Schema [RDFS]) [BG04] defines a simple modeling language on top of RDF. RDFS is intended to provide the primitives that are required to describe the vocabulary used in particular RDF models. This description is achieved by expressing set membership of objects in property and class extensions. Therefore RDFS uses classes, subsumption relationships on both classes and properties, and global domain and range restrictions for properties as modeling primitives. However, RDFS is too weak to describe resources in sufficient detail. Moreover, it does not allow to define localised range and domain constraints, existence/cardinality constraints and more expressive properties such as transitive, inverse and symmetrical.

2.2.2 Web Ontology Language (OWL)

It is desirable in the Semantic Web to extend RDFS with a well-designed, well-defined, and web-compatible ontology language with supporting reasoning tools. The syntax of this language should be both intuitive to human users and compatible with existing web standards such as XML, RDF, and RDFS. Its semantics should be formally specified since otherwise it could not be machine-interpretable.

To come forward with such a language, the Web Ontology Working Group of W3C defined OWL, the language that is standardized and broadly accepted ontology language of the Semantic Web [PSHH03, HEPS03]. Historically, OWL emerged from several former knowledge representation and description languages, like SHOE [Hef01] and DAML+OIL [MFHS02].

OWL facilitates greater machine interpretability of Web content than that supported by XML(S) and RDFS by providing additional vocabulary along with a formal semantics. It has a richer set of operators – *e.g.*, intersection, union and negation. Complex concepts, also called classes, can therefore be built up in definitions out of simpler concepts by using such logical operators. Furthermore, its formal semantics allows the use of a reasoner which can check whether or not all of the statements and definitions in the ontology are mutually consistent.

Several subsets (species) of OWL were defined in the standard to accommodate various interest groups and allow to safely ignore language features that are not needed for certain applications. In OWL 1.0, there are three species such as OWL-Lite, OWL-DL, and OWL-Full. Table 2.1 describes these species. OWL-DL is much more expressive than OWL-Lite and is based on Description Logics that are a decidable fragment of First Order Logic and are therefore amenable to automated reasoning.

OWL-Lite	OWL-Lite is the smallest standardized subset. In OWL-Lite, there are limitations on how a class can be asserted and the restrictions that can be placed on a class. Hence, it is mainly to support the design of classification hierarchies and simple constraints.
OWL-DL	OWL-DL allows full use of the core OWL language, but with some limitations on class restrictions. Those limitations in contrast to OWL-Full ease the development of tools and allow complete inference. For OWL-DL, practical reasoning algorithms are known, and increasingly more tools support this or slightly less expressive languages.
OWL-Full	OWL-Full is the most expressive of the three, allowing expressions of higher order predicate that classes can be also properties and instances.

Table 2.1: OWL 1.0 species

Note that we will give a comprehensive detail of the description logic $\mathcal{SHOIN}$ which forms the formal foundations of OWL-DL in Section 3.2, Chapter 3.

Contrary to OWL-Lite and OWL-DL which impose restrictions on the use of the vocabulary, OWL-Full supports the full syntactic freedom of RDF. Moreover, OWL-Full is intended to be used in situations where very high expressiveness is more important than being able to guarantee the decidability or computational completeness of the language. It is therefore not possible to perform automated reasoning on OWL-Full ontologies.

The syntax of the OWL language is layered on that of the RDF language. Therefore, the official syntax of OWL is the syntax of the RDF language. However, OWL extends RDF with additional vocabulary. In the following, we will describe the basic elements of OWL such as classes, properties, and individuals in detail.

Class Description

Classes are the basic building blocks of an OWL ontology and defined using `owl:Class` element. There are two kinds of classes to represent concepts in a domain: *defined classes* and *primitive class*. A class is a defined class if it has at least one set of necessary and sufficient conditions. Such classes have a definition, and any individual that satisfies the definition will belong to the class. Classes that do not have any sets of necessary and sufficient conditions (only have necessary conditions) are known as primitive classes or named classes.

OWL comes with two predefined classes: `owl:Thing` and `owl:Nothing`. Each ontology has the unique class `owl:Thing` that is the class that represents the set containing all individuals. Since OWL classes are interpreted as sets of individuals, all classes are subclasses of `owl:Thing`. `owl:Nothing` is an empty class, and thus every class is a superclass of `owl:Nothing`. OWL supports six main ways of describing classes. The simplest of these is a named class. The other types are: intersection classes, union classes, complement classes, restrictions, enumerated classes. Intersection classes are formed by combining two or more classes with the intersection (AND) operator. Union classes are formed using the union (OR) operator with two or more classes. A complement class is specified by negating another class. It will contain the individuals that are not in the negated class.

Relationship Description

A property is a binary relation that specifies class characteristics. They are attributes of instances and sometimes act as data values or link to other instances. In OWL there are two kinds of properties: *Object properties* and *Datatype properties*. Object properties relate individuals to other individuals. Object properties are expressed using `owl:ObjectProperty`. Datatype properties relate individuals to datatype values, such as integers, floats, and strings. OWL makes use of XML Schema for defining datatypes. Datatype properties are expressed using `owl:DatatypeProperty`. A property can have a domain and range associated with it. Each property can be put into one of the following categories:

- *Functional*: For a given object, the property takes only one value. Examples include a person's age, height, or weight. Functional properties are expressed using `owl:FunctionalProperty`.

- *Inverse functional*: The property defines that two different individuals cannot have the same value. For example, the bankNumber or SSN properties are unique for each person. Inverse functional properties are expressed using `owl:InverseFunctionalProperty`.

- *Symmetric*: The property defines a symmetric property such that if a property links A to B, then one can infer that it links B to A. Examples of symmetric properties includes "is sibling of" or "is same as". Symmetric properties are expressed using `owl:SymmetricProperty`.

- *Transitive*: The property defines a transitive property, such as if a property links A to B and B to C, then one can infer that it links A to C. For example, if A is taller than B and B is taller than C, then A is taller than C. Transitive properties are expressed using `owl:TransitiveProperty`.

Furthermore, OWL allows one to apply various restrictions to classes and properties. Restrictions describe a class of individuals based on the type and possibly number of relationships that they participate in. Restrictions can be grouped into three main categories: quantifier restrictions such as existential and universal, cardinality restrictions, has-value restriction. The existential restriction means ″some values from", or at least one. An existential restriction describes the class of individuals that have at least one kind of relationship along a specified property to an individual that is a member of a specified class. Cardinality restrictions allow us to talk about the number of relationships that a class of individuals participate in. Has-value restrictions specifies that class of individuals that participate in a specified relationship with a specific individual. An enumeration class is specified by explicitly and exhaustively listing the individuals that are members of the enumeration class. To specify an enumerated class, the individuals that are members of the class are listed inside curly brackets.

Individuals

Individuals are instances of classes, and properties can relate one individual to another. For example, one might describe an individual named *Smith* as an instance of the class PERSON, and might use the property hasEmployer to relate *Smith* to the individual *Webify Solutions*, signifying that *Smith* is an employee of *Webify Solutions*.

The Next Step for OWL

OWL 2[5] is the recent extension to and revision of the OWL language. It extends the OWL language with a small but useful set of features that have been requested by vendors and implementers. The new features include increased expressive power for properties, extended support for datatypes, simple metamodeling capabilities, extended annotation capabilities, and keys. OWL 2 also defines several profiles – OWL 2 language subsets that may better meet certain performance requirements or may be easier to implement.

The interested reader may refer to [GHM$^+$08, MPSG09] for additional material on the recent proposal of OWL 2. For this work, OWL-DL would be most useful as it is the OWL dialect which is mostly used in practice as it is decidable although still very expressive. For a comprehensive overview of the current status and future prospectives of the field of ontologies, the interested reader might refer to [SS09].

2.3 Other Ontology Formalisms and Tools

A expressive formalism for representing formal ontologies in the Semantic Web is Frame Logic (F-Logic), that is a deductive, object-oriented and frame-based language. Originally, the language was developed for deductive and object-oriented databases. Later on, however, it has been applied for the implementation of ontologies. F-Logic provides constructs for defining declarative rules which infer new information from the available information. Furthermore, queries can be asked that directly use parts of the ontology. From a syntactic point of view, F-Logic is a superset of first-order

[5] http://www.w3.org/2007/OWL/wiki/Document_Overview [accessed 2009-07-22]

logic. Currently a new version of FLogic is defined as a community process taking current developments from the semantic web area into account. Readers interested in a more thorough treatment of F-logic might refer to [KLW95].

API and Software Support

There are a number of ontology management tools and APIs. Several of them, in particular, such APIs with reasoning capabilities and some DL reasoning systems are briefly introduced in the following.

OWL API

The OWL API [BVL03] (the current version OWL 1.1 [HBN07]) is designed to facilitate the manipulation of OWL 1.1 ontologies at a high level of abstraction for use by editors, reasoners and other tools. The API is based on the OWL 1.1 specification. Furthermore, it provides general reasoner interfaces to describe different reasoner functionality, ranging from consistency checkers through to class based reasoning. A salient feature of the OWL API is to subscribe to the axiom centric view of an ontology.

Jena

Jena [McB01] is a Java framework for building Semantic Web applications. The latest version of Jena provides a programmatic environment for RDF, RDFS and OWL[6], including a rule-based inference engine. Jena grew out of work in the HP Labs[7] Semantic Web Programme. The Jena framework includes an RDF API and an OWL API. Jena takes an RDF-centric view, which treats RDF triples as the core of RDFS and OWL. Among other things, it includes several rule engines.

Pellet

For the time being, Pellet[8] [SPG$^+$07] is the only reasoner fully conforming OWL 2 DL, currently in version 2.0 RC7. Its implementation is based on the tableau decision procedure, however, it also implements special support for the OWL 2 EL profile. Pellet is tightly integrated with the OWL API and implements all of its reasoning methods.

FaCT++

Available in its version 1.3.0, FaCT++[9] [TH06] is a reasoner fully conforming OWL 2 DL except for some datatypes. It implements optimised tableau decision pro-

[6] `http://jena.sourceforge.net` [accessed 2009-06-20]

[7] `http://www.hp.com` [accessed 2009-06-20]

[8] `http://clarkparsia.com/pellet` [accessed 2009-06-20]

[9] `http://owl.man.ac.uk/factplusplus` [accessed 2009-06-20]

cedures. Since FaCT++ is implemented in C++, it needs JNI[10] to be accessible from the Java™ based OWL API. However, not all of the OWL API's reasoning methods have been implemented for FaCT++.

RacerPro

The RacerPro system [HM01b] is an optimized tableau reasoner for SHIQ(D). For concrete domains, it supports integers and real numbers, as well as various polynomial equations over those, and strings with equality checks. It can handle several TBoxes and several ABoxes and treats individuals under the unique name assumption. Besides basic reasoning tasks, such as satisfiability and subsumption, it offers ABox querying based on the nRQL optimizations. It is implemented in the Common Lisp programming language. Recently, RacerPro has been turned into the commercial (free trials and research licenses available) RacerPro system [11].

KAON2

KAON2 [Mot06] is an infrastructure[12] for managing OWL-DL ontologies. It also supports Semantic Web Rule language (SWRL)[13] and F-Logic. KAON2 provides a built-in reasoner for the SHIQ(D) subset of OWL-DL. This includes all features of OWL-DL apart from nominals (also known as enumerated classes). KAON2 also supports the so-called DL-safe subset of SWRL to achieve decidable reasoning [MSS04]. Reasoning is based on algorithms, which reduce an OWL ontology to a (disjunctive) datalog program. Its performance with large ABoxes compares favourably with other state-of-the-art OWL DL reasoners [Mot06].

We have discussed the APIs for OWL ontologies that are widely used within the Semantic Web community. Reasoners, such as FaCT ++ and Pellet, are the reasoners that implement the tableaux-based reasoning algorithms, while KAON2 applies disjunctive datalog techniques, especially for providing scalable assertional reasoning.

[10] http://en.wikipedia.org/wiki/JNI [accessed 2009-06-20]

[11] http://www.RacerPro-systems.com/ [accessed 2009-11-29]

[12] http://kaon2.semanticweb.org [accessed 2009-06-20]

[13] It extends OWL with Horn-like rules that are interpreted according to first-order semantics.

Chapter 3

Description Logics

This chapter introduces the basic concepts and notations of description logics that are relevant for this thesis. The fundamentals of description logics are mainly taken from [BCM$^+$03]. Starting from the simplest Description logic $\mathcal{AL}$, Section 3.1 presents the basic terminology of description logics. The formal syntax and semantics of a more expressive language $\mathcal{SHOIN}$ is presented in Section 3.2. Standard inference (reasoning) services in DLs are described in Section 3.3.

3.1 Basic Notions

Description Logics[1] (DLs) [BCM$^+$03] have initially been designed to fit object-centric knowledge representation formalisms like semantic networks and frame systems with a formal and declarative semantics. During the 25 years of research in this field of knowledge representation a family of logics has evolved, which can be distinguished from each other by the constructors and axioms available in each language. Generally, the particular selection of constructors and axioms is made such that inferencing with the logic is decidable.

In DLs, the terminological knowledge of an application domain is represented in terms of *concepts* (unary predicates) such as HUMAN and WOMAN, and *roles* (binary predicates) such as hasChild. Concepts denote sets of individuals and roles denote binary relations between individuals. Based on basic concept and role names, complex concept descriptions are built inductively using concept constructors.

Typically, DLs provide the boolean concept constructors, namely conjunction ($\sqcup$), disjunction ($\sqcap$), and negation ($\neg$); DLs usually support constructors to restrict the quantification of roles, specifically universal ($\forall$) and existential ($\exists$) restrictions. Furthermore, complex concept descriptions are built inductively using concept constructors.

The language $\mathcal{AL}$ is a minimal attributive language [BCM$^+$03] in which complex concept descriptions are built according the syntax rule show in Table 3.1. In that

[1] `http://dl.kr.org` [accessed 2009-06-20]

C, D	$\longrightarrow$	A	atomic concept
		$\top$	universal (top) concept
		$\bot$	bottom concept
		$\neg\, \mathsf{A}$	atomic negation
		$\mathsf{C} \sqcap \mathsf{D}$	intersection
		$\forall R.C$	value restriction
		$\exists R.\top$	limited existential quantification

Table 3.1: Syntax rule in the language $\mathcal{AL}$

abstract notation, A and B denote atomic concepts, R denotes an atomic role, and C and D denote concept descriptions. Hence, the following $\mathcal{AL}$-concept description represents all women that have at least one human child, *i.e.,* who are a mother:

$$\text{WOMAN} \sqcap \exists \mathsf{hasChild}.\text{HUMAN}.$$

Additional constructors including cardinality restrictions on roles and more expressive roles (*e.g.,* inverse and transitive roles) are provided in some DLs. The expressive power of a description logic depends on the provided constructors from which concepts and relations can be composed, and the kinds of axiom supported.

Symbol	Available Constructs
$\mathcal{AL}$	conjunction, universal value restriction and limited existential quantification
$\mathcal{C}$	disjunct and full existential quantification with full negation
$\mathcal{R}+$	transitive role
$\mathcal{S}$	shortcut for $\mathcal{ALC}_{R+}$
$\mathcal{H}$	role hierarchy
$\mathcal{I}$	inverse role
$\mathcal{F}$	functional role
$\mathcal{O}$	nominals, i.e. enumeration of classes or data values
$\mathcal{Q}$	qualified number restrictions
$\mathcal{N}$	unqualified number restrictions
D	concrete domains

Table 3.2: Description logic variants [BCM^{+}03]

Table 3.1 gives a short overview of the constructs available in a particular description logic. The first column indicates an informal naming convention, roughly describing the constructors allowed.

As an example, $\mathcal{ALC}$ is a centrally important description logic from which comparisons with other varieties can be made. $\mathcal{ALC}$ extends $\mathcal{AL}$ by general concept negation ($\neg$C) and unlimited existential restriction ($\exists R.C$), and is thus the most basic DL closed under Boolean operators. A further example, the description logic $\mathcal{SHIQ}$ is the logic $\mathcal{ALC}$ plus extended cardinality restrictions, and transitive and inverse roles. Note that the naming conventions are not purely systematic so that the logic $\mathcal{ALCOIN}$ might be referred to as $\mathcal{ALCNIO}$ and abbreviations are made where possible, $\mathcal{ALC}$ is used instead of the equivalent $\mathcal{ALUE}$.

DL Knowledge Base

A description logic knowledge base usually consists of a set of axioms, which can be distinguished into terminological axioms and assertional axioms. The terminological axioms are called TBox and it introduces the terminology, *i.e.,* the vocabulary of an application domain. The assertional axioms are called ABox and it contains assertions about named individuals in terms of the vocabulary.

Terminologies (TBox)

The TBox contains intensional knowledge (axioms about concepts) in the form of a terminology. Terminological axioms are introduced, which make statements about how concepts or roles are related to each other. Terminologies are composed of terminological axioms which can be *definitions* and *inclusion assertions*.

The axioms in the TBox can be built using the previously mentioned concept constructors, as well as concept inclusion axioms ($\sqsubseteq$), which state inclusion relations between DL concepts. For example, one can state that any technology company is a company via the following axiom: $\textsc{TechnologyCompany} \sqsubseteq \textsc{Company}$.

In contrast, definitions allow to give a meaningful name (concept name or symbolic name) to concept descriptions, *e.g.,* to define that a mother is a woman that has at least one human child one can write:

$$\textsc{Mother} \equiv \textsc{Woman} \sqcap \exists \mathsf{hasChild}.\textsc{Human} \tag{3.1}$$

Here, $\textsc{Mother}$ is the concept name that identifies the concept description (on the right-hand side of the equivalent symbol). $\textsc{Woman}$ and $\textsc{Human}$ are atomic concepts. If in a terminology an atomic concept appears only on the right-hand side of a concept description, then it is called a *primitive concept*, otherwise it is a *defined concept*.

Note that there has been substantial work on determining the computational impact of allowing various constructs in DLs (see [BCM$^+$03, Tob01] for an overview). Much of this work has focussed on determining decidability and complexity results when different constructors and restrictions are supported or imposed on the particular DL. For example, one such restriction is to only allow definitorial TBoxes; more specifically, only inclusion axioms of the form $A \sqsubseteq C$ and $A \equiv C$ (note that $A \sqsubseteq C$ is an abbreviation for $A \sqsubseteq C$and $C \sqsubseteq A$) are allowed, such that A is an atomic concept and the definitions are unique and acyclic (*i.e.,* the right hand side of an axiom cannot directly or indirectly refer to the concept on its left hand side). It has been shown that this greatly simplifies reasoning complexity [Tob01]. If the $\mathcal{T}$ contains an axiom of the form $C \sqsubseteq D$ where C is a complex concept, then this axiom is referred to as a *general concept inclusion axiom* (GCI) and the $\mathcal{T}$ is referred to as a *general TBox*.

World Descriptions (ABox)

The second component of a DL-knowledge base is the world description or ABox. In the ABox, one introduces individuals by giving them names, and one asserts properties of these individuals. There are two kinds of basic assertions: concept assertions and role assertions. By a concept assertion, one states that a certain individual a belongs to a concept C, written $C(a)$. By a role assertion, one states that an individual c is a filler of the role R for an individual b, written $R(b, c)$. For instance, to denote that *Tumee* is a mother, and *Khangal* is a man who is the son of *Tumee*, we write:

$$\textsc{Mother}(\mathit{Tumee}) \quad \textsc{Man}(\mathit{Khangal}) \quad \mathsf{hasChild}(\mathit{Tumee}, \mathit{Khangal})$$

where the first two are concept assertions, and the third is a role assertion. Furthermore, expressive DLs provide equality and inequality assertations.

3.2 Expressive Description Logic $\mathcal{SHOIN}$

In this section, we introduce a more expressive description logic $\mathcal{SHOIN}$ which provides the formal underpinning for OWL-DL.

Formal Syntax and Semantics

The countably infinite sets N_I, N_C and N_r of individual names, concept names and role names, respectively, form the basis to construct the syntactic elements of $\mathcal{SHOIN}$ according to the following grammar:

$$\begin{array}{rcl} C & \longrightarrow & A \mid \bot \mid \top \mid \neg C \mid C_1 \sqcap C_2 \mid C_1 \sqcup C_2 \mid \exists r.C \mid \forall r.C \\ & & \mid \geq n\, r \mid \leq n\, r \mid \{a_1, \ldots, a_n\} \\ r & \longrightarrow & s \mid s^- \end{array}$$

, where $A \in \mathsf{N}_C$ denotes an atomic concept, C, C_i denote complex concepts, $s \in \mathsf{N}_r$ denotes an atomic role, r denotes a possibly inverse role, $a_i \in \mathsf{N}_I$ denote individuals and n denotes a natural number.

A $\mathcal{SHOIN}$ knowledge base *KB* is a set of *axioms* that are formed by concepts, roles and individuals according to the rules illustrated in Table 3.3 on page 29.

A *concept inclusion* is an axiom of the form $C_1 \sqsubseteq C_2$ that states the subsumption of the concept C_1 by the concept C_2, while a *role inclusion* is an axiom of the form $r_1 \sqsubseteq r_2$ that states the subsumption of the role r_1 by the role r_2. An equivalence axiom of the form $C_1 \equiv C_2$ for concepts or $r_1 \equiv r_2$ for roles is a shortcut for two inclusions $C_1 \sqsubseteq C_2$ and $C_2 \sqsubseteq C_1$ or $r_1 \sqsubseteq r_2$ and $r_2 \sqsubseteq r_1$. A *role transitivity* statement is an axiom of the form $\mathsf{Trans}(r)$ that states transitivity for the role r. A *concept assertion*

α_T	$\longrightarrow$	$C_1 \sqsubseteq C_2$	$\mid$	concept inclusion
		$C_1 \equiv C_2$	$\mid$	concept equivalence
α_R	$\longrightarrow$	$r_1 \sqsubseteq r_2$	$\mid$	role inclusion
		$r_1 \equiv r_2$	$\mid$	role equivalence
		$\mathsf{Trans}(r)$		role transitivity
α_A	$\longrightarrow$	$C(a)$	$\mid$	concept assertion
		$r(a_1, a_2)$	$\mid$	role assertion
		$a_1 \approx a_2$	$\mid$	individual equality
		$a_1 \not\approx a_2$		individual inequality

Table 3.3: Axioms in $\mathcal{SHOIN}$: α_T, α_R and α_A denote axioms

is an axiom of the form $C(a)$ that assigns the membership of an individual a to a concept C. A *role assertion* is an axiom of the form $r(a_1, a_2)$ that assigns a directed relation between two individuals a_1, a_2 by the role r. The axioms in *KB* are partitioned into a $\mathcal{T}$, an R-Box and an $\mathcal{A}$ and take the forms α_T, α_R and α_A, respectively. The $\mathcal{T}$ and R-Box describe terminological knowledge in terms of general statements about the domain, whereas the $\mathcal{A}$ describes assertional knowledge in terms of particular instance situations. By $\sigma(KB)$ we denote the signature of the knowledge base *KB*, which is the set of all concept, role and individual names that occur in the axioms of *KB*.

The semantics of the syntactic elements of $\mathcal{SHOIN}$ is defined in terms of an *interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ with a non-empty set $\Delta^{\mathcal{I}}$ as the *interpretation domain* and an *interpretation function* $\cdot^{\mathcal{I}}$ that maps each individual $a \in \mathsf{N}_I$ to a distinct element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$ and that interprets (possibly) complex concepts and roles as indicated in Table 3.4 on page 30.

An interpretation $\mathcal{I}$ satisfies a concept inclusion $C_1 \sqsubseteq C_2$ if $C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$ and a concept equivalence $C_1 \equiv C_2$ if $C_1^{\mathcal{I}} = C_2^{\mathcal{I}}$. Similarly, it satisfies a role inclusion $r_1 \sqsubseteq r_2$ if $r_1^{\mathcal{I}} \subseteq r_2^{\mathcal{I}}$ and a role equivalence $r_1 \equiv r_2$ if $r_1^{\mathcal{I}} = r_2^{\mathcal{I}}$. A transitivity axiom $\mathsf{Trans}(r)$ is satisfied in $\mathcal{I}$ if for all $a_1, a_2, a_3 \in \Delta^{\mathcal{I}}$ $(a_1^{\mathcal{I}}, a_2^{\mathcal{I}}) \in r^{\mathcal{I}}$ and $(a_2^{\mathcal{I}}, a_3^{\mathcal{I}}) \in r^{\mathcal{I}}$ together imply $(a_1^{\mathcal{I}}, a_3^{\mathcal{I}}) \in r^{\mathcal{I}}$. Moreover, $\mathcal{I}$ satisfies a concept assertion $C(a)$ if $a^{\mathcal{I}} \in C^{\mathcal{I}}$, a role assertion $r(a_1, a_2)$ if $(a_1^{\mathcal{I}}, a_2^{\mathcal{I}}) \in r^{\mathcal{I}}$, an individual equality $a_1 \approx a_2$ if $a_1^{\mathcal{I}} = a_2^{\mathcal{I}}$ and an individual inequality $a_1 \not\approx a_2$ if $a_1^{\mathcal{I}} \neq a_2^{\mathcal{I}}$. An interpretation that satisfies all axioms of a knowledge base *KB* is called a *model* of *KB*, and we denote by $\mathcal{M}(KB)$ the set of all models of *KB*. Reasoning with a description logic knowledge base is defined in terms of this notion of a model.

Note that the general semantics of DLs as given in Table 3.4 includes (in)equality between individuals, a feature also supported by OWL. However, in applications individuals are often treated as uniquely named such that no two named individuals coincide in the interpretation domain, which is known as the *unique name assumption* [BCM+03]. The unique name assumption can be axiomatised in OWL by including

$$
\begin{array}{rcl}
\top^{\mathcal{I}} & = & \Delta^{\mathcal{I}} \quad , \quad \bot^{\mathcal{I}} \; = \; \emptyset \\
A^{\mathcal{I}} & \subseteq & \Delta^{\mathcal{I}} \quad , \quad r^{\mathcal{I}} \; \subseteq \; \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \\
(C_1 \sqcap C_2)^{\mathcal{I}} & = & C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}} \\
(C_1 \sqcup C_2)^{\mathcal{I}} & = & C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}} \\
(\neg C)^{\mathcal{I}} & = & \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}} \\
(\forall\, r\,.C)^{\mathcal{I}} & = & \{a \in \Delta^{\mathcal{I}} \mid \forall b.(a,b) \in r^{\mathcal{I}} \rightarrow b \in C^{\mathcal{I}}\} \\
(\exists\, r\,.C)^{\mathcal{I}} & = & \{a \in \Delta^{\mathcal{I}} \mid \exists b.(a,b) \in r^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\} \\
(\geq n\, r.)^{\mathcal{I}} & = & \{a \in \Delta^{\mathcal{I}} \mid \#\{b \mid (a,b) \in r^{\mathcal{I}}\} \geq n\} \\
(\leq n\, r.)^{\mathcal{I}} & = & \{a \in \Delta^{\mathcal{I}} \mid \#\{b \mid (a,b) \in r^{\mathcal{I}}\} \leq n\} \\
\{a_1, \ldots, a_n\}^{\mathcal{I}} & = & \{a_1^{\mathcal{I}}, \ldots, a_n^{\mathcal{I}}\} \\
(r^{-})^{\mathcal{I}} & = & \{(b,a) \mid (a,b) \in r^{\mathcal{I}}\}
\end{array}
$$

Table 3.4: Model-theoretic semantics for $\mathcal{SHOIN}$ descriptions

inequality axioms of the form $a_1 \not\approx a_2$ for any pair a_1, a_2 of known individuals.

3.3 DL Reasoning Problems

For DLs various reasoning tasks are usually considered. These tasks allow to draw new conclusions about the knowledge base or check its consistency. In this section standard reasoning problems are introduced. The interested reader may refer to [BCM$^+$03] for more details.

Concept Satisfiability. Given a concept C, checking if C is satisfiable with respect to knowledge base KB is the task of determining if there exists an interpretation $\mathcal{I}$ of KB such that the interpretation of C is not equal to the empty set (i.e. $C^{\mathcal{I}} \neq \emptyset$).

Concept Subsumption . Given concepts C, D, checking if C is subsumed by D relative to KB, denoted $KB \models C \sqsubseteq D$, is the process of determining if for all interpretations $\mathcal{I}$ of KB, $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$.

Instance checking. Instance checking ensures whether or not an individual a is an instance of a concept C with respect to KB. It is denoted by $KB \models C(a)$, if $a^{\mathcal{I}} \in C^{\mathcal{I}}$ holds for all models $\mathcal{I}$ of KB.

Instance Retrieval. An extended form of instance checking is the so called retrieval problem. It is stated as follows. Given a concept C, find all individuals a such that $KB \models C(a)$.

Note that all reasoning tasks can be reduced to ABox consistency checking [BN03]. This is exemplified by the following example: suppose that we want to check if an individual a instantiates a concept C with respect to a knowledge base KB; this is accomplished by checking the consistency of $KB \cup \{\neg C(a)\}$. If this is not consistent, then it must be the case that there does not exist an interpretation which satisfies $\neg C(a)$, therefore all interpretations must satisfy $C(a)$.

3.4 DL Reasoning Complexities

This section briefly introduces various complexity results on DLs. This introduction shall also clarify the usefullness of approximate logic reasoning methods which aim to lower worst the case complexity of exponentional algorithms.

Description logic	Complexity
$\mathcal{S}$	PSPACE-complete (without TBox)
$\mathcal{SI}$	PSPACE-complete
$\mathcal{SH}$	EXPTIME-complete
$\mathcal{SHIF}$	EXPTIME-complete
$\mathcal{SHIQ}$	EXPTIME-complete
$\mathcal{SHOIQ}$	NEXPTIME-hard

Table 3.5: Complexity of Satisfiability for $\mathcal{S}$ languages

For description logics without full negation, *e.g.,* AL, all inferences can be reduced to subsumption. For example, a class C is unsatisfiable iff C is subsumed by $\perp$. If, a description logic offers both intersection and full complement, satisfiability becomes the key inference of terminologies, since all other inferences can be reduced to satisfiability [AKPS94]. Consequently, algorithms for checking satisfiability are sufficient to obtain decision procedures for any of the inference problems discussed in Section 3.3. Moreover, this observation gave rise to the research on specialized tableau calculi which are used in the current generation of DL systems.

Naturally, the question arises how difficult it is to deal with the reasoning problems introduced in Section 3.3. Traditionally, the complexity of reasoning has been one of the major issues in the development of description logics. While studies about the complexity of reasoning problems initially were focused to polynomial-time versus intractable [BL84], the focus nowadays has shifted to very expressive logics such as $\mathcal{SHIQ}$ whose reasoning problems are EXPTIME-hard or worse.

One can see that exponential-time behavior of some description logics is due to two independent origins: an AND source which corresponds to the size of a single model candidate and an OR source which is constituted by the number of model candidates that have to be checked. An example OR source is disjunction, where we can alternatively assign an element of a model to several classes and we have to check every alternative. No OR source therefore means that we can check the validity of a single model. However, if an AND source such as the existential restriction is present, we may have to expand the model with new elements. The complexity of the $\mathcal{S}$ family of languages that are relevant for the Semantic Web is listed in Table 3.5.

The presence of a cyclic TBox can increase the complexity of reasoning problems. Even for $\mathcal{AL}$, presence of a general TBox leads to EXPTIME-hardness [Neb90], which also effects description logics like $\mathcal{SH}$, which allow the internalization[2] of general inclusion axioms. An extension with datatypes also effects the complexity. Given that we distinguish datatype properties, satisfiability is decidable if the inference problems for the concrete domain are decidable [HS01]. Complexity results also give motivation why no other property constructors have been presented here. For example, adding role composition to $\mathcal{AL}$ already leads to undecidability of the description logic [BCM$^+$03].

[2] A process of transforming axioms into concept expressions.

Chapter 4

Logic Programming, Relational and Deductive databases

This chapter is divided into three sections. Section 4.1 contains elementary fundamentals of logic programming. The basic concepts presented in Section 4.1 are used in Section 6.3, Chapter 6 where knowledge compilation methods are considered. The reader may therefore have a look at this section if some considerations of Section 6.3 are not clear enough. Section 4.2 provides elementary fundamentals of relational databases which are used in Section 7.3, Chapter 7. A brief introduction to deductive databases is presented in Section 4.3.

4.1 Logic Programming

Logic programming (LP) is a well-known declarative of knowledge representation and programming based on the idea that the language of first order logic is well-suited for both representing data and describing desired outputs [Llo87]. LP was developed in the early 1970's based on work in automated theorem proving [BG01, DP01], in particular, on Robinson's resolution principle [Rob65]. In the following we give some basic concepts of logic programming and refer the reader to [Llo87] for a more detailed treatment. We use letters $p, q, \ldots$ for predicate symbols, $X, Y, Z \ldots$ for variables, $f, g, h, \ldots$ for function symbols, and $a, b, c, \ldots$ for constants.

Basic Notions

Logic programms are formulated in a language $\mathcal{L}$ of predicates and functions of non-negative arity; 0-ary functions are constants. A language $\mathcal{L}$ is *function-free* if it contains no function symbols of arity greater than 0.

A *term* is inductively defined as follows: each variable X and constant c is a term, and if f is a n-ary function symbols and $t_1, ..., t_n$ are terms, then $f(t_1, ..., t_n)$ is a term. A term is a *ground term*, if no variable occurs in it. An *atom* is a formula $p(t_1, ..., t_n)$, where p is a predicate symbol of arity n and each t_i is a term. An atom is *ground*, if all t_i are ground.

A logic program consists of a set of rules, also called Horn clauses of the form:

$$A_0 \leftarrow A_1, ..., A_m \qquad (m \geq 0)$$

where each A_i is a atomic formula and all variables occurring in a formula are (implicitly) universally quantified over the whole formula. The formulas of this form are called *definite clauses*, also called rules. The atomic formula A_0 is called the *head* of the clause whereas $A_1, ..., A_m$ is called its *body*. A rule r of the form $A_0 \leftarrow$, *i.e.*, whose body is empty, is called a fact, and if A_0 is a ground atom, then r is called a ground fact. A clause or logic program is ground, if all terms in it are ground.

The *Herbrand Universe* of a program P, denoted by $U_\mathcal{P}$, is the set of all ground terms that can be formed from the constants and function symbols in P. The *Herbrand Base* of a logic program P, denoted by $B_\mathcal{P}$, is defined as the set of all ground atoms that can be formed by using predicates from P with terms from the Herbrand Universe $U_\mathcal{P}$ as arguments [Llo87]. A *Herbrand interpretation I* for P is a subset of the Herbrand Base of P. A *Herbrand model* of P is a Herbrand interpretation of P such that for each rule $A_0 \leftarrow A_1, ..., A_m$ in P, this interpretation satisfies the logical formula $\forall X(A_1 \wedge ... \wedge A_m) \Rightarrow A_0)$, where X is a list of the variables in the rule and $\Rightarrow$ is logical implication.

So far a restricted form of clauses is concerned. In order to allow more suiteable knowledge representation and to increase expressiveness, other forms of clauses are required. A clause containing disjunction is called disjunctive rule and is of the form:

$$A_1 \vee ... \vee A_n \leftarrow B_1 \wedge ... \wedge B_m$$

where $n \geq 0$, $m \geq 0$, and A_i and B_i are atomic formulas. Furthermore, each rule must be *safe*; that is, each variable occurring in a head literal must occur in a body literal as well. For a disjunctive rule r, the set of atoms $\mathsf{head}(r) = \{A_i \,|\, 1 \leq i \leq n\}$ is called the *rule head*, whereas the set of atoms $\mathsf{body}(r) = \{B_i \,|\, 1 \leq i \leq m\}$ is called the *rule body*. A rule with an empty body is called a *fact*. A finite set of such clauses is called a disjunctive datalog program. A clause with the empty head and a non-empty body is called an *integrity constraint*. A program containing no disjunctive clauses is called a *Horn logic program*, and especially a Horn logic program containing no integrity constraint is called a *definite logic program*. For more materials on disjunctive logic programming, the reader might refer to [EGM97].

SLD-Resolution

In traditional (imperative) programming languages, a program is a procedural specification of how a problem needs to be solved. In contrast, a logic program concentrates on a declarative specification of what the problem is. Rather than viewing a computer program as a step-by-step description of an algorithm, a logic program is conceived as a logical theory (knowledge base), and a procedure call is viewed as a theorem of which the truth needs to be established. Thus, executing a program means searching for a proof by applying inference rules. SLD resolution (**S**elective **L**inear **D**efinite clause resolution) is the basic inference rule used in logic programming. It is a refinement of resolution[1], which is both sound and refutation complete for Horn clauses.

Roughly, SLD-resolution can be described as follows. A goal is a conjunction of atoms. A substitution is a function θ that maps variables $v_1, ..., v_n$ to terms $t_1, ..., t_n$. The result of simultaneous replacement of variables v_i by terms t_i in an expression E is denoted by $E\theta$. For a given goal $\mathcal{G}$ and $\mathcal{P}$, SLD-resolution tries to find a substition $E\theta$ such that logically follows from $\mathcal{P}$.

The initial goal is repeatedly transformed until the empty goal is obtained. Each transformation step is based on the application of the resolution rule to a selected atom B from the goal $B_1, ..., B_m$ and a clause $A_0 \leftarrow A_1, ..., A_n$ from $\mathcal{P}$. SLD-resolution tries to unify B_i with the head A_0 *i.e.,* to find a substition θ such that $A_0\theta = B_i\theta$. If such a unifier θ is found, the goal is transformed into

$$(B_1, ..., B_{i-1}, A_1, ..., A_n, B_{i+1}, ..., B_m)\theta.$$

Although SLD-Resolution is complete and sound [Llo87], its implementation in Prolog[2] is not sound in general [O'K90]. Readers interested in a more thorough treatment of the SLD-resolution might refer to [Rob65].

4.2 Relational Databases

There are many similarities between relational databases and logic programming in that they are both used to describe relations between objects.

Relational Algebra is the formal underpinning of modern relational database systems and is used to formalise database operations on the relational model originally introduced by Codd [Cod83]. The main construct for representing data in the relational model is a *relation*.

Let $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n$ be collections of symbols called domains. In the context of database theory the domains are usually assumed to be finite although, for practical reasons, they normally include an infinite domain of numerals. In addition, the members of the domains are normally assumed to be atomic or indivisible – that is, it is not possible to access a proper part of a member.

A *database relation* R over the domains $\mathcal{D}_1, \ldots, \mathcal{D}_n$ is a subset of $\mathcal{D}_1 \times \cdots \times \mathcal{D}_n$ and represents a database table with attributes a_1 to a_n and rows that instantiate the columns as tuples of values. A relational database is a finite number of such (finite) relations. R is in this case said to be n-ary and denoted by $R(a_1, \ldots, a_n)$,

[1] In mathematical logic and automated theorem proving, resolution is a rule of inference leading to a refutation theorem-proving technique for sentences in propositional logic and first-order logic.

[2] A general purpose logic programming language.

Relational algebra expressions are used to formulate queries on the thus represented database tables and result themselves in relations, such that expressions can be nested. Attributes in a relation can be referred to by means of path expressions of the form $R\,.\,a_i$, *e.g.,* within conditions. The primitive operations of relational algebra are union, set difference, cartesian product, projection and selection.

Given two n-ary relations over the same domains, the union of the two relations, R_1 and R_2 (denoted $R_1 \cup R_2$), is the set:

$$\{\langle x_1, \ldots, x_n \rangle \mid \langle x_1, \ldots, x_n \rangle \in R_1 \vee \langle x_1, \ldots, x_n \rangle \in R_2\}$$

The difference $R_1 \setminus R_2$ of two relations R_1 and R_2 over the same domains yields the new relation:

$$\{\langle x_1, \ldots, x_n \rangle \in R_1 \mid \langle x_1, \ldots, x_n \rangle \notin R_2\}$$

The *cartesian* (cross) product of two relations R_1 and R_2 (denoted $R_1 \times R_2$) yields the new relation:

$$\{\langle x_1, \ldots, x_m, y_1, \ldots, y_n \rangle \mid \langle x_1, \ldots, x_m \rangle \in R_1 \wedge \langle y_1, \ldots, y_n \rangle \in R_2\}$$

Note that R_1 and R_2 may have both different domains and different arities. Moreover, if R_1 and R_2 contain disjoint sets of attributes they are carried over to the resulting relation. However, if the original relations contain some joint attribute the attribute of the two columns in the new relation must be renamed into distinct ones. This can be done *e.g.,* by prefixing the joint attributes in the new relation by the relation where they came from. For instance, in the relation $R(A, B) \times S(B, C)$ the attributes are, from left to right, A, $R.B$, $S.B$ and C. Obviously, it is possible to achieve the same effect in other ways.

A *projection* $\pi_{[a_1,\ldots,a_m]}(R(a_1, \ldots, a_n))$ restricts the columns of the resulting relation to the attributes $a_1, \ldots, a_m$ for $m < n$.

The *selection* of a relation R is denoted $\sigma_{[F]}(R)$ (where F is a formula) and is the set of all tuples $\langle x_1, \ldots, x_m \rangle \in R$ such that F is true for $(x_1, \ldots, x_m)$.

Some other operations (like natural join and composition) are sometimes encountered in relational algebra but they are usually all defined in terms of the mentioned, primitive ones and are therefore not discussed here. For a detailed description of relational algebra, the reader might refer to [Cod83].

4.3 Deductive Databases

Since conventional database systems suffer from the limited power of their query languages, there has been a growing interest in the database community to use logic programs as a language for representing *data*, *integrity constraints*, *views* and *queries* in a single uniform framework. Deductive databases grown out of this interest to combine logic programming with relational databases to construct systems that support a powerful formalism and are still fast and able to deal with very large datasets.

They provide a declarative, logic-based language for expressing queries, reasoning, and complex applications on databases.

Datalog is a restricted version of logic programs without functional symbols and is the language typically used to specify facts, rules and queries in deductive databases. Over database languages such as relational algebra or SQL[3], datalog has the advantage of being able to express recursive queries. In datalog, relations that are are physically stored in the database are called extensional database (EDB) relations and are identical to relations in the relational data model. The main difference between datalog and relational model is that it also allows relations which are defined by logical rules, called intensional database (IDB) relations. Datalog also has a number of built-in predicates for standard arithmetic comparison. Any predicate that is not built-in is called ordinary.

A salient feature of deductive databases is the development of efficient query evaluation algorithms for handling cycles of recursive rules. This is a most useful feature since cyclic graphs are often stored in database relations, and derived relations can also be circular.

The standard methods for evaluating queries in logic are backward-chaining (or top-down) and forward-chaining (or bottom-up). In backward-chaining, the system uses the query as a goal and creates more goals by expanding each head into its body. This approach ensures that only potentially relevant goals are explored but can result in infinite loops. Forward-chaining starts with the EDB and repeatedly uses the rules to infer more facts. As such, it avoids the problems of looping, but may infer many irrelevant facts. An important result from deductive databases is the magic sets technique [CFGL04], which rewrites rules so that a forward-chaining evaluation will only consider potentially relevant goals similar to those explored by backward-chaining.

For comprehensive materials on deductive databases, the interested reader might refer to [ABW88]. For query evaluation techniques, please refer to [CFGL04].

[3] `http://en.wikipedia.org/wiki/SQL` [accessed 2009-07-22]

Part II

Approximate Reasoning For the Semantic Web

Chapter 5

A Foundation of Approximate Reasoning Research

Approximate reasoning for the Semantic Web is based on the idea of sacrificing soundness or completeness for a significant speed-up of reasoning. This is to be done in such a way that the number of introduced mistakes is at least outweighed by the obtained speed-up. When pursuing such approximate reasoning approaches, however, it is important to be critical not only about appropriate application domains, but also about the quality of the resulting approximate reasoning procedures. With different approximate reasoning algorithms discussed and developed in the literature, it needs to be clarified how these approaches can be compared, *i.e.,* what it means that one approximate reasoning approach is better than some other.

This chapter provides a foundation for approximate reasoning research. We will clarify – by means of notions from statistics – how different approximate algorithms can be compared, and ground the most fundamental notions in the field formally.

5.1 Motivation

In different application areas of Semantic Technologies, the requirements for reasoning services may be quite distinct; while in certain fields (as in safety-critical technical descriptions) soundness and completeness are to be rated as crucial constraints, in other fields less precise answers could be acceptable if this would result in a faster response behaviour.

Introducing approximate reasoning in the Semantic Web field is motivated by the following observation: most nowadays' specification languages for ontologies are quite expressive, reasoning tasks are supposed to be very costly with respect to time and other resources – this being a crucial problem in the presence of large-scale data. As a prominent example, note that reasoning in most description logics which include general concept inclusion axioms (which is simply standard today, and *e.g.*, the case in OWL DL) is at least EXPTIME complete, and if nominals are involved (as for OWL

DL) even NEXPTIME complete. Although those worst case time complexities are not likely to be thoroughly relevant for the average behaviour on real-life problems, this indicates that not every specifiable problem can be solved with moderate effort.

In many cases, however, the time costs will be the most critical ones, as a user will not be willing to wait arbitrarily long for an answer. More likely, she would be prone to accept "controlled inaccuracies" as a tradeoff for quicker response behaviour. However, the current standard reasoning tools (though highly optimized for accurate, *i.e.,* sound and complete reasoning) do not comply with this kind of approach: in an all-or-nothing manner, they provide the whole answer to the problem after the complete computation. It would be desirable, however, to have reasoning systems at hand which can generate good approximate answers in less time, or even provide "anytime behaviour", which means the capability of yielding approximate answers to reasoning queries during ongoing computation: as time proceeds, the answer will be continuously refined to a more and more accurate state until finally the precise result is reached. Clearly, one has to define this kind of behaviour (and especially the notion of the intermediate inaccuracy) more formally.

These ideas of approximate reasoning are currently cause for controversial discussions. On the one hand, it is argued that soundness and completeness of Semantic Web reasoning is not to be sacrificed at all, in order to stay within the precise bounds of the specified formal semantics. On the other hand, it is argued that the nature of many emerging Semantic Web applications involves data which is not necessarily entirely accurate, and at the same time is critical in terms of response time, so that sacrificing reasoning precision appears natural [FH07].

Another way of achieving scalable reasoning is to restrict knowledge representation to so-called tractable fragments that allow for fast sound and complete reasoning. Although this might be useful in scenarios where all essential knowledge can be modelled within the restricted fragment, in general there are strong arguments in favor of the usage of expressive formalisms:

- Real and comprehensive declarative modelling should be possible. A content expert wanting to describe a domain as comprehensive and as precisely as possible will not want to worry about limiting scalability or computability effects.

- As research proceeds, more efficient reasoning algorithms might become available that could be able to more efficiently deal with expressive specification formalisms. Having elaborated specifications at hand enables to reuse the knowledge in a more advanced way.

- Finally, elaborated knowledge specifications using expressive logics can reduce engineering effort by horizontal reuse: Knowledge bases could then be employed for different purposes because the knowledge is already there. However, if only shallow modelling is used, updates would require overhead effort.

From our perspective, it depends on the specifics of the problem at hand whether approximate reasoning solutions can or should be used. We see clear potential in the fields of information retrieval, semantic search, as well as ontology engineering support, to name just a few examples.

At the same time, however, we would like to advocate that allowing for unsound and/or incomplete reasoning procedures in such applications must not lead to arbitrary “guessing” or to deduction algorithms which are not well-understood. Quite on the contrary, we argue that in particular for approximate reasoning, it is of utmost importance to provide ways of determining how feasible the approximations are, *i.e.,* of what quality the answers given by such algorithms can be expected to be.

Obviously, soundness and completeness with respect to the given formal semantics of the underlying knowledge representation languages cannot be used as a measure for assessing the quality of approximate reasoning procedures. Instead, they must be evaluated experimentally, and analysed by statistical means.

5.2 A Mathematical Framework for the Study of Approximate Reasoning

In this section, we lay the foundations for a statistical approach to evaluating approximate reasoning algorithms. We will do this in an abstract manner, which can be made concrete in different ways, depending on the considered use case. At the same time, we use this statistical perspective to precisely define approximate reasoning notions.

First, let us stipulate some abbreviations which we will use in the sequel: let
$\mathbb{R}^+ = \{x \in \mathbb{R} : x \geq 0\}$ and $\mathbb{R}^+_\infty = \{x \in \mathbb{R} : x \geq 0\} \cup \{+\infty\}$.

First of all, we have to come up with a general and generic formalization of the notion of a *reasoning task*. Intuitively, this is just a *question* (or query) posed to a system that manages a knowledge base, which is supposed to deliver an *answer* after some processing *time*. The (maybe gradual) validity of the given answer can be evaluated by investigating its compliance with an abstract semantics. We will extend this classical conceptualisation in the following way: we allow an algorithm to – roughly spoken – change or refine its output as time proceeds, thus capturing the notion of *anytime behaviour*, as a central concept in approximate reasoning. Yet in doing so, we have to take care not to lose the possibility of formalizing “classical” termination. We solve this by stipulating that every output of the system shall be accompanied by the information, whether this output is the ultimate one.

In the sequel we will formalize those intuitions. By the term INPUT SPACE we denote the set of possible concrete reasoning tasks. Formally, we define the input space as a probability space (Ω, P), where Ω is some set (of inputs) and P is a probability measure on Ω. The probability $P(\omega)$ encodes how often a specific input (knowledge base, query) ω occurs in practice *i.e.,* how relevant it is for practical purposes. Naturally, information about the probability distribution of inputs will be difficult to obtain in practice (since, *e.g.,* in general there can be infinitely many different inputs). So rules of thumb, like giving short queries a higher probability than long ones, or using some kind of established benchmarks, will have to be used until more systematic data is available.

The use of having a probability on the set of inputs is quite obvious: as already stated before, correctness of results cannot be guaranteed in the approximate case. So in order to estimate how good an algorithm performs in practice, it is not only important, how much the given answer to a specific input deviates from the correct one, but also how likely (or: how often) that particular input will be given to the system. Certainly, a wrong (or strongly deviant) answer to an input will be more tolerable if it occurs less often.

For actual evaluations, one will often use a discrete probability space. For the general case – for developing the theory in the sequel – we will assume that all occurring functions are measurable (*i.e.,* integrals over them exist), which is obviously a very mild assumption from a computer science perspective.

The OUTPUT SPACE comprises all possible answers to any of the problems from the input space. In our abstract framework, we define it simply as a set X. A function $e : X \times X \to \mathbb{R}^+$ – which we call *error function* – gives a quantitative measure as to what extent an output deviates from the desired output (as given by a sound and complete algorithm). More precisely, the real number $e(x, y)$ stands for the error in the answer x, assuming that y would be the correct answer. For all $x \in X$ we assume $e(x, x) = 0$, but we place no further constraints on e. It will be determined by the problem under investigation, though a suitable example could be $1 - f$, where f is the *f-measure* as known from information retrieval. In cases, it might be also useful to put more constraints on the error function, one could *e.g.,* require it to be a metric,[1] if the output space has a structure where this seems reasonable.

We will assess the usefulness of an approximate reasoning algorithm mainly by looking at two aspects: Runtime and error when computing an answer. By introducing the error function, we are able to formalize the fact that out of two wrong answers one might still be better than the other since it is "closer" to the correct result. While this might not seem to make much sense in some cases (*e.g.,* when considering the output set $\{$*true*,*false*$\}$ or other nominal scales[2]), it might by quite valuable in others: When we consider an instance retrieval task, the outputs will be sets of domain individuals. Obviously, one would be more satisfied with an answer where just one element out of hundred is missing (compared to the correct answer) than with a set containing, say, only non-instances.

We assume X to contain a distinguished element $\perp$ which denotes *no output*. This is an issue of "backward compatibility", since classical algorithms – and also many approximate reasoning algorithms – usually do not display any output until termination. So, to include them into our framework, we define them to deliver $\perp$ before giving the ultimate result. $\perp$ will also be used as output value in case the algorithm does not terminate on the given input.

[1] A distance function as used in the mathematical theory of metric spaces.

[2] Although also these cases can seamlessly be covered by choosing a discrete error function.

Since by this definition, $\perp$ contains no real information, one could argue about additional constraints for the error function with respect to this distinguished element, *e.g.*, $e(\perp, y) \geq \sup_{x \in X}\{e(x, y)\}$ or even $e(\perp, y) \geq \sup_{x,z \in X}\{e(x, z)\}$. We do not need to impose these in general, however.

After having formalized inputs and outputs for problems, we now come to the actual algorithms. In order not to unnecessarily overcomplicate our formal considerations, we make some additional assumptions: We assume that hardware etc. is fixed, *i.e.*, in our abstraction, an algorithm is always considered to include the hard- and software environment it is run in. I.e., we can, for example, assign any algorithm-input pair an exact runtime (which may be infinite). This assumption basically corresponds to a "laboratory" setting for experiments, which abstracts from variables currently not under investigation.

So, let $\mathcal{A}$ be a set of algorithms. To every algorithm $a \in \mathcal{A}$ we assign an IO-FUNCTION $f_a : \Omega \times \mathbb{R}^+ \to X \times 2$ with $2 := \{0, 1\}$. Hereby, $f_a(\omega, t) = (x, b)$ means that the algorithm a applied to the input (task, problem, ...) ω yields the result x after running time t together with the information whether the algorithm has already reached its final output ($b = 1$) or not yet ($b = 0$). As a natural constraint, we require f_a to additionally satisfy the condition that for all $t_2 \geq t_1$ we have that

$$f_a(\omega, t_1) = (x, 1) \text{ implies } f_a(\omega, t_2) = (x, 1),$$

i.e., after having indicated termination, the output of the algorithm (including the termination statement) will not change anymore. For convenience we write $f_a^{\mathrm{res}}(\omega, t) = x$ and $f_a^{\mathrm{term}}(\omega, t) = b$, if $f_a(\omega, t) = (x, b)$.

By $f_0 : \Omega \to X$ we denote the CORRECT OUTPUT FUNCTION, which is determined by some external specification or formal semantics of the problem. This enables us to verify the (level of) correctness of an answer $x \in X$ with respect to a particular input ω by determining $e(x, f_0(\omega))$ – the smaller the respective value, the better the answer. By our standing condition on e, $e(x, f_0(\omega)) = 0$ ensures $f_0(\omega) = x$ coinciding with the intuition.

To any algorithm a, we assign a RUNTIME FUNCTION $\varrho_a : \Omega \to \mathbb{R}^+_\infty$ by setting

$$\varrho_a(\omega) = \inf\{t \mid f_a^{\mathrm{term}}(\omega, t) = 1\},$$

being the smallest time, at which the algorithm a applied to input ω indicates its termination.[3] Note that this implies $\varrho_a(\omega) = \infty$ whenever we have $f_a^{\mathrm{term}}(\omega, t) = 0$ for all $t \in \mathbb{R}^+$. Algorithms, for which for all $\omega \in \Omega$ we have that $\varrho_a(\omega) < \infty$ and $f_a^{\mathrm{res}}(\omega, t) = \perp$ for all $t < \varrho_a(\omega)$ are called ONE-ANSWER ALGORITHMS: They give only one output which is not $\perp$, and are guaranteed to terminate[4] in finite time.

Clearly, for a given time t, the expression $e(f_a^{\mathrm{res}}(\omega, t), f_0(\omega))$ provides a measure of how much the current result provided by the algorithm diverges from the correct

[3] We make the reasonable assumption that f_a^{res} is right-continuous.

[4] We impose termination here because our main interest is in reasoning with description logics for the Semantic Web. The same notion *without* imposing termination would also be reasonable, for other settings.

solution. Moreover, it is quite straightforward to extend this notion to the whole input space (by taking into account the occurrence probability of the single inputs). This is done by the next definition.

The DEFECT $\delta(a,t)$ ASSOCIATED WITH AN ALGORITHM $a \in \mathcal{A}$ AT A TIME POINT t is given by

$$\delta : \mathcal{A} \times \mathbb{R}^+ \to \mathbb{R}^+_\infty : \delta(a,t) = E(e(f_a^{\text{res}}(\omega,t), f_0(\omega))) = \sum_{\omega\in\Omega} e(f_a^{\text{res}}(\omega,t), f_0(\omega))P(\omega).$$

Note that E denotes the expected value, which is calculated by the rightmost formula.[5] Furthermore, one can even abstract from the time and take the results after waiting "arbitrarily long": The (ULTIMATE) DEFECT of an algorithm $a \in \mathcal{A}$ is given by

$$\delta : \mathcal{A} \to \mathbb{R}^+_\infty : \delta(a) = \limsup_{t\to\infty} \delta(a,t).$$

By the constraint put on the IO-function we get

$$\delta(a) = E(e(f_a^{\text{res}}(\omega, \varrho_a(\omega)), f_0(\omega))) = \sum_{\omega\in\Omega} e(f_a^{\text{res}}(\omega, \varrho_a(\omega)), f_0(\omega))P(\omega).$$

if an algorithm a terminates for every input.

5.2.1 Comparing Algorithms After Termination

For $a, b \in \mathcal{A}$, we say that a is MORE PRECISE THAN b if it has smaller ultimate defect, *i.e.,* if

$$\delta(a) \leq \delta(b).$$

Furthermore, it is often interesting to have an estimate on the runtime of an algorithm. Again it is reasonable to incorporate the problems' probabilities into this consideration. So we define the AVERAGE RUNTIME[6] of algorithm a by

$$\alpha(a) = E(\varrho_a(\omega)) = \sum_{\omega\in\Omega} \varrho_a(\omega)P(\omega).$$

This justifies to say that a is QUICKER THAN b if

$$\alpha(a) \leq \alpha(b).$$

Note that this does not mean that a terminates earlier than b on every input. Instead, it says that when calling the algorithm very often, the overall time when using a will be smaller than when using b – weighted by the importance of the input as measured by P.

[5] The sum could easily be generalised to an integral – with P being a probability measure –, however it is reasonable to expect that Ω is discrete, and hence the sum suffices.

[6] We are aware that in some cases, it might be more informative to estimate the runtime behaviour via other statistical measures as *e.g.,* the median.

Throughout the considerations made until here, it has become clear that there are two dimensions along which approximate reasoning algorithms can be assessed or compared: runtime behaviour and accuracy of the result. Clearly, an algorithm will be deemed better, if it outperforms another one with respect to the following criterion:

Definition 5.1. *For $a, b \in \mathcal{A}$, we say that a* IS STRONGLY BETTER THAN *b if a is more precise than b and a is quicker than b.*

The just given definition is very strict; a more flexible one will be given below, when we introduce the notion that an algorithm a is *better than* an algorithm b.

Note, however, that this definition puts strong constraints on the compared algorithms' termination behaviour. A more sensitive notion would be the following:

We define the ...WHATEVER...FUNCTION

$$\theta_a : \mathbb{R}^+ \to \mathbb{R}^+ : \theta_a(e) = \inf_{t \in \mathbb{R}^+} \{t \mid \forall t' \geq t : \delta(a, t') \leq e\}.$$

5.2.2 Anytime Behaviour

The definitions just given in Section 5.2.1 compare algorithms after termination, *i.e.,* anytime behaviour of the algorithms is not considered. In order to look at anytime aspects, we need to consider the continuum of time points from initiating the anytime algorithm to its termination.

For $a, b \in \mathcal{A}$, we say that a is MORE PRECISE THAN b AT TIME POINT t if it has smaller defect wrt. a and t, *i.e.,* if

$$\delta(a, t) \leq \delta(b, t).$$

We say that $a \in \mathcal{A}$ REALISES A DEFECTLESS APPROXIMATION if

$$\lim_{t \to \infty} \delta(a, t) = 0.$$

Note that $\delta(a) = 0$ in this case.

Definition 5.2. *We say that an algorithm $a \in \mathcal{A}$ is an* ANYTIME ALGORITHM *if it realizes a defectless approximation. We say that it is a* MONOTONIC ANYTIME ALGORITHM *if it is an anytime algorithm and furthermore $\delta(a, t)$ is monotonically decreasing in t, i.e., if $\delta(a, \cdot) \searrow 0$.*

Obviously, is reasonable to say about two algorithms a and b – be they anytime or not –, that (1) a is better than b if a is more precise than b at any time point. A less strict – and apparently more reasonable – requirement accumulates the difference between a and b over the entire runtime, stating that (2) a is better than b if

$$\sum_{\omega \in \Omega} P(\omega) \int_{t=0}^{\max\{\varrho_a(\omega), \varrho_b(\omega)\}} (e(f_a^{\text{res}}(\omega, t), f_0(\omega)) - e(f_b^{\text{res}}(\omega, t), f_0(\omega)) \mathrm{d}t \leq 0.$$

We find formula (2) still not satisfactory as it ignores the reasonable assumption that some time points might be more important than others, *i.e.,* they need to be weighted more strongly. Formally, this is done by using a different measure for the integral or – equivalently – a density function $\bar{f} : \mathbb{R}^+ \to \mathbb{R}^+$, which modifies the integral. Summarizing, we now define for two (not necessarily anytime) algorithms a and b that (3) a IS BETTER THAN b (wrt. a given density function $\bar{f}$) if

$$\sum_{\omega \in \Omega} P(\omega) \int_{t=0}^{\max\{\varrho_a(\omega),\varrho_b(\omega)\}} \big(e(f_a^{\text{res}}(\omega,t), f_0(\omega)) - e(f_b^{\text{res}}(\omega,t), f_0(\omega))\big)\bar{f}(t)\mathrm{d}t \leq 0.$$

Our definition (3) specialises to the case in (2) for the constant density function $\bar{f} \equiv 1$. We cannot capture (1) with our definition by one specific choice of $\bar{f}$, so in the case of (1) we simply say that a is more precise than b at any time point.[7]

Clearly, the choice of the density function depends on the considered scenario. In cases where only a fixed time t_{timeout} can be waited before a decision has to be made based on the results acquired so far, the value $\bar{f}(t)$ of density function would be set to zero for all $t \geq t_{\text{timeout}}$. Usually earlier results are preferred to later ones which would justify the choice of an $\bar{f}$ that is monotonically decreasing.

5.3 State-of-the art of Benchmarking

The previous section provided a mathematical framework for measuring the performance and quality of approximate reasoning algorithms. This section discusses state-of-the art of benchmarking of ontology reasoning systems. In the area of ontology reasoning, benchmarking is often used to evaluate and compare performance of numerous implementations of reasoning algorithms. Systemantic benchmarking can reveal performance bottlenecks and bad design decisions as well as implementation errors. Benchmark results are regularly used both by the entire community, to explore strengths and weaknesses of reasoning systems, running on different benchmarks, and by the reasoning tool developers, to evaluate the performance of their tools on critical knowledge bases, to assess implementation changes that may have an impcat on performance, and to determine the extend of such impact.

Similar to benchmarking conventional software systems, the main goal of benchmarking a complete and sound reasoning system is to measure various performances such as loading, preprocessing and query response time, assuming that the system has correctly implemented its underlying reasoning algorithm.

The main concern of designing such a benchmark is to develop a complex ontology populated with a large set of instance data, and handpick some queries based on certain criteria. In this regard, a typical benchmark on complete and sound ontology reasoning systems comes with one or more test ontologies and a relatively small set

[7] However, (1) could be formulated in terms of (3) as a being better than b for all Dirac delta functions that have their singularity at a nonnegative place.

of test queries. There are already several well-known benchmarks and benchmarking frameworks for evaluating DL reasoning systems. An overview of the well-known benchmarks is shown in Table 5.1.

Table 5.1: Well-known benchmark ontologies used for evaluating ontology reasoning systems [Mot06]

ontology	expressivity	test queries
vicodi	ALHI	2
semintec	ALCOIF	2
lubm	ALEHI(D)	14
wine	SHOIN	0
dolce	SHIN(D)	0
galen	ALEHIF+	0

It shows some statistics of the benchmark ontologies which indicate their expressivity and the number of the test queries used to evaluate DL reasoning systems [Mot06]. There are two main reasons that these benchmarks are not well-suited for the evalutaton of approximate reasoning methods. One reason is that they include relatively few test queries which are not sufficient to measure the quality of approximate reasoning methods. The other reason is that these benchmark ontologies are not expressive enough to show the advantage of approximate reasoning systems can be shown.

In summary, existing ontology reasoning benchmarks focus on measuring performances while measuring soundness and completeness has been limited to occasional measurement and comparison. Automated, thorough, soundness and completeness benchmarking has been neglected in most of the benchmarks available in the area of ontology reasoning.

5.4 Benchmarking of Approximate Instance Retrieval Algorithms

Clearly, the ultimate goal of approximate reasoning research is to develop approximate reasoning systems that are both effective and efficient. The performance of an approximate reasoning algorithm can be easily evaluated by measuring response times. Evaluating the quality of an approximate reasoning algorithm, however, is a highly complex issue. This section analyses the issues which make the evaluation of approximate reasoning methods difficult.

1. *Creation of benchmarks*

   The first difficulty is that we do not only want to measure query response time, but we also need to determine empirically to what extent the evaluated algorithms are sound and complete statistically. In order to do this, we need to test a suitable and large enough sample of queries while for comparing complete and sound reasoning systems usually a few complex queries suffice. This poses the question, however, which of the potentially infinitely many queries should be used for the testing. Obviously, we have to restrict our attention to a finite set, but two general problems arise: It is not possible to do unbiased random selections from an infinite set, and many randomly generated queries would simply result in no answer at all.

2. *Execution of benchmarks*

   Even when there is enough resources to create a benchmark, our experience shows that it is non-trivial to conduct the actual measurements and conduct them repeatedly. The problem is caused by the fact that the benchmark suite can contain several thousands of test queries and reasoning intensive complex ontologies. So its execution takes hours, days or weeks of running time to complete. The amount of collected data can easily reach the order of gigabytes, *e.g.,* . approximate intance retrieval would make a large set of new facts explicit in case of ontologies with a large set of instance data. Without a mechanism for automated execution and collection of benchmark data, the task is on the verge of possibility.

3. *Evaluation of benchmarks*

   The data collected during the automated execution of a benchmark suite are mostly raw numbers, in amount, which prevents direct human analysis. The data must be first processed by a machine and the size and detail reduced to a manageable level. An important requirement during the development of approximate reasoning systems is automated evaluation of the data with respect to previous runs of the benchmark, which should be able to detect changes and alert the developers to pay extra attention to the results. Without automated processing facilities, the benchmark results are merely gigabytes of useless data.

To address these issues, the difficulty of evaluating approximate reasoning algorithms calls for an automated benchmarking framework to help developers in designing and executing benchmarks.

5.5 Requirements For an Automated Benchmarking Framework

This section examines some requirements suggested in [WLLB06, GQPH07, MYQ^{+}06] for designing benchmarks for the evaluation of sound and complete ontology reasoning systems which can be also considered for the evaluation of approximate reasoning systems. In selecting these requirements, we focus on those requirements which are important to develop an automated, scalable benchmarking framework. We extend these requirements by identifying several specific requirements which an automated evaluation framework has to meet for the evaluation of approximate reasoning algorithms.

5.5.1 Requirements For Measuring Performance

In [WLLB06, WLL$^+$07], creating OWL benchmarks for ABox-related ontology reasoning services have been discussed and several requirements are suggested. Given the focus of this thesis, the following two requirements are relavent.

(A1) The benchmark should also pinpoint the influence of TBox complexity on ABox reasoning. Thus TBox complexity should gradually be increased. In one setting this increase in complexity should influence the ABox reasoning task while in a separate setting TBox axioms which are unrelated to the actual benchmark should be added. The second setting is to trick the reasoner into switching off optimizations even if this would not have been necessary for the actual reasoning task. A well implemented reasoner should thus continue with optimizations whereas other systems might show a significant decrease in performance.

(A1) Include benchmarks, that comprise TBoxes modeled in a way such that adding explicit knowledge to the ABox also adds large quantities of implicit knowledge (*e.g.,* transitive properties). This is to reveal the possibly negative influences of materialization approaches or maintenance of index structures.

In [GQPH07], a collection of requirements for benchmarking have been proposed. Most of them are borrowed from the requirements used in the field of benchmarking database systems. Analysing the common characteristics between both database systems and knowledge representation systems, the authors identify the following requirements which we have to consider for deriving our requirements.

(B1) The benchmark infrastructure should make it easy to add a system under test (SUT). This involves providing hooks to the benchmkark infrastructure so that one can incorporate new SUTs with the least amount of additional work.

(B2) It should provide meaningful metrics. A variety of metrics are needed in order to capture every aspect of system performance. Furthermore, this requirement involves supplementing the benchmark with guidelines and reproducible manner.

(B3) The metrics should be able to capture the extent to which the SUTs return the expected results to a query.

In addition to the above requirements mainly concerning performance measurement, measuring soundness and completeness poses demands that are not answered by conventional benchmarking frameworks, and thus leading to the need for additional requirements. In order to derive these requirements, we analysed the difficulty in depth, particularly, in case of benchmarks with expressive ontologies with large data sets.

5.5.2 Requirements For Measuring Soundness and Completeness

The principal goals of an automated benchmarking execution framework are (1) to provide an infrastructure for accumulating challenging benchmarks, (2) to facilitate executing OWL reasoners and approximate reasoners under the same conditions, guaranteeing reproducible and reliable performance results, and (3) to measure query soundness and completeness. We suggest the following requirements for designing an automated benchmarking framework.

(R1) In some cases, existing benchmark ontologies are not sufficient to highlight the strengths and weaknesses of approximate reasoning systems; therefore various ontology generation methods for artificial ontologies as well as population methods to make TBoxes more complex are needed. Such methods help to make a better study on the performance gain and effectiveness of approximate reasoning systems.

(R2) Various ABox population tools are required to generate realistic data sets as well as random and repeatable data.

(R3) It is important to automatically generate test queries needed for measuring completeness and soundness, and store them in a proprietary format.

(R4) A well-defined benchmark suite allow reducing the cost of the experimentation and the comparison of the results obtained by different groups of people. It may include various tools involved in benchmarking. It can also include other benchmarks that are to be executed consecutively, and thus becoming more and more complex. To create a complex benchmark suite and control its workflow, a configuration describing the workflow is required.

(R5) In benchmarking approximate reasoning systems, a very large amount of data such as inferred ABox assertions and other experimental results are produced. In order to manage such experimental data, a flexible storage solution is needed.

(R6) A benchmark may fail for various reasons, *e.g.,* when the benchmarked application crashes due to a lack of memory or other resources. In such cases, a benchmark framework ought to make it possible to recover previous computed experimental results and to continue performing the benchmark from the interruptted point, in order to run a benchmark with test queries which require intensive computation.

(R7) The characteristics of the ontology under test is important to explain the effect of an approximate reasoning algorithm. Therefore, it is valuable to have an metrics tool which enables to inspect the structure of benchmark ontologies.

(R8) An experiment report provides technical details on the experimental data and results on the experimentation while the benchmarking report is intended to provide an understandable summary of the benchmarking process carried out. The benchmarking report must be written taking into account that the document can show different perspectives of the results by presenting in appropriate formats like in graphics or table sheets. It is in general not clear how to present the evaluated data the most. It is desireable to make the reporting process automated as much as possible.

(R9) A frequent question related to measuring the degree of soundness and completeness is to explore and explain why the answer of an approximate reasoning algorithm differs from the correct one. In case of dealing with large and complex knowledge bases, this inspection task is extremely difficult and time-consuming, yet, in many cases, it is impossible to derive the right information to explain an approximate answer. Therefore, tool support is valuable for explaning approximate answers.

(R10) The architecture of a benchmarking framework should not be tied to any specific benchmark or reasoning system. It should be modular, extendable and capable of embedding extern tools which contribute to a benchmark.

It has been revealed that the existing approaches above mentioned mainly focus on the design of benchmarks. We recognise the difficulty in executing benchmarks for OWL reasoners as well as approximate reasoners with the respect to soundness and completeness measurement, and thus our focus is not only the design of benchmarks, but also the execution of benchmarks and the analysis of experimental results. A concrete benchmarking framework, which shall meet the requirements outlined above, is described in Section 9.5, Chapter 9.

5.6 Conclusions

Approaches to approximate reasoning tackle the problem of scalability of deducing implicit knowledge. Especially if this is done on the basis of large-scale knowledge bases or even the whole Web, often the restriction to 100% correctness has to be abandoned for complexity reasons, in particular if quick answers to posed questions are required. Anytime algorithms try to fulfill both needs (speed and correctness) by providing intermediate results during runtime and continually refining them.

This chapter has provided solid mathematical foundations for the assessment and comparison of approximate reasoning algorithms with respect to correctness, runtime and anytime behaviour. We are confident that this general framework can serve as a means to classify algorithms with respect to their respective characteristics and help in deciding which algorithm best matches the demands of a concrete reasoning scenario.

In most practical cases, it will be unfeasible or even impossible to measure the whole input space as it will be too large or even infinite. That is where statistical considerations come into play: one has to identify and measure representative samples of the input space. The first part of this is far from trivial: for fixed settings with frequently queried knowledge bases, such a sample could be determined by protocolling the actually posed queries over a certain period of time. Another way would be to very roughly estimate a distribution based on plausible arguments. Respective heuristics would be: (1) the more complex a query the more unlikely, (2) queries of similar structure are similarly frequent resp. likely, (3) due to some bias in human conceptual thinking, certain logical connectives (*e.g.,* conjunction) are preferred to others (*e.g.,* disjunction, negation) which also gives an opportunity to estimate a query's frequency based on the connectives it contains. Admittedly, those heuristics are still rather vague and more thorough research is needed to improve reliability of such estimates.

In general, intelligent combination of several approximate algorithms with different soundness/completeness properties (as well as being specialised to certain logical fragments) can increase speed and might help avoid heavy-weight reasoning in cases. We are confident, that this idea can be easily generalised to reasoning tasks other than instance retrieval. Obviously, this strategy comes with an immediate opportunity of parallelisation even if the single algorithms have to be treated as black boxes. Hence, this approach could also be conceived as a somewhat exotic approach to distributed reasoning.

Towards a concrete implementation of the present abstract framework, a number of requirements have been derived which are essential to develop an automated benchmarking framework for approximate instance retrieval algorithms. The implementation of this concrete framework will be described in Chapter 9.

Chapter 6

Knowledge Compilation

Due to the inherent computational complexity of reasoning with expressive ontologies with a large data set, it is to be expected that some application settings will defy even the most sophisticated approaches for achieving sound and complete scalable algorithms. Recently, there has been a growing trendency towards addressing the inherent computational complexity of many reasoning problems by compiling knowledge bases and/or queries into restricted forms where query answering is easier than in the original language. Such methods are very useful in practice when a knowledge base does not change over time.

This chapter presents the concept of a knowledge compilation method SCREECH which allows for various Horn transformations of disjunctive datalog. This chapter is structured as follows: Section 6.1 introduces the general concept of addressing the scalability of reasoning in the context of knowledge compilation. Section 6.3 details the SCREECH approach while Section 6.4 describes its variants and their characteristics. Section 6.5 presents a comprehensive experimental analysis conducted with well-known benchmarking ontologies.

6.1 Principle Idea of Knowledge Compilation

Logic has been used to formalize knowledge representation and reasoning in several areas of Artificial Intelligence, such as planning, model-based diagnosis, belief revision and reasoning about actions. The main argument against the use of logic has always been the high computational complexity involved. For instance, solving a problem using a sound and complete procedure for logical entailment even within propositional logic is a NP-complete problem. The data complexity of satisfiability checking in $\mathcal{SHIQ}(\mathbf{D})$ is NP-complete [HMS05].

Computational efficiency is a central concern in the design of knowledge representation systems. In order to obtain computationally efficient representation systems it has been suggested that one should restrict the expressive power of the knowledge representation language or use an incomplete inference mechanism. In the first approach, decreasing the expressive power usually renders the language too limited for practical applications, and leaves unanswered the question of what to do with information that *cannot* be represented in the restricted form. The second approach involves either resource-bounded reasoning or the introduction of a non-traditional semantics. Resource-bounded reasoning is concerned with the construction of intelligent systems that can operate in real-time environments under uncertainty and limited computational resources such as time, memory, or information. That is, in resource-bounded reasoning, inference is limited by bounding the number of inference steps performed by the inference procedure. It therefore becomes difficult to characterize exactly what can and cannot be inferred, that is, the approach lacks a "real" semantics. Moreover, no information is provided if a proof cannot be found within the time bound.

Knowledge compilation (KC) [SK91, SK96] is a family of approaches for addressing the intractability of reasoning problems in logic-based formalisms. In contrast to the approaches restricting the expressive power, it allows the knowledge base to be specified in a general, unrestricted knowledge representation language. The central idea of Knowledge compilation is to transform a knowledge base with respect to which reasoning is intractable into one or more approximate or equivalent knowledge bases with respect to which reasoning can be tractable, or at least more efficient. The central idea is to invest time and space in an extra preprocessing effort which will later substantially speed up query answering at runtime. In particular, a KC method is useful for applications of expressive knowledge bases where the knowledge base remains unchanged over a long period of time and is used to answer many queries.

Knowledge compilation is like pre-processing that is quite common in computer science. For example, compilers usually optimize object code or a graph can be preprocessed to obtain a data structure that allows for a fast node reachability. While preprocessing techniques in computer science are usually implemented for problems, that are already solvable in polynomial time, knowledge compilation deals with reasoning problems that are often NP-hard. In knowledge compilation query answering is divided into two phases:

- In the first phase the knowledge base is preprocessed by translating it into an appropriate data structure, which allows for more efficient query answering. This phase is also called off-line reasoning.

- In the second phase, the data structure, which results from the previous phase, is used to answer the query. This phase is also called on-line reasoning.

The challenge here is to minimize the size of the compiled knowledge base and compilation time, while still ensuring that every query of interest can be answered in polynomial time. However it is not always possible to develop a method to make online-reasoning more efficient without giving up the soundness and completeness. KC may be exact or approximate, depending on whether all queries can be answered

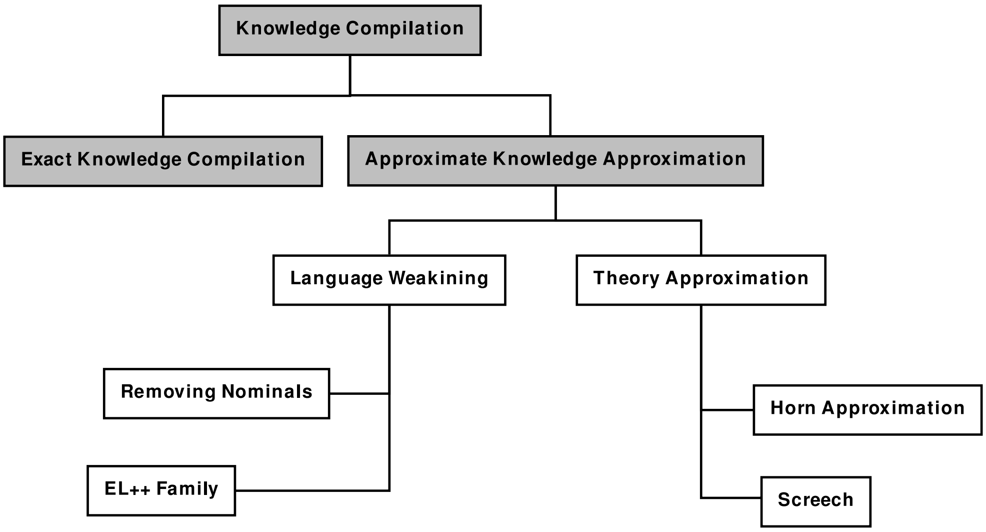


Figure 6.1: A classification of knowledge compilation methods

tractably, or only a subset thereof. In this regard, an important aspect in the development of a knowledge compilation is to study its characateristics, *i.e.,* to ensure whether a transformation causes breaking the soundness and completeness of the original knowledge base.

6.2 A Classification of Knowledge Compilation Methods

In the context of our work, we will only be able to discuss a small fraction of the wide variety of knowledge compilation techniques [CD97, DM02]. Before digging any deeper into the matter, we will attempt to provide a classification of knowledge compilation methods as overview.

Over the past years, a number of knowledge compilation methods have been intvestigated. Figure 6.1 sketches a rough taxonomy of knowledge compilation methods. Generally, knowledge compilation methods can be divided in two basic classes such as exact compilation and approximate compilation.

6.2.1 Exact Knowledge Compilation

Exact knowledge compilation (EKC) refers to methods that exactly translate a knowledge base into another form in such a way that the reasoning later provides the same answers as reasoning in the original knowledge base, *i.e.,* the knowledge bases are logically equivalent.

Cadoli in [SC95] proposed exact knowledge compilation in three main methods: (1) use of prime implicants[1] or prime implicates[2]; (2) add to the knowledge base only those prime implicates that make any deduction possible by unit resolution; (3) use prime implicates with respect to a tractable theory. We do not elaborate on this subject as the focus of this thesis is on approximation. Interested reader may refer to [CD97].

6.2.2 Approximate Knowledge Compilation

Approximate knowledge compilation (AKC) refers to methods that translate a knowledge base into a form that approximates the original knowledge base. For some important reasoning problems, it is not always possible to develop exact compilation techniques. Furthermore, it is not always possible to make on-line reasoning more efficient in all cases without approximation. As approximate knowledge compilation, two classical approaches have been developed for addressing the computational hardness of reasoning problems, which are *language restriction* and *theory approximation*.

Language Weakening. The expressive power of the language used to represent knowledge may be restricted by neglecting some expensive language constructors. The resulting language cannot allow for all problems in a domain of interest, but, once a problem is presented using such a language, it shall be possible to solve the problem more efficiently. In general, in order to obtain a computationally efficient representation system one either restricts the expressive power of the knowledge representation language or one uses an incomplete inference mechanism.

In the first approach, the representation language is often too limited for practical applications [CD97]. In both the description logic and semantic web community, it is common to build tractable reasoning procedures for certain fragments of description logics or the OWL language, *i.e.,* to identify certain fragments with polynomial data complexity at least. As an example of language weakening, the description logic EL++ constitutes a tractable knowledge representation formalism [BBL05]. Neglecting some expressive concept constructors such as disjunction and universal restriction and number restriction, the description logic EL++ is an important DL fragment in which reasoning can be performed in polynomial time. Moreover, reasoning over EL TBoxes with arbitrary (possibly cyclic) GCIs is polynomial (even with additional constructs), while this is not the case for $\mathcal{AL}$ (the basic description logic), for which the same problem is EXPTIME-complete. Another example of language weakening is to eliminate the nominal constructor in the OWL language. The nominal constructor allows to define a concept by finite enumeration of its elements. For example, the atomic concept *WineColor* can be defined, using nominals, as follows: $WineColor = \{red, white\}$ where the elements of the enumeration are individuals in the knowledge base. From a logical point of view, the nominal constructor transforms the individual name i into the concept description $\{I\}$, which is evaluated, by every

[1] Logically weakest terms implying a formula.

[2] Logically strongest clausel consequences of a formula.

model-theoretic interpretation, to a singleton set with i as its only element. In other words, ABox assertions can affect concept satisfiability and TBox classification.

As a consequence, ontologies with nominals in the TBox and large number of instances in the ABox are likely to compromise the performance of DL reasoners. To provide praticable reasoning with large ABoxes, nominals have been partially approximated in DL reasoners by treating them as pair-wise disjoint atomic concepts, commonly called *pseudo-nominals*. For instance, KAON2 cannot currently handle ontologies with nominals; thereofore such ontologies are processed by removing nominals before reasoning with KAON2. This technique of removing nominals is known to lead to incorrect inferences in some cases, and thus one can consider it as a language weakening technique.

Theory approximation. One can give up the soundness or completeness in the answer to the original reasoning problem. This is means that either certain information is lost in the off-line reasoning phase, or that the theory is compiled into an equivalent theory and the soundness or incompletenes in the answer is lost in the on-line reasoning phase.

Horn approximation proposed by Selman and Kautz [SK91] is a method to approximate knowledge bases. It shows how a propositional theory can be compiled into a tractable form – consisting of a set of Horn clauses – that guarantees polynomial-time inference. As a logically equivalent set of Horn clauses does not always exist, the basic idea of this method is to bound a set of models of the original theory from below (*i.e.,* complete) and from above (*i.e.,* sound). For this, they proposed to use Horn lower-bound and Horn upper-bound to approximate the original theory.

Let Σ be a set of clauses (the original theory), the sets Σ_{lb} and Σ_{ub} of Horn clauses are respectively a Horn lower-bound and a Horn upper-bound of Σ iff $\mathcal{M}(\Sigma_{lb}) \subseteq \mathcal{M}(\Sigma) \subseteq \mathcal{M}(\Sigma_{ub})$, or, equivalently, $\Sigma_{lb} \models \Sigma \models \Sigma_{ub}$. This says the Horn lower-bound is logically stronger than the original theory and the Horn upper-bound is logically weaker than it. Instead of using any pair of bounds to characterize the original theory, the best possible bounds are used. This leads to a greatest Horn lower bound and a least Horn upper bound.

A Horn lower-bound $\mathcal{M}(\Sigma_{glb})$ is a greatest Horn lower-bound iff there is no set $\Sigma^{'}$ of Horn clauses such that $\mathcal{M}(\Sigma_{lb}) \subset \mathcal{M}(\Sigma^{'}) \subseteq \mathcal{M}(\Sigma)$. A Horn upper-bound $\mathcal{M}(\Sigma_{lub})$ is a least Horn upper-bound iff there is no set Σ of Horn clauses such that $\mathcal{M}(\Sigma)$ $\subseteq \mathcal{M}(\Sigma^{'}) \subset \mathcal{M}(\Sigma_{lub})$. It is shown that each theory has a unique LUB (up to logical equivalence), but can have many different GLBs. In this way, inference can be approximated by using the Horn GLBs and Horn LUBs. The inference could be unsound or incomplete, but it is always possible to spend more time and use a general inference procedure on the original theory.

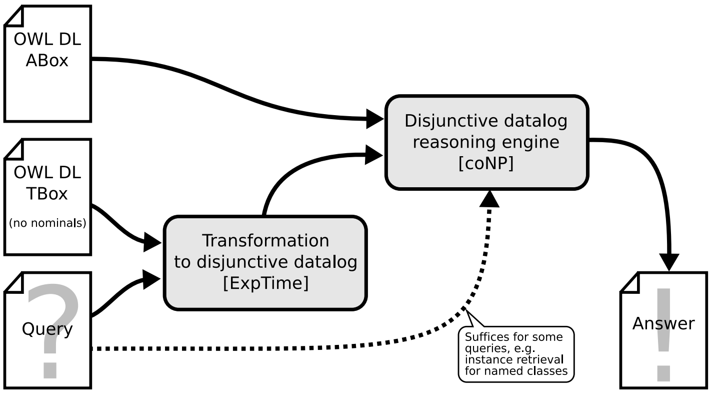

OWL DL ABox
OWL DL TBox (no nominals)
Query
Transformation to disjunctive datalog [ExpTime]
Disjunctive datalog reasoning engine [coNP]
Suffices for some queries, e.g. instance retrieval for named classes
Answer

Figure 6.2: KAON2 approach to reasoning

6.3 Approximate Reasoning with SCREECH

In the previous section, we have given an overview of knowledge compilation methods, which can be used to address the high computational complexity of logical inference. The central idea behind knowledge compilation is to tranform an expressive knowledge base into a restricted form with respect to which reasoning can be tractable, or at least more efficient. Furthermore, we have briefly introduced the underlying ideas of several knowledge compilation methods.

In this section, we present our approximate knowledge compilation method, called Screech. The SCREECH approach for instance retrieval is based on the fact that data complexity is polynomial for non-disjunctive datalog, while for $\mathcal{SHOIN}$ it is coNP complete even in the absence of nominals [HMS05]. SCREECH utilises the KAON2 algorithms, but rather than doing (expensive) exact reasoning over the resulting disjunctive datalog knowledge base, it does approximate reasoning by treating disjunctive rules as if they were non-disjunctive ones, *i.e.,* the disjunctive rules are approximated by Horn rules. In the rest of this section the SCREECH approach is described in full detail. We start introducing the underlying system KAON2.

6.3.1 The KAON2-Transformation

Reasoning with KAON2 is based on special-purpose algorithms which have been designed for dealing with large ABoxes. They are detailed in [Mot06] and we present a birds' eyes perspective here, which suffices for our purposes. The underlying rationale of the algorithms is that algorithms for deductive databases have proven to be efficient in dealing with large numbers of facts. The KAON2 approach utilises this by transforming OWL-DL ontologies to disjunctive datalog, and by the subsequent application of the mentioned and established algorithms for dealing with disjunctive datalog.

A birds′ eyes perspective on the KAON2 approach is depicted in Figure 6.2. KAON2 can handle $\mathcal{SHIQ}(\mathbf{D})$ ontologies. The TBox, together with a query are processed by the sophisticated KAON2-transformation algorithm which returns a disjunctive datalog program. This, together with an ABox, is then fed into a disjunctive datalog reasoner which eventually returns an answer to the query.

In some cases, *e.g.,* when querying for instances of named classes, the query does not need to be fed into the transformation algorithm but instead needs to be taken into account only by the datalog reasoner. This allows to compute the disjunctive datalog program offline, such that only the disjunctive datalog engine needs to be invoked for answering the query. All experiments we report on have been performed this way, *i.e.,* they assume an offline transformation of the TBox prior to the experiments.

The program returned by the transformation algorithm is in general not logically equivalent to the input TBox. The exact relationship is given below in Theorem 6.1 due to [Mot06]. Note that statement (b) suffices for our purposes. It also shows that the KAON2 datalog reasoning engine can in principle be replaced by other (sound and complete) reasoning engines without changing the results of the inference process.

Theorem 6.1. *Let KB be a $\mathcal{SHIQ}(\mathbf{D})$ TBox and $D(KB)$ be the datalog output of the KAON2 transformation algorithm on input KB. Then the following claims hold.*

(a) KB is unsatisfiable if and only if $D(KB)$ is unsatisfiable.

(b) $KB \models \alpha$ if and only if $D(KB) \models \alpha$, where α is of the form $A(a)$ or $R(a,b)$, for A a named concept and R a simple role.

(c) $KB \models C(a)$ for a nonatomic concept C if and only if, for Q a new atomic concept, $D(KB \cup \{C \sqsubseteq Q\}) \models Q(a)$.

6.3.2 From Disjunctive Datalog to Horn Clauses

We will first describe the basic variant of SCREECH, which was introduced in [HV05], and which we call SCREECH-ALL here. SCREECH-ALL is complete, but may be unsound in cases. Its data complexity is polynomial. Two other variants of SCREECH, SCHREECH-NONE and SCREECH-ONE, will be described in Section 6.4.

SCREECH-ALL uses a modified notion of *split program* [SI94] in order to deal with the disjunctive datalog. Given a disjuctive rule

$$H_1 \vee \cdots \vee H_m \leftarrow A_1, \ldots, A_k$$

as an output of the KAON2 transformation algorithm, the *derived split rules* are defined as:

$$H_1 \leftarrow A_1, \ldots, A_k \qquad \ldots \qquad H_m \leftarrow A_1, \ldots, A_k.$$

For a given disjunctive program P its *split program* P' is defined as the collection of all split rules derived from rules in P.

It can be easily shown that for instance retrieval tasks, the result obtained by using the split program instead of the original one is complete but may be unsound. As the following proposition shows, this is even the case if all integrity constraints, *i.e.,* rules of the form

$$\leftarrow B_1, \ldots, B_n$$

are removed from the split program.

Proposition 6.2. *Consider a $\mathcal{SHIQ}(\mathbf{D})$ knowledge base KB that is logically consistent, let $D(KB)$ denote a disjunctive datalog program obtained by applying KAON2 to KB, and let P be the logic program obtained from $D(KB)$ by* SCREECH-ALL. *Then P has a least Herbrand model which satisfies any atomic formula that is true in some minimal Herbrand model of $D(KB)$.*

Especially, P entails all atomic formulae that are true in all (minimal) models of $D(KB)$, i.e., SCREECH-ALL *is complete for instance retrieval on consistent $\mathcal{SHIQ}(\mathbf{D})$ knowledge bases.*

Proof. First, note that we can restrict to propositional programs obtained as the (finite) ground instantiations of the relevant datalog programs. Hence it suffices to consider propositional models.

The fact that P has a least Herbrand model is a standard conclusion from the fact that P is a definite logic program. To show the rest of the claim, consider any minimal Herbrand model $\mathcal{M}$ of the ground instantiation of $\mathsf{D}(KB)$ (note that KB has some Herbrand model by consistency, and that some of those must be minial since only finitely many ground interpretations exist). Define a ground program $Q_{\mathcal{M}}$ as follows: $Q_{\mathcal{M}} = \{H_i \leftarrow B_1 \wedge \ldots \wedge B_m \in P \mid \mathcal{M} \models B_1 \wedge \ldots \wedge B_m$ and $\mathcal{M} \models H_i, 1 \leq i \leq n)\}$.

We claim that $Q_{\mathcal{M}}$ is a definite program with least Herbrand model $\mathcal{M}$. Clearly $Q_{\mathcal{M}}$ is definite (thus has some least Herbrand model), and has $\mathcal{M}$ as a Herbrand model. But obviously any Herbrand model of $Q_{\mathcal{M}}$ that is strictly smaller than $\mathcal{M}$ would also satisfy all rules of $\mathsf{D}(KB)$, thus contradicting the assumed minimality of $\mathcal{M}$. Now clearly $Q_{\mathcal{M}}$ is a subset of the screeched program P, and hence any Herbrand model of P must be greater or equal to the least Herbrand model $\mathcal{M}$ of $Q_{\mathcal{M}}$. Since $\mathcal{M}$ was arbitrary, this shows the claim. □

In the following we intruduce the algorithms for the SCREECH approach. Split function defined in Algorithm 1 derives split rules from each disjuctive rule. Here, a disjunctive rule is given as a pair of two lists and is denoted $\langle H, B \rangle$. H is a list of negative literals while B is a list of possitive literals. The algorithm accepts a disjunctive rule $\langle H, B \rangle$ and generates a list of Horn rules.

Horn rules are generated from a disjunctive rule in such a way that each literal in the list of H is combined with the body of the given disjunctive rule, as shown in lines 4 to 7. In the algorithm a temporary list and its access function *addListItem* are used to collect the Horn rules are used. Refer to Appendix 13.1 for specific details of abstract data type list and its access functions.

Algorithm 1: *split*($\langle H, B \rangle$)

```
Input: ⟨H, B⟩: a disjunctive rule as a pair where H is a list of negative literals
and B is a list of positive literals
Result: P_Horn: a list of Horn rules
Data: i: counter variable
begin
    $i \longleftarrow len(H) - 1$;
    $P_{Horn} \longleftarrow nil$;
    while $i > 0$ do
        $addListItem(P_{Horn}, \langle H[i], B \rangle)$;
        $i \longleftarrow i - 1$;
    return $P_{Horn}$;
end
```

Screech-all which generates a split program from $\mathsf{D}(KB)$ is defined in Algorithm 2 using the *split* function. The algorithm assumes that a Horn program is a list of Horn clauses and a disjunctive program is a list of (disjunctive) clauses. The algorithm generates a Horn program from a given disjunctive program as follows. First, an empty list P_{Horn} is created as in line 3. Then, it applies the algorithm 1 to each disjunctive rule in $\mathsf{D}(KB)$, and adds the corresponding split clauses to P_{Horn}. As given in lines 9 to 10, the ground Horn clauses contained in the original program are inserted into P_{Horn}, except for integrity constrains.

In practice, the Screech-all approach can be applied for approximate ABox reasoning for $\mathcal{SHIQ}$ as follows:

1. Apply transformations as in the KAON2 system in order to obtain a negation-free disjunctive program.

2. Obtain the split program by applying the Algorithm 2.

3. Do reasoning with the split program, e.g. using the KAON2 datalog engine.

Given a TBox KB, the split program obtained from KB by steps 1 and 2 is called the *screeched version* of KB. The first two steps can be considered to be preprocessing steps for setting up the intensional part of the database. ABox reasoning is then done in step 3. The resulting approach is complete with respect to OWL-DL semantics and data complexity is polynomial.

Since our approach is based on KAON2, we have so far described the Screech approach for $\mathcal{SHIQ}$ which covers a significant portion of OWL-DL. However, it is possible to apply the Screech approach for expressive OWL-DL knowledge bases by removing nominals.

Algorithm 2: *screech_All*$(\mathsf{D}(KB))$

```
Input: $\mathsf{D}(KB)$: a disjunctive datalog program
Data: $i$: counter variable
Result: $P_{Horn}$: a Horn program
begin
    $i \longleftarrow len(\mathsf{D}(KB)) - 1$;
    $P_{Horn} \longleftarrow nil$;
    while $i \geq 0$ do
        $rule \longleftarrow \mathsf{D}(KB)[i]$;
        if $isDisjuctiveRule(rule)$ then
            $H_{split} \longleftarrow split(rule)$;
            $appendList(P_{Horn}, H_{split})$;
        else if $\neg(isIntegrityConstraint(rule))$ then
            $addListItem(P_{Horn}, rule)$;
        $i \longleftarrow i - 1$;
    return $P_{Horn}$;
end
```

6.3.3 A Simple Example

We demonstrate the approach on a simple OWL-DL ontology. It contains only a class hierarchy and an ABox, and no roles, but this will suffice to display the main issues.

The ontology is shown in Figure 6.3, and its intended meaning is self-explanatory. Note that the axiom in line 4 is translated into the four clauses:

$$\begin{aligned} \texttt{luxembourgian}(x) \vee \texttt{dutch}(x) \vee \texttt{belgian}(x) &\leftarrow \texttt{beneluxian}(x), \\ \texttt{beneluxian}(x) &\leftarrow \texttt{luxembourgian}(x), \\ \texttt{beneluxian}(x) &\leftarrow \texttt{dutch}(x), \\ \text{and} \qquad \texttt{beneluxian}(x) &\leftarrow \texttt{belgian}(x). \end{aligned} \tag{6.1}$$

Thus, our approach changes the ontology by treating the disjunctions in line (6.1) as conjunctions. Effectively, this means that the rule (6.1) is replaced by the three rules

$$\begin{aligned} \texttt{luxembourgian}(x) &\leftarrow \texttt{beneluxian}(x), \\ \texttt{dutch}(x) &\leftarrow \texttt{beneluxian}(x), \\ \text{and} \qquad \texttt{belgian}(x) &\leftarrow \texttt{beneluxian}(x). \end{aligned}$$

This change affects the soundness of the reasoning procedure. However, in the example most of the ABox consequences which can be derived by the approximation are still correct. Indeed, there are only two derivable facts which do not follow from the knowledge base by classical reasoning, namely dutch(saartje) and luxemburgian(saartje). All other derivable facts are correct.

$$\begin{aligned}
\text{serbian} \sqcup \text{croatian} &\sqsubseteq \text{european}\\
\text{eucitizen} &\sqsubseteq \text{european}\\
\text{german} \sqcup \text{french} \sqcup \text{beneluxian} &\sqsubseteq \text{eucitizen}\\
\text{beneluxian} &\equiv \text{luxembourgian} \sqcup \text{dutch} \sqcup \text{belgian}
\end{aligned}$$

serbian(ljiljana)	serbian(nenad)	german(philipp)	french(julien)
chinese(yue)	german(peter)	german(stephan)	mongolian(tuvshintur)
indian(anupriya)	belgian(saartje)	german(raphael)	chinese(guilin)

Figure 6.3: Example ontology

6.4 Variants of Screech

In SCREECH-ALL, we consider all split clauses of each disjunctive rule. As a result, SCREECH-ALL is complete, but unsound. That means that we will find more individuals which should not belong to a named class when performing instance retrieval. In order to reduce finding wrong individuals, one might think to restrict the set of split clauses derived from a disjunctive rule. In the following we will deal with such restrictions which lead to two other approximations; we here call SCREECH-ONE and SCREECH-NONE.

SCREECH-ONE is defined by replacing each disjuntive rules by *exactly one* of the split rules. This selection can be done randomly, but will be most useful if the system has some knowledge – probably of statistical nature – on the size of the extensions of the named classes. For our example from Section 6.3.3, when considering rule (6.1) we can use the additional knowledge that there are more dutch people than belgians or luxenbourgians, thus this rule is replaced by the single rule

$$\texttt{dutch}(x) \leftarrow \texttt{beneluxian}(x).$$

Following this idea, SCREECH-ONE is defined in Algorithm 4, in such a way that only one Horn rule is selected from the split rule of a disjunctive rule whose head has the most extension. The *extractHornRule* function, which – as the name says – extract a Horn rule, is defined in Algorithm 3. The algorithm uses two variables, *maxInd* and *maxExt*, which are initially assigned to 0 as well as the extension of the first disjunction of the given disjunctive rule. Note that H is assumed to be a list of disjunctions, the extension of each $H[i], (i > 0)$ (denoted as $|H[i]|$) is compared with *maxExt*. In case $|H[i]|$ is more than *maxExt*, both variables will be modified by the assignment to i and $|H[i]|$, as given in lines 7 and 8.

The algorithm for SCREECH-ONE is defined in the analogous way as for SCREECH-ALL. The only difference is that the *extractHornRule* function instead of the split function is used. Note that all integrity constraints after the translation are also removed. The resulting reasoning procedure is neither sound nor complete.

Algorithm 3: *extractHornRule*$(\langle H, B\rangle)$

```
Input: $\langle H, B\rangle$: a disjunctive rule as a pair where $H$ is a list of negative literals
and $B$ is a list of positive literals
Data: $i$: counter variable
Result: A list of Horn rules (clauses)
begin
    $maxInd \longleftarrow 0$;
    $maxExt \longleftarrow |H[0]|$;
    $i \longleftarrow 1$;
    while $i < len(H)$ do
        if $|H[i]| > maxExt$ then
            $maxExt \longleftarrow |H[i]|$;
            $maxInd \longleftarrow i$;
        $i \longleftarrow i + 1$;
    return $(H[maxInd], B)$;
end
```

Algorithm 4: *Screech_One*$(\mathsf{D}(KB))$

```
Input: $\mathsf{D}(KB)$: a disjunctive program
Data: $i$: counter variable
Result: $P_{Horn}$: a Horn program
begin
    $i \longleftarrow len(\mathsf{D}(KB)) - 1$;
    $P_{Horn} \longleftarrow nil$;
    while $i \geq 0$ do
        $rule \longleftarrow \mathsf{D}(KB)[i]$;
        if $isDisjuctiveRule(rule)$ then
            $h \longleftarrow extractHornRule(rule)$;
            $addListItem(P_{Horn}, h)$;
        else if $\neg(isIntegrityConstraint(rule))$ then
            $addListItem(P_{Horn}, rule)$;
        $i \longleftarrow i - 1$;
    return $P_{Horn}$;
end
```

SCREECH-NONE is defined in Algorithm 5 by simply removing all disjunctive rules (and all integrity constraints) after the transformation by the KAON2-algorithm. For the example from Section 6.3.3, this means that rule (6.1) is simply deleted. The resulting reasoning procedure is sound, but incomplete, on $\mathcal{SHIQ}$ knowledge bases. We thus obtain the following result.

Algorithm 5: *Screech_None*($\mathsf{D}(KB)$)

```
Input: $\mathsf{D}(KB)$: a disjunctive program
Data: $i$: counter variable
Result: $P_{Horn}$: a Horn program
begin
    $i \longleftarrow len(\mathsf{D}(KB)) - 1;$
    $P_{Horn} \longleftarrow nil;$
    while $i \geq 0$ do
        $rule \longleftarrow \mathsf{D}(KB)[i];$
        if $\neg(isDisjuctiveRule(rule) \vee isIntegrityConstraint(rule))$ then
            $addListItem(P_{Horn}, rule);$
        $i \longleftarrow i - 1;$
    return $P_{Horn};$
end
```

Proposition 6.3. *Instance retrieval with* SCHREECH-NONE *is sound but incomplete. Instance retrieval with* SCHREECH-ONE *in general is neither sound nor complete.*

Proof. Soundness of SCHREECH-NONE is immediate from the fact that calculations are performed on a subset of the computed clauses, together with monotonicity of the employed datalog variant. For all other claims it is easy to find counterexamples. □

Table 6.1: SCREECH variants and their basic properties

variant	description	sound	complete
SCREECH-ALL	use *all* of the split rules	no	yes
SCREECH-ONE	use *one* of the split rules	no	no
SCREECH-NONE	use *none* of the split rules	yes	no

The properties of SCREECH are summarised in Table 6.1. From a theoretical point of view, it would be satisfying to characterize the described approximations in terms of extremal bounds in certain logical fragments. However, we remark that the unsound screech variants do not yield greatest Horn lower bounds in the sense of [SK91] w.r.t. the disjunctive datalog program, not even if we modify the definition to allow only definite Horn rules. As a counterexample for SCREECH-ALL, consider the program $\{\leftarrow C(a), C(a) \vee C(b) \leftarrow\}$. Its screeched version is $\{C(a) \leftarrow, C(b) \leftarrow\}$, but its greatest lower bound in the sense of [SK91] would be $\{C(b) \leftarrow\}$.

Analogously, we note that SCREECH-ONE yields no greatest lower bound, even if integrity constraints are included (which obviously makes the procedure complete while still being unsound). To see this, consider the program

$$\{C(a) \leftarrow, C(b) \leftarrow, \leftarrow A(a), \leftarrow B(b), A(x) \vee B(x) \leftarrow C(x)\}.$$

Its (extended) SCREECH-ONE versions are

$$\{C(a) \leftarrow, C(b) \leftarrow, \leftarrow A(a), \leftarrow B(b), A(x) \leftarrow C(x)\}$$

and

$$\{C(a) \leftarrow, C(b) \leftarrow, \leftarrow A(a), \leftarrow B(b), B(x) \leftarrow C(x)\}$$

, but its greatest lower bound would be $\{C(a) \leftarrow, C(b) \leftarrow, B(a) \leftarrow, A(b) \leftarrow\}$.

Combining the three variants SCREECH-ALL, SCREECH-ONE and SCREECH-NONE, we define the overall SCREECH approach in Algorithm 6.

Algorithm 6: *Screech*(KB, C, t)

```
   Input: KB: a $\mathcal{SHIQ}$ knowledge base
   Input: C: a named class
   Input: t: the required SCREECH approximation
   Result: Inv: a set of individuals
1  begin
2      $\mathsf{D}(KB) \longleftarrow tboxTranslation(KB)$;
3      $P_{Horn} \longleftarrow 0$;
4      switch $t \in \{screech\_one, screech\_one, screech\_all\}$ do
5          case t is SCREECH-ALL
6              $P_{Horn} \longleftarrow$ SCREECH-ALL$(\mathsf{D}(KB))$;
7          case t is SCREECH-ONE
8              $P_{Horn} \longleftarrow$ SCREECH-ONE$(\mathsf{D}(KB))$;
9          case t is SCREECH-NONE
10             $P_{Horn} \longleftarrow$ SCREECH-NONE$(\mathsf{D}(KB))$;
11
12     return $Inv \longleftarrow \{x \mid P_{Horn} \vdash_{datalog} C(x)\}$;
13 end
```

As given in line 2, the given knowledge base *KB* is transformed into a disjunctive datalog program $\mathsf{D}(KB)$ by the *tboxTranslation* function[3]. Then, according to

[3] This is the KAON2 TBox transformation.

a given approximation variant $t \in \{screech\_one, screech\_one, screech\_all\}$, the algorithm compiles $\mathsf{D}(KB)$ into a split program. Finally, it runs a datalog engine, namely the KAON2-datalog, on the resulting split program to compute the approximate extension of a named class C from a $\mathcal{SHIQ}$ knowledge base KB.

Expected results

Prior to performing our experiments – which we will report in Section 6.5 – we formulated the expected outcome from the different variants of SCREECH.

- SCREECH-ONE – assuming the mentioned knowledge about the size of the extensions of atomic classes – compared to SCREECH-ALL should show overall *less* errors for some suitable knowledge bases. We also expected SCREECH-ONE to be quicker than SCREECH-ALL.
- SCREECH-NONE should be quicker than SCREECH-ALL and SCREECH-ONE. We expected that the number of errors should be comparable with SCREECH-ALL, but more severe than SCREECH-ONE.

We furthermore expected, that the parallel execution of the two variants SCREECH-ALL and SCREECH-NONE should help to determine *exact* answers in some cases quicker than using the KAON2 datalog reasoner. This expectation is based on the following fact: If the extensions of some class C as computed by SCREECH-ALL and SCREECH-NONE are *of the same size*, then the computed extensions are actually correct (sound and complete) with respect to the original knowledge base.

6.5 Experimental Results

An approximate reasoning procedure needs to be evaluated on real data from practical applications. Handcrafted examples are of only limited use as the applicability of approximate methods depends on the structure inherent in the experimental data.

So we based our tests on some popular publicly available ontologies used for the performance evaluation of KAON2. The information about the structure of the ontologies we used is summarized in Table 6.2.

In some cases we had to cautiously modify them in order to enable KAON2 to perform reasoning tasks on them, but the general approach was to first use KAON2 for transforming the TBoxes to disjunctive datalog. Also offline, a screeched version of the TBox was produced. We then invoked the KAON2 disjunctive datalog engine on both the resulting disjunctive datalog program and on the screeched version, to obtain a comparison of performance. For all our experiments, we used a T60p IBM Thinkpad with 1.9GB of RAM, with the Java 2 Runtime Environment, Standard Edition (build 1.5.0_09-b03). The maximum memory amount allowed to Java was set to 512MB for each experiment.

Table 6.2: Statistics of test ontologies

ontology	$C \sqsubseteq D$	$C \equiv D$	$C \sqcap D \sqsubseteq \bot$	functional	domain	range	expressivity	$C(a)$	$R(a,b)$
DOLCE	203	27	42	2	253	253	SHIN(D)	0	0
GALEN	3237	699	0	133	0	0	ALEHIF	0	0
WINE	126	61	1	6	6	9	SHOIN	10127	10086
VICODI	193	0	0	0	10	10	ALHI	84710	183555
SEMINTEC	55	0	113	16	16	16	ALCOIF	89705	236240

Table 6.3: Statistics of the disjunctive datalog programs obtained by KAON2-Transformation and the screeched versions

ontology	**variant**	**integrity constraints**	**disjunctive rules**	**split rules**	**split program**
GALEN	SCREECH-ALL	149	52	105	1737
	SCREECH-ONE	149	52	52	1684
	SCREECH-NONE	149	52	0	1632
DOLCE	SCREECH-ALL	189	71	178	1692
	SCREECH-ONE	189	71	71	1585
	SCREECH-NONE	189	71	0	1515
WINE	SCREECH-ALL	3	24	48	575
	SCREECH-ONE	3	24	24	551
	SCREECH-NONE	3	24	0	527
SEMINTEC	SCREECH *	113	0	0	104
VICODI	SCREECH *	0	0	0	223

Results in a nutshell

We performed comprehensive experiments with GALEN, WINE, DOLCE, and SEMINTEC. Before we report in more detail, we list a summary of the results.

- SCREECH-ALL shows an average speedup in the range between 8% and 67%, depending on the queried class and the ontology under consideration, while 38% to 100% of the computed answers are correct. Most interestingly, a higher speedup usually seemed to correlate with less errors.

- SCREECH-ONE compared to SCREECH-ALL has overall *less* errors. In most cases, all correct class members are retrieved. Runtime is similar to SCREECH-ALL.

- SCREECH-NONE compared to SCREECH-ALL shows similar run-time. In most cases, the extensions are computed *correctly* – with the exception of WINE, for which we get 2% missing answers.

- Running SCREECH-ALL and SCREECH-NONE in parallel and comparing the results, allows the following: If the computed extensions are of the same size, then we know that all (and only correct) class members have been found. This is the case for more than 62.1% of all classes we computed.

GALEN

We first report on our experiments with the OWL DL version of the GALEN Upper Ontology.[4] As it is a TBox ontology only, GALEN's 175 classes were randomly populated with 500 individuals. GALEN does not contain nominals or concrete domains. GALEN has 673 axioms (the population added another 500).

As shown in Table 6.3 on page 70, the disjunctive datalog program obtained by the TBox transformation contained ca. 1781 disjunctive datalog rules, 52 of which contained disjunctions. The split program for SCREECH-ALL resulted in a knowledge base with 1737 rules by removing all 149 integrity constraints and complying the 52 disjunctive rules into 105 Horn rules. The split program for SCREECH-ONE resulted in a knowledge base with 1684 rules while the split program for SCREECH-NONE resulted in a knowledge base with 1632 rules. Note that the exact numbers of the rules obtained by the TBox transformation differ slightly on different runs, as the KAON2 translation algorithm is non-deterministic. Finally, the knowledge bases (split programs) compiled by SCREECH-ALL, SCREECH-ONE and SCREECH-NONE were used to query all named classes for their extensions by running the KAON2 datalog engine. Furthermore, the original knowledge base was used to query the same queries by running the KAON2 datalog engine. The purpose of this experiment is to measure the running times required to answer the queries and their extensions.

Table 6.4: Summary of the three SCREECH versions on GALEN

variant (v_i)	**miss**	**corr**	**more**	**time (v_i)**	**time (KAON2)**	**f-meas**	**corr.class**	$\frac{t_{screech}}{t_{kaon2}}$
SCREECH-ALL	0	5187	465	255132	1007527	0.957	0.78	0.25
SCREECH-ONE	5	5182	134	277009	1007527	0.987	0.98	0.27
SCREECH-NONE	10	5177	0	244994	1007527	0.999	0.78	0.24

A summary of the results can be seen in Table 6.4. Here, *miss* indicates the elements of the extensions which were *not* found by the approximation, *corr* indicates those which were correctly found, and *more* indicates those which were incorrectly computed to be part of the extension. *time* gives the runtime (in ms) for the respective SCREECH version, while *time(KAON2)* gives the runtime (in ms) using the disjunctive rules. *f-meas* is the f-measure known from information retrieval, computed as $(2 \cdot \text{precision} \cdot \text{recall})/(\text{precision} + \text{recall})$ with $\text{precision} = \text{corr}/(\text{corr} + \text{more})$ and $\text{recall} = \text{corr}/\text{number of actual instances}$, *corr.class* gives the fraction of classes for

[4] http://www.cs.man.ac.uk/$\sim$rector/ontologies/simple-top-bio/ [accessed 2009-5-11]

which the extension was computed correctly, and *time/KAON2* is the ratio between time and KAON2.

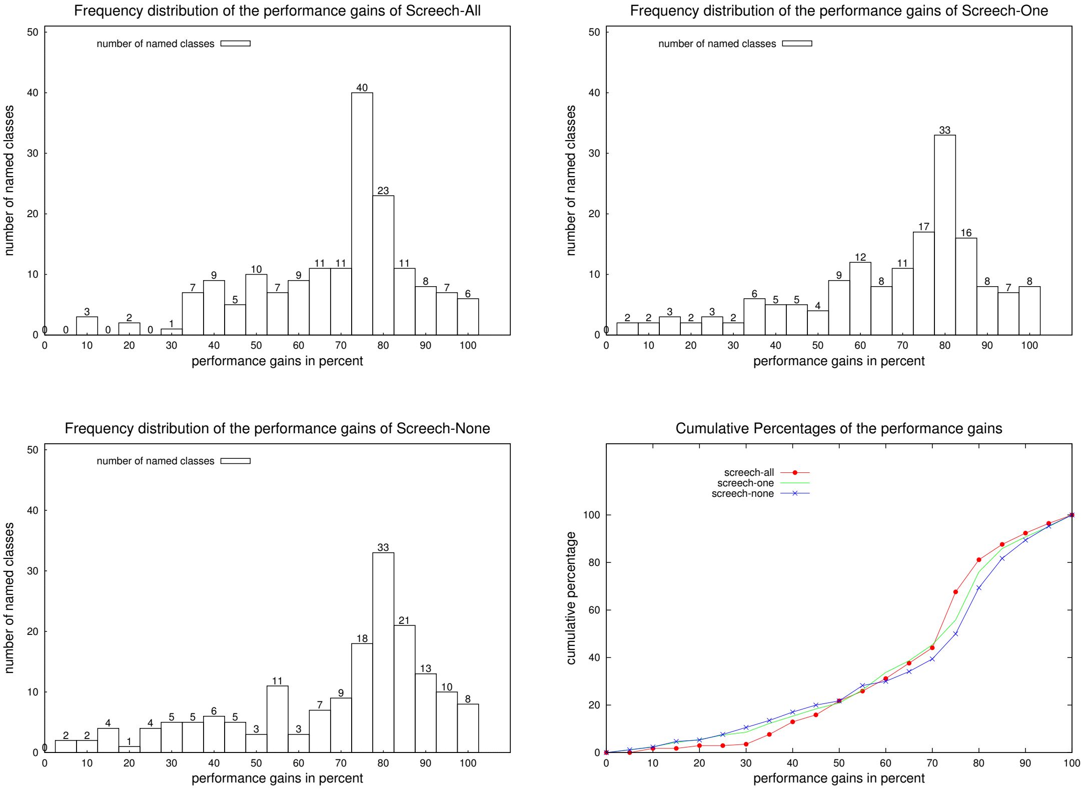


Figure 6.4: Distributions of the performance gain of SCREECH variants compared to KAON2 and the corresponding cumulative percentages. Test ontology: GALEN

For 137 of the 175 classes (*i.e.,* 78%), the computed extensions under SCREECH-ALL and SCREECH-NONE had the same number of elements, which allows to conclude – without using the disjunctive rules – that for those classes the extensions were computed correctly. For some classes, so for the class `Physical-occurrent-entity`, computing the extension under SCREECH-ALL saved 99% of the runtime.

While the different versions of SCREECH have about the same runtime, the differences in the number of introduced errors is remarkable. Indeed, SCREECH-NONE makes almost no mistakes. The parallel execution of SCREECH-NONE and SCREECH-ALL, as mentioned, allows to compute the correct extensions of 78% of the classes – and to know that the computations are correct – in less than a quarter of the time needed by using the unmodified knowledge base.

Examining the frequency distribution of the performance gain of SCREECH variants for each query: Going from left to right, top to bottom, four graphs are plotted in Figure 6.4. The first three graphs illustrate the frequency distributions of the performance gain of the Screech variants compared to KAON2. To compare and contrast these cumulative percentages of the respective performance gain, the graph in the lower right displays them combined. For instance, the distribution graph for Screech-all reveals that there is a time saving of 75 % for 40 queries. Furthermore, it has been explored that 75 % of all the queries tested dispose of a performance gain up to 80%.

DOLCE

DOLCE[5] (a Descriptive Ontology for Linguistic and Cognitive Engineering) is a foundational ontology, developed by the Laboratory for Applied Ontology in the Institute of Cognitive Sciences and Technology of the Italian National Research Council. In full, it exceeds the reasoning capabilities of current reasoners, hence we used a fraction for our experiments consisting of 1552 axioms. Since DOLCE is a pure TBox-Ontology, we randomly populated it with 502 individuals to be able to carry out instance retrieval. As given in Table 6.3 on page 70, the conversion into disjunctive datalog yielded ca. 1774 rules of which ca. 71 are disjunctive. The Screech-all split resulted in 178 new rules, replacing the disjunctive ones. We also removed ca. 189 integrity constraints. The knowledge bases compiled by Screech-all, Screech-one and Screech-none had 1692, 1585 and 1515 rules, respectively. As before, all named classes were queried for their extensions using the KAON2 datalog engine, both for processing the disjunctive datalog program and for the various splits.

Table 6.5: Summary of the three Screech versions on DOLCE

variant (v_i)	**miss**	**corr**	**more**	**time (v_i)**	**time (KAON2)**	**f-meas**	**corr.class**	$\frac{t_{screech}}{t_{kaon2}}$
Screech-all	0	3697	2256	365889	516064	0.766	0.76	0.70
Screech-one	0	3697	512	425748	516064	0.935	1.0	0.82
Screech-none	0	3697	0	397260	516064	1.0	1.0	0.77

Table 6.5 summarizes. In Screech-all, 93 of the 123 classes (*i.e.,* 76%) are correctly queried, while in Screech-one 100 classes are correctly queried.

Remarkable under DOLCE is that Screech-none makes no mistakes, while the runtime improvement is rather mild. Figure 6.5 on page 74 shows the distribution of the performance gain of each variant along with their cumulative percentages. For instance, there is a time saving of 50 % for 18 classes in Screech-one while there is a time saving of 70 % for 16 classes in Screech-none.

[5] `http://www.loa-cnr.it/DOLCE.html` [accessed 2009-5-11]

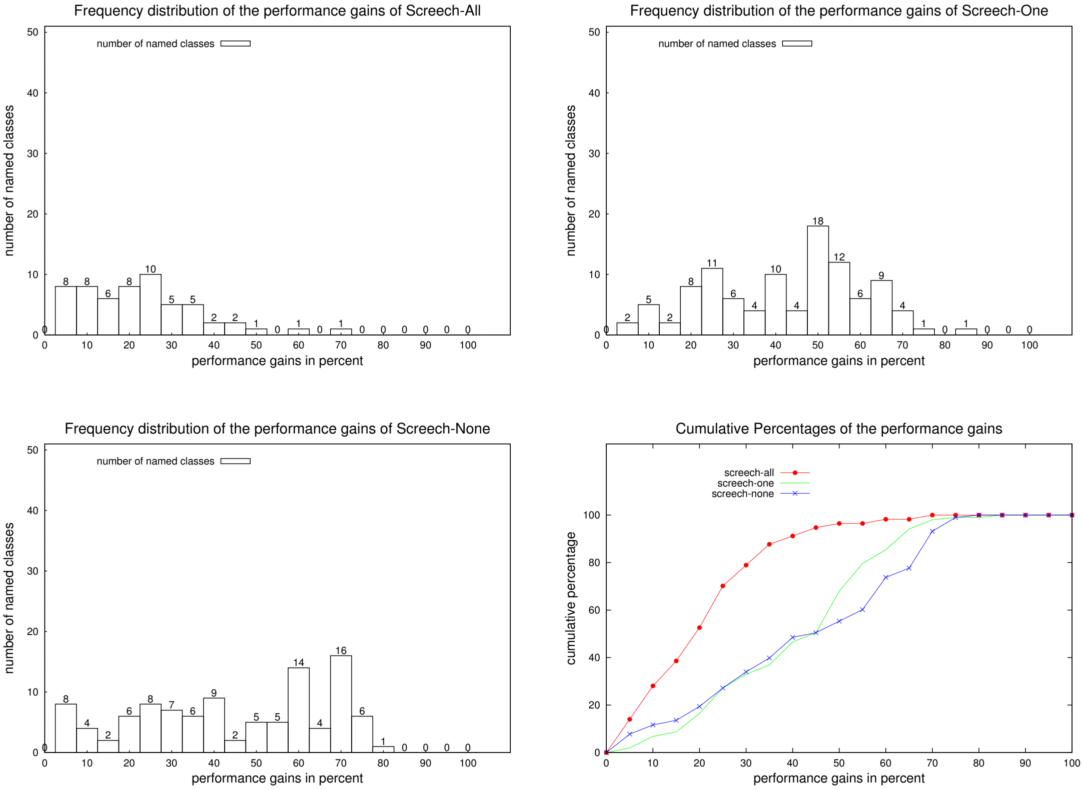


Figure 6.5: Distributions of the performance gain of SCREECH variants compared to KAON2 and the corresponding cumulative percentages.Test ontology: DOLCE

WINE

The next ontology tested was the WINE ontology.[6] It is a well-known ontology containing a classification of wines. Moreover, it is one of the rare ontologies with both an ABox and a nontrivial TBox. It also contains nominals, which we removed in an approximate way following [HV05]. Note that the TBox *after* that processing is used as baseline, since we are interested in the comparion of the different versions of SCREECH. The resulting ontology contains 582 axioms, including functionality, disjunctions, and existential quantifiers. The corresponding ABox contains 22386 axioms.

As given in Table 6.3 page 70, the translation procedure into disjunctive datalog produces altogether ca. 554 rules, among them 24 disjunctive rules. The split program for SCREECH-ALL resulted in a knowledge base with 575 rules by removing 3 integrity constraints and complying the 24 disjunctive rules into 48 Horn rules. As before, all

[6] `http://www.schemaweb.info/schema/SchemaDetails.aspx?id=62` [accessed 2009-5-11]

named classes were queried for their extensions using the KAON2 datalog engine, both for processing the disjunctive datalog program and for the various splits.

A summary of the results can be seen in Table 6.6. For 130 of the 140 classes under Screech-all we obtained 1353 incorrect extensions, while under Screech-one 132 classes are correct queried. Under Screech-none, the number of the classes correctly queried is 126, and totally 697 extensions were missing.

variant (v_i)	miss	corr	more	time (v_i)	time (KAON2)	f-meas	corr.class	$\frac{t_{screech}}{t_{kaon2}}$
Screech-all	0	30627	1353	463396	707476	0.978	0.93	0.65
Screech-one	0	30627	615	494456	707476	0.990	0.94	0.70
Screech-none	697	29930	0	504914	707476	0.988	0.90	0.71

Table 6.6: Summary of the three Screech versions on WINE

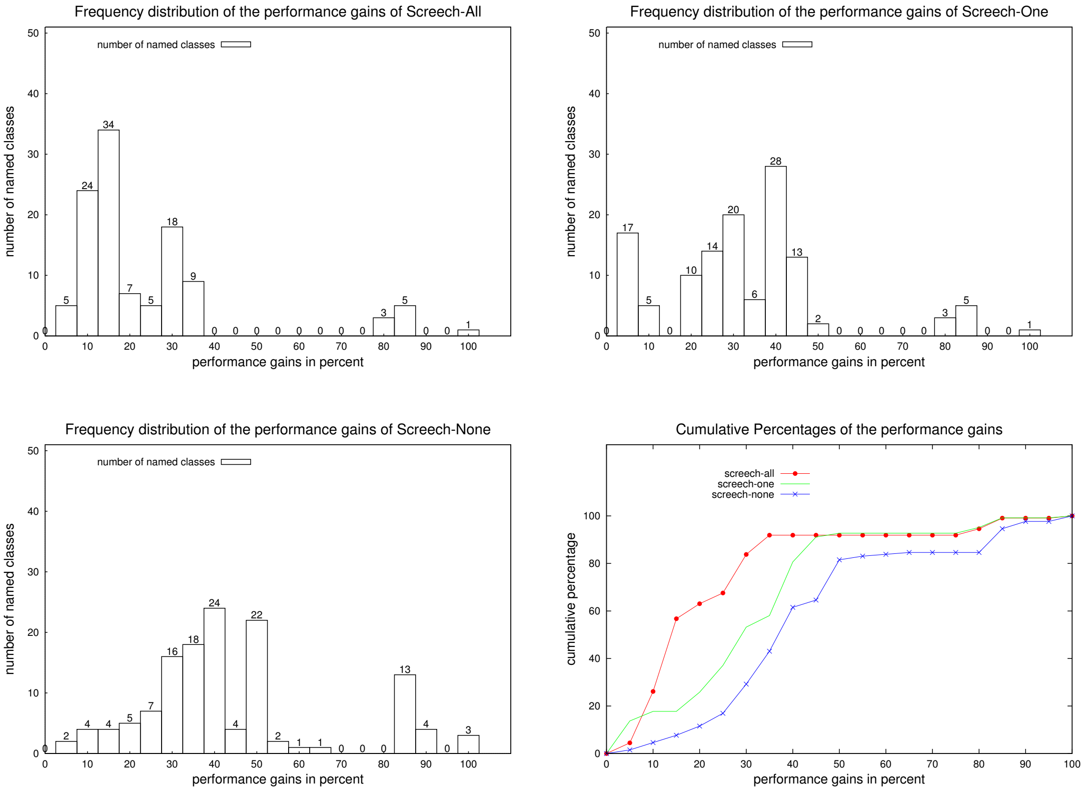


Figure 6.6: Distributions of the performance gain of Screech variants compared to KAON2 and the corresponding cumulative percentages. Test ontology: WINE

WINE is the only ontology we tested for which SCREECH-NONE resulted in a mildly significant number of mistakes. However, recall is still at 0.977, *i.e.,* very good. Please examine Table 6.9 on page 79 for considering recall and precision results for Wine. Considering the fact that WINE was created to show the expressiveness of OWL DL, it is remarkable that all three SCREECH versions show a very low amount of errors, while runtime decreases by 29–35% as illustrated in Table 6.9 on page 79. For some classes – e.g. for `Chianti`, over 91% of the runtime was saved using SCREECH-ALL. Figure 6.6 shows the performance gains obtained in all the variants in more detail.

SEMINTEC

Table 6.7: Summary of SCREECH on SEMINTEC – note that all three versions of SCREECH coincide, since no disjuntive rules are produced by the translation

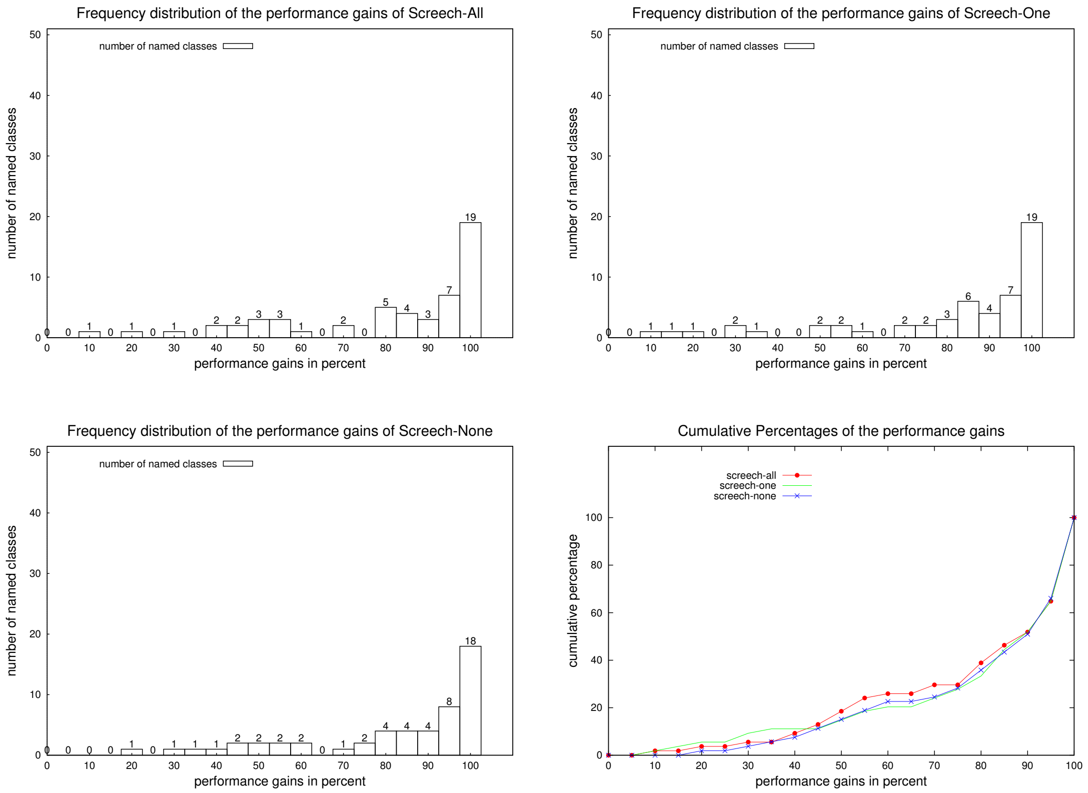
variant (v_i)	**miss**	**corr**	**more**	**time (v_i)**	**time (KAON2)**	**f-meas**	**corr.class**	$\frac{t_{screech}}{t_{kaon2}}$
SCREECH-ALL	0	51184	0	31353	94981	1.0	1.0	0.33
SCREECH-ONE	0	51184	0	32200	94981	1.0	1.0	0.33
SCREECH-NONE	0	51184	0	32032	94981	1.0	1.0	0.33

We also considered an ontology, the translation of which turned out to not contain proper disjunctive rules. Nevertheless, removing integrity constraints is supposed to result in improving runtime behaviour (while in this case even preserving soundness). So, the last ontology we considered is from the SEMINTEC project[7] at the university of Poznan and concerns financial services. Its TBox contains 130702 axioms of comparably simple structure, apart from some functionality constraints which require equality reasoning. The ABox contains 71764 axioms. The TBox translation generated 217 rules, all of them being Horn, among which were 113 integrity constraints. As before, all named classes were queried for their extensions using the KAON2 datalog engine, both for processing the disjunctive datalog program and for the various splits.

A summary of the results can be seen in Table 6.7. As in the case of absence of disjunctive rules all three variants of SCREECH coincide, for all of the 60 classes, the extensions were computed correctly. For SEMINTEC, a performance improvement of 67% has been achieved while the computation remains correct. Figure 6.7 on page 77 shows the distributions of the performance gains of all variants. For some classes – in particular for some with very small extensions, computing the extension under SCREECH-ALL saved about 95% of the runtime. For some classes with larger extension – like `Leasing`, 92% of runtime was saved.

[7] `http://www.cs.put.poznan.pl/alawrynowicz/semintec.htm` [accessed 2009-5-11]

Figure 6.7: Distributions of the performance gain of SCREECH variants compared to KAON2 and the corresponding cumulative percentages. Test ontology: SEMINTEC



VICODI

Another ontology containing no disjunctive rules is the VICODI ontology. But it has a large ABox. It also has no integrity constraints. Hence, the knowledge bases generated are the same; as given in Table 6.3 on page 70, they resulted in a datalog program with 223 Horn rules. A summary of the results can be seen in Table 6.8 on page 78. Like SEMINTEC, for all of the 194 classes, the extensions were computed correctly in each SCREECH variant. The performance gain for SCREECH-ALL, SCREECH-ONE and SCREECH-NONE is 55%, 54% and 53%, respectively. Table 6.8 and Figure 6.8 on page 78 show the performance gain in more detail.

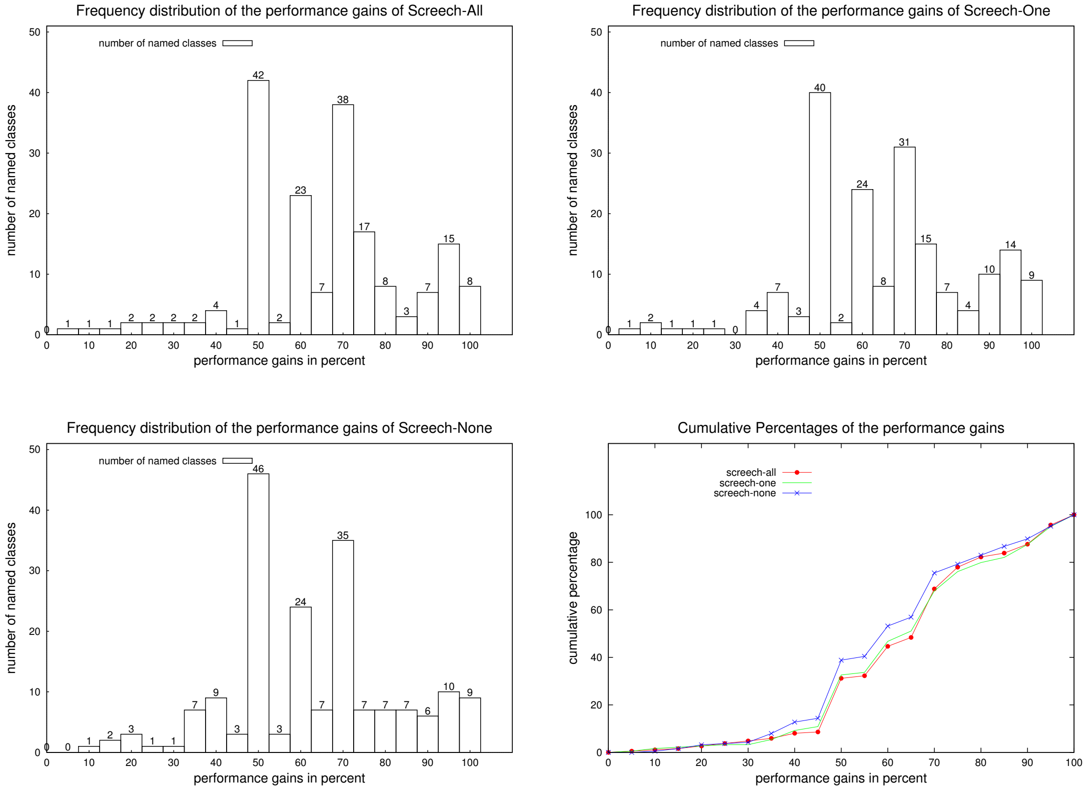
6.6 Conclusions

Motivated by the obvious need for techniques enhancing the scalability of reasoning related tasks, we have investigated three variants of the SCREECH approach to

Table 6.8: Summary of SCREECH on VICODI – note that all three versions of SCREECH coincide, since no disjuntive rules are produced by the translation

variant (v_i)	miss	corr	more	time (v_i)	time (KAON2)	f-meas	corr.class	$\frac{t_{screech}}{t_{kaon2}}$
SCREECH-ALL	0	282564	0	3228	7192	1.0	1.0	0.45
SCREECH-ONE	0	282564	0	3295	7192	1.0	1.0	0.46
SCREECH-NONE	0	282564	0	3346	7192	1.0	1.0	0.47

Figure 6.8: Distributions of the performance gain of SCREECH variants compared to KAON2 and the corresponding cumulative percentages. Test ontology: VICODI



approximate reasoning in OWL ontologies. On the theoretical side, we gave the completeness result for SCREECH-ALL and the soundness result for SCREECH-NONE, yet a desirable characterisation of the approximations in terms of extremal bounds following the theory of Horn-approximations was shown not to hold by providing counterexamples.

However, on the practical side the obtained results were promising: the performance improvement is stable over all ontologies which we included in our experiments. The performance gain varied between 17.5 and 76%, while the amount of correctly retrieved classes was above 62% for all but one of the ontologies – see Table 6.9. It is encouraging to see that the approach appears to be feasible even for the sophisticated WINE ontology, and also for the SEMINTEC ontology, although in the latter case we only remove integrity constraints.

Table 6.9: Overview of SCREECH evaluations. Mark that for due to the completeness of SCREECH-ALL, the recall values are always 100% as well as the precision values for SCREECH-NONE due to its soundness

	SCREECH-ALL			SCREECH-ONE			SCREECH-NONE		
ontology	time saved	prec	recall	time saved	prec	recall	time saved	prec	recall
GALEN	75%	91.7%	100%	73%	97.4%	99.9%	76%	100%	99.8%
DOLCE	30%	62.1%	100%	17.5%	87.8%	100%	23.0%	100%	100%
WINE	35%	95.8%	100%	30%	98.0%	100%	29%	100%	97.7%
SEMINTEC	67%	100%	100%	67%	100%	100%	67%	100%	100%
VICODI	55%	100%	100%	54%	100%	100%	53%	100%	100%

Concerning the comparatively bad results on DOLCE, we note that the results are quite counterintuitive. One would naively expect that performance improvements go hand-in-hand with loss of precision. However, for DOLCE we measured both the least runtime improvement and the worst performance in terms of correctness. Concerning correctness, we suspect that the comparatively large number of incorrect answers is caused by the fact that DOLCE uses a large number of complete partitions of the form $A \equiv A_1 \sqcup \cdots \sqcup A_n$, where all the A_i are also specified to be mutually disjoint. It is intuitively clear that this kind of axioms introduces disjunctive (non-Horn-style and therefore harder to approximate) information on the one hand and integrity constraints (those being neglected in our approximation) on the other. However, this does not in itself explain why we did not observe a higher speedup. This indicates that the properties of ontologies, which lead to performance improvement through the translation of SCREECH, are obviously less straightforward than initially expected. For a clarification, more evaluations taking into account a wider range of ontologies with differing characteristics w.r.t. expressivity, used language features, or statistical measures like degree of population will lead to more substantial hypotheses.

In general, we see a great potential in the strategy to combine various (possibly approximate) algorithms having known properties as soundness and/or completeness maybe with respect to differing types of queries. For instance, the proposed "sandwich technique" can be used to solve instance retrieval tasks in some cases even without calling the more costly sound and complete reasoners. If the sets of individuals I_S and I_C retrieved by two algorithms—one of those being sound and the other one complete—coincide, the result is known to be exact. But even if not, the result is at least partly determined (as all elements of I_S are definitely instances and all individuals *not* in I_C are not) and it might be beneficial to query a sound and complete reasoner for class membership only for individuals of the set $I_C \setminus I_S$ of individuals for which class membership is still undecided. Clearly, the strategy to combine several approximate algorithms will be especially reasonable if parallel computation architectures are available.

Chapter 7

Query Approximation

This chapter discusses the development of an approximate method for instance retrieval of complex concept queries. Instance retrieval is one of the most important inference services in many practical application systems based on DLs, however, it is still one of the major bottlenecks in reasoning over expressive ontologies, in particular, when the number of instances as well as the ontology structure becomes large and complex.

This chapter is structured as follows: Section 7.2 introduces the notion of approximate concept extension. Based on this notion, Section 7.3 provides several approximate algorithms to compute approximate extensions of complex concept queries. Furthermore, a comprehensive empirical analysis of our approach reporting positive results is presented in Section 7.4. Section 7.5 discusses several optimizations to maximize the utility of the approximate algorithms.

7.1 Approximation of Instance Retrieval

Instance retrieval is concerned with finding all the named individuals that are instances of a named class or complex description in a DL knowledge base. The previous chapter demonstrated how to approximate instance retrieval for named classes within the KAON2 approach. This chapter now argues how instance retrieval for complex concept queries can be approximated by reducing it to instance retrieval for named classes.

In most modern DL systems, instance retrieval is tackled by tableaux algorithms. These algorithms work by trying to construct a model of the concept, starting from an individual instance. Tableaux expansion rules constituting to a tableaux algorithm decompose concept expressions, add new individuals and merge existing individuals. Non-determinism resulting from the expansion of disjunctions is dealt with by searching the various possible models. For an unsatisfiable concept, all possible expansions will lead to the discovery of an obvious contradiction known as a clash. For a satisfiable concept, a complete and clash-free model will be constructed.

Adding inverse properties makes practical implementations more problematical as several important optimisation techniques become much less effective. It is well known that basic class consistency/subsumption reasoning problem is EXPTIME-complet for $\mathcal{SHIQ}$, and for $\mathcal{SHOIN}$ this jumps to NExpTime-complete [Tob01].

Coping with the large volumes of instance data that will be required by many applications (*i.e.,* millions of individuals) will be extremely challenging. It seems doubtful that, in the case of instance data, the necessary improvement in performance can be achieved by optimising tableaux based algorithms, which are inherently limited by the need to build and maintain a model of the whole ontology including all of the instance data.

To address this issue related to tableaux algorithms, a significant effort has been investigated which is not based on tableaux algorithms. The main idea in this effort is to apply efficient disjunctive datalog (optimisation) techniques in order to speed up reasoning with large volumes of instance data. Theoretical investigations of this technique have revealed that the data complexity of satisfiability checking against a fixed ontology and set of instance data in $\mathcal{SHIQ}(\mathbf{D})$ is NP-complete. That is, the complexity of answering queries based on this approach is significantly lower than the EXPTIME combined complexity of class consistency reasoning (assuming NP $\subset$ ExpTime[1]). However, this technique has to perform an expensive translation process which compiles a $\mathcal{SHIQ}$ DL knowledge base into a disjunctive datalog program. The resulting disjunctive datalog program is in turn to be queried using a disjunctive reasoning engine. The complexity of the translation algorithms is EXPTIME which means that it would be very time-consuming in case of reasoning over ontologies with a large TBox [Mot06].

This approach to query answering has shown itself to be practical and effective in cases where the TBox is rather simple but the ABox contains large amounts of data. From this result, it poses the question, how query answering can be improved over expressive ontologies with a large TBox as well as ABox. More concretely, how such an expensive TBox-translation for instance retrieval of complex concepts can be properly handled.

In this chapter, we are concerned with these questions and present an approximate reasoning method for scalable query answering of ontologies with a large TBox as well as ABox. Moreover, our approxmate reasoning technique relies on a combination of DL reasoning and database techniques in order to speed up instance retrieval of complex queries.

7.2 Notion Of Approximate Extension

Our approach for the approximation of instance retrieval queries is based on the notion of the *approximate extension* $\langle C \rangle$ of a concept C with respect to a knowledge base KB. Intuitively, $\langle C \rangle$ is the set of instances that are obtained through interpreting com-

[1] `http://en.wikipedia.org/wiki/Computational_complexity` [accessed 2009-07-11]

Table 7.1: Definition of an approximate extension. A stands for atomic classes while C and D stand for complex (non-atomic) classes. R stands for roles and n for a natural number

Approximate Extensions		
$\langle\top\rangle$	$=$	$\lvert\top\rvert$
$\langle\bot\rangle$	$=$	$\varnothing$
$\langle A\rangle$	$=$	$\lvert A\rvert$
$\langle\neg A\rangle$	$=$	$\lvert\neg A\rvert$
$\langle R\rangle$	$=$	$\{(x,y) \mid KB \models r(x,y)\}$
$\langle R^-\rangle$	$=$	$\{(x,y) \mid KB \models r(y,x)\}$
$\langle C \sqcap D\rangle$	$=$	$\langle C\rangle \cap \langle D\rangle$
$\langle C \sqcup D\rangle$	$=$	$\langle C\rangle \cup \langle D\rangle$
$\langle\neg C\rangle$	$=$	$\langle\top\rangle \setminus \langle C\rangle$
$\langle\exists R.C\rangle$	$=$	$\{x \in \langle\top\rangle \mid \exists y : (x,y) \in \langle r\rangle \wedge y \in \langle C\rangle\}$
$\langle\forall R.C\rangle$	$=$	$\{x \in \langle\top\rangle \mid \forall y : (x,y) \in \langle r\rangle \rightarrow y \in \langle C\rangle\}$
$\langle\leq n\, R.C\rangle$	$=$	$\{x \in \langle\top\rangle \mid \#\{y \mid (x,y) \in \langle r\rangle \wedge y \in \langle C\rangle\} \leq n\}$
$\langle\geq n\, R.C\rangle$	$=$	$\{x \in \langle\top\rangle \mid \#\{y \mid (x,y) \in \langle r\rangle \wedge y \in \langle C\rangle\} \geq n\}$

plex concept constructors in C as set operations on the individuals known to KB, starting from the atomic extensions of concepts and roles that occur in C. In this way, the model-theoretic semantics of DLs is approximated by a combination of results for atomic queries that requires less effort to compute than the reasoning process for complex instance retrieval queries in DLs does. The exact definition of an approximate extension is given in Table 7.1 recursively for all language constructs. For an example, consider the knowledge base $KB = \{C \sqsubseteq A \sqcup B, A(a_1), C(a_2)\}$ and the instance retrieval query $A \sqcup B$. The conventional extension of the concept $A \sqcup B$ contains both individuals a_1 and a_2, *i.e.*, $|A \sqcup B| = \{a_1, a_2\}$. However, the approximate extension of $A \sqcup B$ contains only a_1, *i.e.*, $\langle A \sqcup B\rangle = \{a_1\}$.

We will exemplarily explain the computation of the apprximate exteions in case of $\exists R.C$. As defined in Table 7.1, the intention of computing this extension is to determine those individuals that are related to other individuals, which in turn are to be found in the approximate extension of C, *i.e.*, $\langle C\rangle$.

The more complex the query concept C is, the more the approximate extension deviates from the conventional extension. For the simplest queries, such as atomic concepts, the two types of extensions coincide and no errors are made in instance retrieval. This characteristics is captured by the following proposition.

Proposition 7.1 (soundness and completeness of simple approximate extensions)**.** *For a knowledge base KB and a concept C of the form* $C = A_1 \sqcap \cdots \sqcap A_m \sqcap \neg B_1 \sqcap \ldots \neg B_n$*, with all* A_i *and* B_j *atomic, the approximate extension of C is equivalent to its conventional extension, i.e.,* $\langle C\rangle = |C|$.

Proof sketch. Considering Table 7.1 the only difficulty is with conjunction. So let A and B be two atomic concepts (if they are negated atomic concepts, the proof is analogous). Then it suffices to show that $\langle A \sqcap B \rangle = |A \sqcap B|$. Now $\langle A \sqcap B \rangle = \langle A \rangle \cap \langle B \rangle = |A| \cap |B| = \{x \in \sigma(KB) \mid x \in |A| \text{ and } x \in |B|\}$. By definition of the standard description logic semantics, this set contains all $x \in \sigma(KB)$ for which *in all models M of KB* we have both $x^M \in A^M$ and $x^M \in B^M$. But this is equivalent to saying that the set contains all $x \in \sigma(KB)$ for which in all models of M we have $x^M \in A^M \cap B^M = (A \sqcap B)^M$, *i.e.*, for which $x \in |A \sqcap B|$, which was required to show. □

Proposition 7.1 states that, for queries that have the form of conjunctions of possibly negated named concepts, the approximate and conventional extensions have exactly the same instances. In other words, computing the approximate extension is *sound and complete* with respect to the conventional extension.

For more complex queries, however, the approximation might deviate significantly from the correct answer in both that it might miss instances as well as show improper instances. In particular the approximation of the complement constructor is supposed to cause significant deviation as it interprets negation in a closed-world sense, potentially including improper instances in an answer. Hence, we aim at eliminating general complements by means of normalisation, avoiding this source of error.

For standard reasoning in DLs a query concept can be expressed in various normal forms and semantics-preserving transformations do not affect the result of instance retrieval. For the calculation of approximate extensions, however, the result depends on the form of the concept, and different semantically equivalent concept expressions can have different approximate extensions. We can exploit this characteristics by choosing a normal form for query concepts that fits best the process of approximation in terms of both error rate and ease of computation. In this light, we consider the negation normal form [SSS91] of concept expressions for queries, denoted by $\mathsf{NNF}(C)$ for a concept C, in which negation symbols are pushed inside to occur only in front of atomic concepts. This eliminates the case of considering the approximation for general complements with its rather drastic closed-world interpretation. Besides the lower expected error rate this also avoids the computationally costly handling of large sets of individuals in case of large ABoxes by an algorithm that computes approximate extensions. The positive effect that elimination of complement approximation has on the error rate in instance retrieval can be expressed by the following property, which ensures that approximation of concepts in negation formal form only gives up completeness but preserves soundness at least for a certain class of queries.

Proposition 7.2 (soundness of limited approximate instance retrieval)**.** *Let KB be a knowledge base and C be a concept such that $\mathsf{NNF}(C)$ contains no $\forall$- and no $\leq$- and $\geq$-constructs. The approximate extension of $\mathsf{NNF}(C)$ only contains instances that are also contained in the conventional extension of C with respect to KB, i.e., $\langle \mathsf{NNF}(C) \rangle \subseteq |C|$.*

Proof sketch. The proof is by structural induction over C with the base cases A and $\neg A$, using model-theoretic arguments as for Proposition 7.1. □

Proposition 7.2 states that, for queries that do not make use of the $\forall$, $\geq$ and $\leq$ constructs (after normalisation), the approach of approximating concepts in their negation normal form yields an extension that might miss some instances but has no improper instances in it. In other words, computing the approximate extension is *sound* with respect to the conventional extension.

7.3 Computing Approximate Extensions

In this section, we will design algorithms for computing the approximate extension of a complex concept query. We will also lay out the architecture of a system for approximate instance retrieval. The resulting system is called AQA which stands for **A**pproximate **Q**uery **A**nswering.

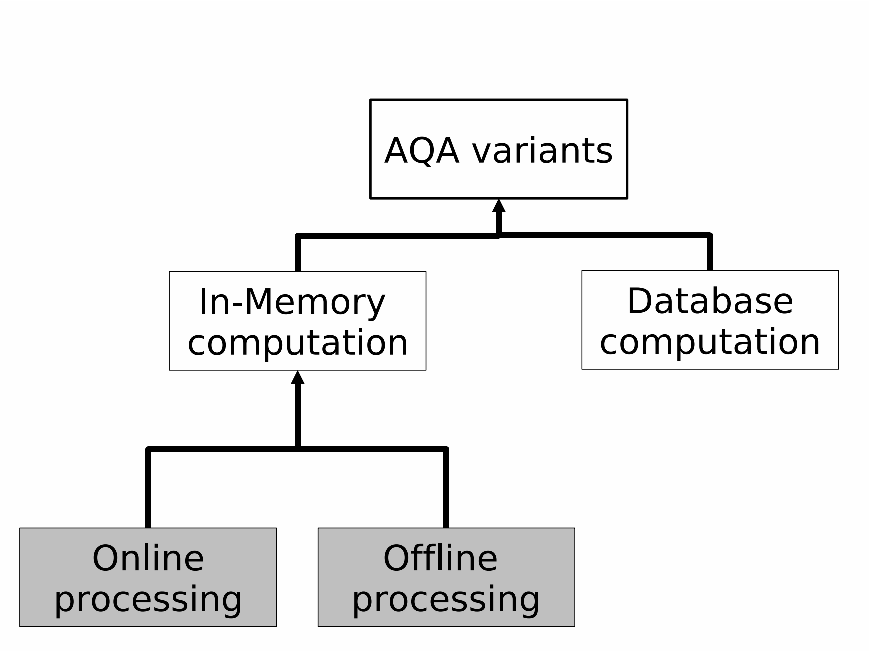


Figure 7.1: A hierarchy of the AQA variants for computing approximate extensions

7.3.1 System Architecture

The core of the AQA system is to compute $\langle Q \rangle$ for a complex query Q. The principle behind this computation is always to start from the individuals in the conventional extensions of (possibly negated) atomic concepts and (possibly inverse) roles that occur in Q and to recursively combine these according to the structure of concept constructors in Q, reflecting the set operations from Table 7.1. According to Propositions 7.1 and 7.2, this results in an answer that is sound and complete for some cases,

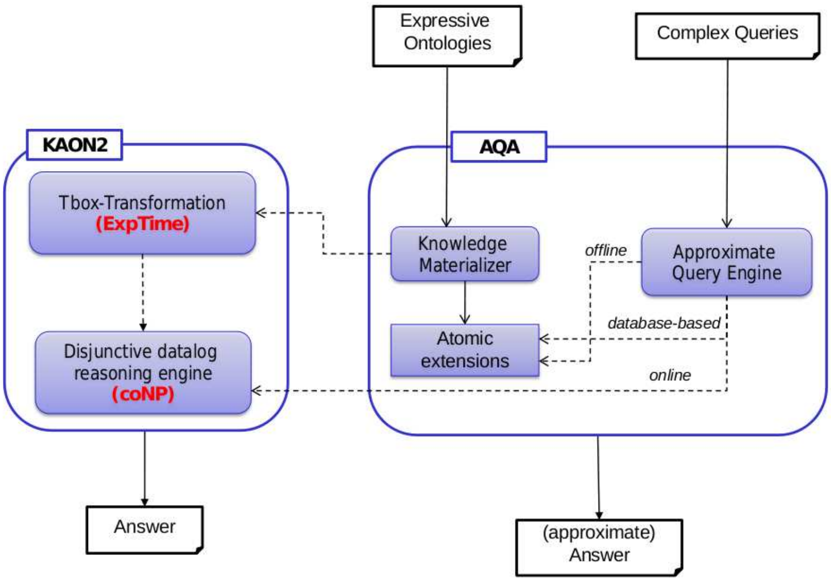


Figure 7.2: An overview of the AQA system architecture

only sound for others, or neither sound nor complete, depending on the language constructs used in the query.

The algorithms for computing the approximate extension can be implemented in several variants. Figure 7.1 on page 85 shows a hierarchy of the variants we have implemented in AQA. First, we distinguish between *database* and *in-memory* computation: in the first case computation is delegated to underlying database operations, whereas in the second case it is performed in main memory. While for the database variant the atomic extensions are pre-computed prior to query-time and materialised in the database using a sound and complete reasoner, the in-memory variant allows for two possibilities to access the atomic extensions: in *online processing* a sound and complete DL reasoner is invoked at query-time to compute atomic extensions, while in *offline processing* they are again pre-computed and materialised either in a database or in memory if possible. Database computation and offline processing are very useful when dealing with a large data sets, whereas online processing is intended to be used in such cases where a materialisation is hardly manageable or subject to frequent changes.

These variants have been implemented into the AQA system whose system architecture is illlustrated in Figure 7.2. AQA takes a $\mathcal{SHIQ}$ knowledge base KB and a complex query concept Q as input to compute the approximate extension of Q as a set of individuals with respect to KB.

The initial step for the AQA variants is to break down the complex query Q into atomic queries which is performed by the Approximate Query Engine component. Depending on the AQA variant chosen by the respective user, this component computes the approximate extension $\langle Q \rangle$ starting with the conventional extensions of the atomic concepts.

For online processing, we utilize the efficient KAON2 reasoner, as illustrated in Figure 7.2. The reason for this choice is that KAON2 was shown to be an efficient ABox reasoner on knowledge bases with large ABoxes and simple TBoxes in comparison to other state-of-the art DL reasoners, which perform better on knowledge bases with large (or complex) TBoxes and small ABoxes. As depicted on the left-hand side of Figure 7.2, KAON2 transforms the TBox together with complex queries into a disjunctive datalog program in a first step, to perform ABox reasoning in a second step based on the result of this transformation. Hence, for every complex ABox query KAON2 needs to repeatedly perform the TBox transformation, which is computationally costly. For ABox queries that have the form of atomic concepts, however, this transformation is not necessary and can be bypassed. For the variant with online processing we can take advantage of this because for computing atomic extensions with KAON2 the costly TBox translation is saved.

In contrast to the variant that incorporates online processing, both offline and database variants require the availability of conventional extensions of the atomic concepts and roles in KB. Regarding this, a materialization ought to be performed beforehand *e.g.,* in an offline phase. As depicted in Figure 7.2, the Knowledge Materializer component invokes KAON2 and computes the atomic extensions of the atomic concepts and roles of KB as well as stores them into a database.

7.3.2 Delegation of Computation to Database

The variant that performs database computation is a presumably efficient implementation of approximate instance retrieval as the pre-computed atomic extensions are materialised and approximate extensions are computed by making use of highly optimised database operations. This variant is essential in practice for handling ontologies with large ABoxes that cannot be processed efficiently in memory. Here, the recursive combination of atomic extensions in terms of set operations as defined in Table 7.1 is completely delegated to the underlying database, which is very useful in practice when dealing with very large ontologies that can hardly be loaded into main memory.

As a basis for this form of computation we use a database schema that consists of two Relations, namely $\mathsf{Ext}^C(ind, class)$ for storing concept extensions and $\mathsf{Ext}^r(ind_1, role, ind_2)$ for storing role extensions. In their schema, the attribute $ind_{(i)}$ stands for individual names, *class* for names of possibly negated concepts and *role* for names of possibly inverse roles. Starting from a knowledge base KB, these two relations are initialised as follows.

$$\mathsf{Ext}^C(ind, class) = \{(a, C) \mid KB \models C(a)\}, \textit{ for } C = A \mid \neg A \textit{ with } A \in \sigma(KB)$$

$$\mathsf{Ext}^r(ind_1, role, ind_2) = \{(a, r, b) \mid KB \models r(a, b)\}, \textit{ for } r = p \mid p^- \textit{ with } p \in \sigma(KB)$$

Concept Expression	**Relational Algebra Expression**
$\tau_{\mathsf{db}}(A)$	$\pi_{[ind]}(\sigma_{[class=A]}(\mathsf{Ext}^C))$
$\tau_{\mathsf{db}}(\neg A)$	$\pi_{[ind]}(\sigma_{[class=\neg A]}(\mathsf{Ext}^C))$
$\tau_{\mathsf{db}}(\exists r.C)$	$E_C := \tau_{\mathsf{db}}(C)$ $E_r := \sigma_{[role=r]}(\mathsf{Ext}^r)$ $\pi_{[ind_1]}(\sigma_{[ind=ind_2]}(E_C \times E_r))$
$\tau_{\mathsf{db}}(\forall r.C)$	$E_C := \tau_{\mathsf{db}}(C)$ $E_r := \sigma_{[role=r]}(\mathsf{Ext}^r)$ $E\_ := \pi_{[ind_1]}(\sigma_{[ind\neq ind2]}(E_C \times E_r))$ $\pi_{[ind_1]}(\mathsf{Ext}^C) \setminus E\_$
$\tau_{\mathsf{db}}(\leq n\, R.C)$	$E_C := \tau_{\mathsf{db}}(C)$ $E_r := \sigma_{[role=r]}(\mathsf{Ext}^r)$ $\pi_{[ind]}(\sigma_{[count(ind_2)\leq n \,\wedge\, ind=ind_2]}(E_C \times E_r))$
$\tau_{\mathsf{db}}(\geq n\, R.C)$	$E_C := \tau_{\mathsf{db}}(C)$ $E_r := \sigma_{[role=r]}(\mathsf{Ext}^r)$ $\pi_{[ind]}(\sigma_{[count(ind_2)\geq n \,\wedge\, ind=ind_2]}(E_C \times E_r))$
$\tau_{\mathsf{db}}(C_0 \sqcap C_1 \sqcap \cdots \sqcap C_n)$	$\tau_{\mathsf{db}}(C_0) \cap \tau_{\mathsf{db}}(C_1) \cap \cdots \cap \tau_{\mathsf{db}}(C_n)$
$\tau_{\mathsf{db}}(C_0 \sqcup C_1 \sqcup \cdots \sqcup C_n)$	$\tau_{\mathsf{db}}(C_0) \cup \tau_{\mathsf{db}}(C_1) \cup \cdots \cup \tau_{\mathsf{db}}(C_n)$

Table 7.2: Mapping of DL concept expressions to relational algebra expressions

Notice that, for the purpose of approximate instance retrieval, Ext^C and Ext^r form a complete representation of the original knowledge base KB.

A complex query concept Q is answered by transforming its negation normal form into a relational algebra expression according to a mapping τ_{db} posed as a query to the underlying database system. The complete mapping definition for τ_{db} is given in Table 7.2. The left-hand side shows the concept constructors that can occur in $\mathsf{NNF}(Q)$ and the right-hand side shows their respective relational algebra expression. Recursive application of τ_{db} ultimately produces a single database query $\tau_{\mathsf{db}}(\mathsf{NNF}(Q))$ that is used for computing $\langle Q \rangle$.

In the following, we will explain the mapping τ_{db} for $\exists r.C$, which is defined in the third row of Table 7.2. According to the definition given on the right side, τ_{db} is recursively applied to the concept C, which can be a complex class expression. E_C denotes the resulting set, *e.g.*, the approximate extension of C. In case C is an atomic concept, the extension of C is determined by querying the database table Ext^C, as defined in the first row of Table 7.2. In case C is a negated atomic concept, it is determined as defined in the second row of Table 7.2.

Returning to the definition of the mapping τ_{db} for $\exists r.C$, the second step now is to determine those individuals, which are related by role r. E_r denotes the set of those individuals and is determined by querying the database table Ext^r on condition that $[role = r]$. This means that the execution of statement $\sigma_{[role=r]}(\mathsf{Ext}^r)$ accesses the rows of the database table Ext^r. It then filters those rows whose columns contain role r.

The final step is to determine only those individuals from E_r, which are related to individuals that are in E_C. Those are instances of the approximate extension of $\exists r.C$ and are determined by the expression $\pi_{[ind_1]}(\sigma_{[ind=ind_2]}(E_C \times E_r))$, which uses a selection of the cross product of E_C and E_r and projects the resulting set to ind_1.

For an example consider the query $Q = A \sqcap \exists r.\neg B$. The mapping τ_{db} produces the following nested relational algebra expression.

$$\begin{aligned}\tau_{\mathsf{db}}(Q) = {} & \pi_{[ind]}(\sigma_{[class=A]}(\mathsf{Ext}^{\mathsf{C}})) \cap \\ & \pi_{[ind_1]}(\sigma_{[ind=ind_2]}(\sigma_{[role=r]}(\mathsf{Ext}^r) \times \pi_{[ind]}(\sigma_{[class=\neg B]}(\mathsf{Ext}^{\mathsf{C}}))))\end{aligned}$$

When posed to the underlying database, this rather large expression is subject to efficient internal query optimisation strategies as they are typically employed by database systems.

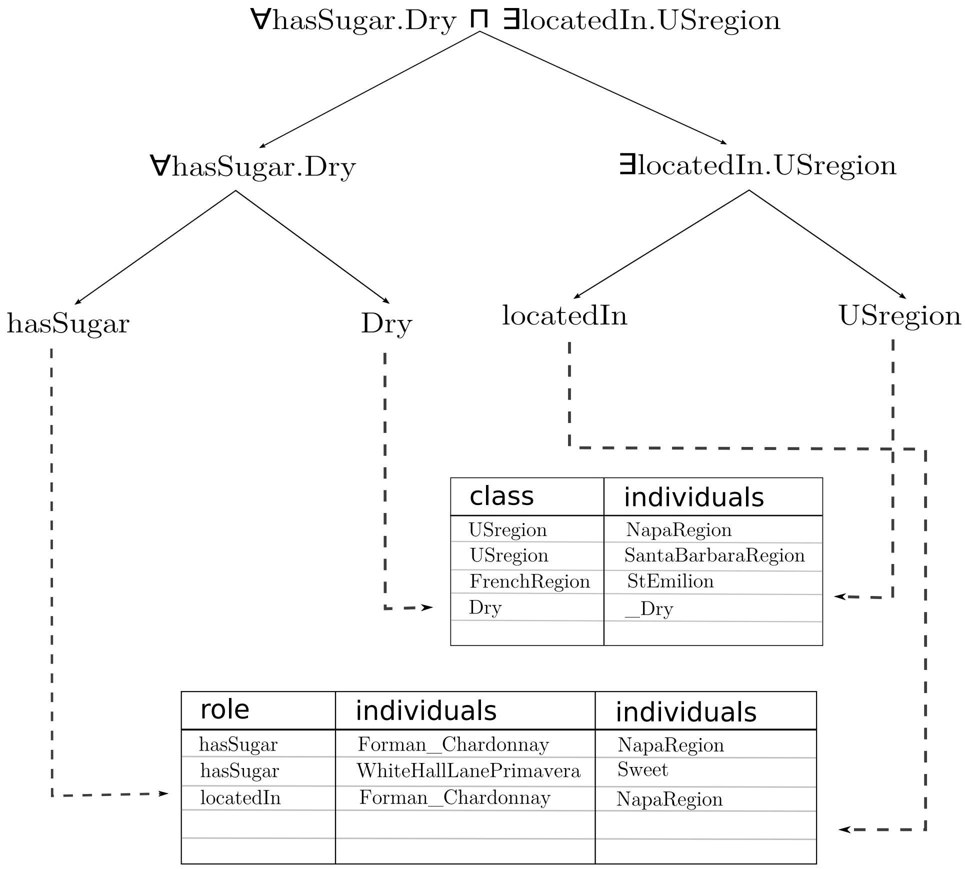
7.3.3 In-memory Computation

Both the variants with online and offline processing share the same implementation of the approximate algorithm of which the pseudocode is described in Algorithm 7. The difference is the handling of atomic extensions which is presented by the function `Compute_Ext`. This function takes as parameters a knowledge base and an atomic concept or atomic role for which the atomic extension is to be computed using a DL reasoner. The algorithm ψ accepts the knowledge base and a complex concept query for which the approximate extension is to be computed.

For the computation of the atomic extension, depending on the chosen variant, `Compute_Ext` invokes either a complete and sound reasoner or retrieves the atomic extension from the database. In the following, we exemplarily describe the most common part of the algorithm. For concepts being of the form $\exists r.C$, it first computes the atomic extension of the role r, represented by $\mathcal{R}$. Next, it recursively calculates the approximate extension of the concept C, which is assigned to a temporary set $\mathcal{C}$. Then, it determines such role assertions in $\mathcal{R}$ whose the second component belongs to $\mathcal{C}$ and returns the set of their first components. A similar procedure is used in the calculation of $\forall r.C$ and cardinality restrictions. However, for cardinality restrictions the number of explicit role fillers is taken into account and for $\forall r.C$ the set difference of $\langle \top \rangle$ and $\mathcal{B}$.

Algorithm 7: $\psi(KB,Q)$

```
Data: $\mathcal{SHIQ}$ knowledge base $KB$ and a complex query concept $Q$ in NNF
Result: The approximate extension of $Q$
if $Q$ is of the form $A$ or $\neg A$ then
    return Compute_Ext$(KB, Q)$
else if $Q$ is of the form $\exists r.C$ then
    $\mathcal{T} := \varnothing; \mathcal{R} :=$ Compute_Ext$(KB, r); \mathcal{C} := \psi(KB, C)$;
    for $(x, y) \in \mathcal{R}$ do
        if $y \in \mathcal{C}$ then
            $\mathcal{T} := \mathcal{T} \cup \{x\}$
    return $\mathcal{T}$
else if $Q$ is of the form $\forall r.C$ then
    $\mathcal{T} := |\top|; \mathcal{B} := \varnothing; \mathcal{R} :=$ Compute_Ext$(KB, r); \mathcal{C} := \psi(KB, C)$;
    for $(x, y) \in \mathcal{R}$ do
        if $y \notin \mathcal{C}$ then
            $\mathcal{B} := \mathcal{B} \cup \{x\}$
    return $\mathcal{T} \backslash \mathcal{B}$;
else if $Q$ is of the form $\geq n\, r.C$ or $\leq n\, r.C$ then
    $\mathcal{T} := \varnothing; \mathcal{K} := \varnothing; \mathcal{R} :=$ Compute_Ext$(KB, r); \mathcal{C} := \psi(KB, C)$;
    for $(x, y) \in \mathcal{R}$ do
        if $y \in \mathcal{C}$ then
            $\mathcal{K} := \mathcal{K} \cup \{(x, y)\}$
    if $\geq n\, r.C$ then
        we determine such $x$ holding the property
        $\{x \in |\top| \mid \#\{y \mid (x, y) \in \mathcal{K}\} \geq n\}$ and put it into $\mathcal{T}$
    else if $\leq n\, r.C$ then
        we determine such $x$ holding the property
        $\{x \in |\top| \mid \#\{y \mid (x, y) \in \mathcal{K}\} \leq n\}$ and put it into $\mathcal{T}$
    return $\mathcal{T}$
else if $Q$ is of the form $C_0 \sqcap C_1 \cdots \sqcap C_n$ then
    return $\psi(KB, C_0) \cap \psi(KB, C_1) \cap \cdots \cap \psi(KB, C_n)$;
else if $C$ is of the form $C_0 \sqcup C_1 \cdots \sqcup C_n$ then
    return $\psi(KB, C_0) \cup \psi(KB, C_1) \cup \cdots \cup \psi(KB, C_n)$;
return $\varnothing$;
```



class	individuals
USregion	NapaRegion
USregion	SantaBarbaraRegion
FrenchRegion	StEmilion
Dry	_Dry

role	individuals	individuals
hasSugar	Forman_Chardonnay	NapaRegion
hasSugar	WhiteHallLanePrimavera	Sweet
locatedIn	Forman_Chardonnay	NapaRegion

Figure 7.3: An example of computing the approximate instance extension in the variant with offline processing

7.3.4 Example of Computing the Approximate Instance Extension

In order to better appreciate the difference between the various variants, we consider a complex query:

$$\forall \mathsf{hasSugar}.\textsc{Dry} \sqcap \exists \mathsf{locatedIn}.\mathrm{US}\textsc{Region}.$$

In the first step, the complex query is splitted in AQA so we get the concept tree illustrated in Figure 7.3. The leaves are labelled with the atomic roles hasSugar and locatedIn and the atomic classes DRY and USREGION. The interior nodes are labelled with the subconcepts ∀hasSugar.DRY and ∃locatedIn.USREGION.

In the second step, the conventional extensions of the atomic classes and roles will be determined in two ways. We can compute the extensions either by querying the atomic classes and roles with a DL reasoner which is the variant with online processing or by retrieving the database in which the extensions are already stored, *i.e.*, offline processing. In the upper part of the Figure 7.3, two database schema are given which contain the corresponding extensions of the atomic classes and roles of the query we consider. Having determined the atomic extensions, the approximation extension of the query will be computed as defined in Algorithm 7.

In contrast to these variants with online as well as offline processing, the database-based variant makes use of the mapping τ_{db}, defined in Table 7.2, produces the following nested relational algebra expression which will be posed to the underlying database:

$$\tau_{\mathsf{db}}(\forall \mathsf{hasSugar}.\textsc{Dry} \sqcap \exists \mathsf{locatedIn}.\mathrm{US}\textsc{Region}) = \tau_{\mathsf{db}}(\forall \mathsf{hasSugar}.\textsc{Dry}) \cap \tau_{\mathsf{db}}(\exists \mathsf{locatedIn}.\mathrm{US}\textsc{Region})$$

, where

$$\tau_{\mathsf{db}}(\forall \mathsf{hasSugar}.\textsc{Dry}) = \pi_{[ind_1]}(\mathsf{Ext}^{\mathsf{C}}) \setminus \pi_{[ind_1]}(\sigma_{[ind \neq ind_2]}(\sigma_{[role=\mathsf{hasSugar}]}(\mathsf{Ext}^{r}) \times \pi_{[ind]}(\sigma_{[class=\textsc{Dry}]}(\mathsf{Ext}^{\mathsf{C}}))))$$

and

$$\tau_{\mathsf{db}}(\exists \mathsf{locatedIn}.\mathrm{US}\textsc{Region}) = \pi_{[ind_1]}(\sigma_{[ind=ind_2]}(\sigma_{[role=\mathsf{locatedIn}]}(\mathsf{Ext}^{r}) \times \pi_{[ind]}(\sigma_{[class=\mathrm{US}\textsc{Region}]}(\mathsf{Ext}^{\mathsf{C}})))).$$

7.4 Experimental Results

We have conducted experiments to determine how effective our approach is at instance retrieval for complex concept queries. In this section, we present the experimental results obtained from the execution of online and offline processing in the in-memory variant as well as the database variant. The primary metrics we have considered are response time and correctness of the computed extensions in terms of precision and recall.

7.4.1 Test Data

There are already several well-known benchmarks for evaluating DL reasoning systems. However, objectively comparing the performance of approximate reasoning systems with that of complete and sound DL reasoners is different, and in fact it is not a priori clear how this should best be done. So currently, there exist no generally accepted benchmarks. The point of difficulty is that we do not only want to measure execution time, but we also need to determine empirically to what extent the evaluated algorithms are sound and complete – in terms of precision and recall. In order to do this, we need to test a suitable and large enough sample of queries whilst for comparing complete and sound reasoning systems usually a few complex queries suffice.

This poses the question, however, which of the potentially infinitely many queries should be used for the testing. Obviously, we want to restrict our attention to a finite set, but two general problems arise: It is not possible to do unbiased random selections from an infinite set, and many randomly generated queries would simply result in no answer at all. For our experiments, we thus confine our attention to relatively simple queries such that we have only a finite set to choose from, and we furthermore restrict our attention to those queries which actually produce non-empty answers using a sound and complete DL reasoner. The queries which we use for testing are of the forms $A \sqcap B$ (we call them $\sqcap$-*queries*), $A \sqcup B$ ($\sqcup$-*queries*) and $\exists R.A$($\exists$-*queries*) where A and B are named classes. Queries of this form suffice to show how our approach performs on queries involving these class constructors, which can obviously also be used to construct more complex queries.

Another issue is to choose appropriate test ontologies. Not all well-known benchmarking ontologies are suitable for highlighting the performance of our approximate algorithms. A particular difficulty is to find realistic ontologies which are of sufficient expressivity in terms of TBox constructors, and at the same time have a reasonably sized ABox. Since we consider scalable reasoning over expressive ontologies with large TBoxes and ABoxes, we decided to use the WINE ontology for our evaluation. WINE has originally been designed as a showcase for the expressivity of OWL, and thus is a hard enough task to tackle using reasoning. WINE has 140 named classes, 10 object properties, 123 subconcept relations, 61 concept equivalences, 10127 concept assertions and 10086 role assertions. As queries, we considered about 9876 queries of the forms $A \sqcap B$ as well as $A \sqcup B$, 107 queries of the forms $\exists R.A$ as well as $\geq n\ R.A$ and 57 queries of the form $\forall R.A$. We refer to these test queries as the basic queries. The approximate algorithms are implemented in Java 6.0 using the KAON2 API. Computation times are reported in milliseconds. All tests were performed on a Lenovo laptop with dual 2.40 GHz Intel(R) Core(TM)2 Duo processors, 2GB of RAM ((with 1024M heap space allocated to JVM)), Ubuntu 8.04, Kernel Linux 2.6.24- 21-generic.

7.4.2 Results

In our experiments, we compared our algorithms with KAON2. The results are summarised in Tables 7.3 through 7.5.

Table 7.3 on page 94 shows the performance of reasoning measured for running the AQA variants and the KAON2 reasoner on basic queries. For $\exists$-queries, we first had to identify a meaningful set of such queries, as randomly generated ones very often had empty extensions. We thus considered only such queries for which KAON2 computed non-empty extensions. This way, we identified 107 $\exists$-queries for testing. Running the approximation algorithm in the database variant, we obtained a significant performance improvement for each $\exists$-query, about 90%. The mean time consumed to answer the queries was 274 ms while it was 2789 ms for KAON2. Running the algorithm in offline processing where the approximation is computated in memory, we obtained another significant performance gain, indeed about 99% compared to KAON2.

Table 7.3: Summary of the performances measured for the offline and database variants, summarized over all considered queries. t_{db} gives the runtime for the database variant, $t_{offline}$ gives the runtime for the offline variant while t_{kaon2} gives the runtime of KAON2 – these times are in ms and are the sums over all considered queries

query (C)	$t_{offline}$	t_{db}	t_{kaon2}	$\frac{t_{offline}}{t_{kaon2}}$	$\frac{t_{db}}{t_{kaon2}}$
$\exists R.A$	3187	29331	298472	0.01	0.098
$\geq n\, R.A$	404	4524	71031	0.005	0.063
$\forall R.A$	545	13847	173088	0.003	0.080
$A \sqcup B$	552	11681	312677	0.001	0.037
$A \sqcap B$	2801	5432	278805	0.01	0.019

In addition to measuring the performance of AQA variants, we have determined the quality of AQA compared with that of KAON2. Table 7.4 on page 94 shows the quality of the AQA approximation for the basic queries we have tested. By comparing the approximate extensions provided by AQA with the conventional extensions provided by KAON2, for each query, we have determined the missing as well as the incorrect individuals, which might appear in the approximate extensions provided by AQA.

Please note that Table 7.4 displays the sum of the missed, incorrect individuals and the correct individuals provided by AQA for the basic queries. *miss* indicates the elements of the conventional extensions which were *not* found by the approximation, *corr* indicates those which were correctly found, and *more* indicates those which were incorrectly computed to be part of the extension. $\langle C \rangle$ indicates the sum of the sizes of the approximate extensions while $|C|$ indicates the sum of the sizes of the conventional extensions.

Table 7.4: Summary of the quality of AQA approximation, summarized over all considered queries

query (C)	miss	corr	more	$\langle C \rangle$	$\|C\|$
$\exists R.A$	35752	68347	0	68347	104099
$\geq n\, R.A$	872	1667	0	1667	2539
$\forall R.A$	0	3730	5109	8839	3730
$A \sqcup B$	0	43050	0	43050	43050
$A \sqcap B$	0	1476	0	1476	1476

Considering the rather drastic speed-up by one respectively two orders of magnitude, as illustrated in Table 7.3, the introduced error for the ∃-queries appears to be rather mild, namely with a recall of 0.66 and a precision of 1.0, resulting in an f-measure of 0.79 as illustrated in Table 7.4 on page 94. Note also that we considered only queries which do have a non-empty extension: Obviously, queries with no answers are also sometimes used. If we would add such queries to our sample set, then recall would be even better, while precision would be unaffected.

For the ⊓-queries we obtained even more favourable results. The performance gain for each query was above 95% while recall, precision and f-measure are 100% as expected by Proposion 7.1. For the database variant the overall time gain was 98 %, while it was 99% for the offline variant.

For the ⊔-queries we find it remarkable that recall turns out to be 100%, which is coincidental.[2] At the same time, we obtained reasonable speed-up for the database variant, namely 99%, and remarkable 99.9%, *i.e.,* three orders of magnitude, for the offline variant. Turning to the online variant, we obviously obtain the same values for precision and recall as for the offline and database variants. As for computation time, we expect an improvement over KAON2 if the TBox translation is time-consuming (since the approximate algorithm does not need it) compared to the ABox reasoning part performed with the datalog reasoner. For WINE, the effect is rather small, so we would need an ontology with a TBox translation with is very expensive compared to the datalog reasoning part. However, we did not find any real ontology with this property.

Applying the measurements conducted in Table 7.4 on page 94, we have determined the corresponding recall, precision and f-measure for each form of the basic queries that are illustrated in Table 7.5 on page 96. The notions used in Table 7.5 are the following: *f-meas* is the f-measure known from information retrieval, computed as

$$\frac{(2 \cdot precision \cdot recall)}{(precision + recall)}$$

with

$$precision = \frac{corr}{(corr + more)} \text{ and } recall = \frac{corr}{number\ of\ actual\ instances}$$

, where values are taken from Table 7.4. Note that precision shows the measure of exactness, whereas recall gives the measure of completeness.

In order to show that for some ontologies we would get the desired effect, we thus modified WINE by multiplying the TBox, *i.e.,* we used *n* renamed copies of the original TBox together with the original ontology (including the ABox). Table 7.6 shows the results for the original WINE, for WINE with additional 4 copies of the TBox, and for WINE with additional 9 copies of the TBox. Indeed the TBox translation took 2629 ms for WINE, 6674 ms for WINE with 4 TBox copies, and 18223 ms for WINE with 9 TBox

[2] The experiments performed in [TRKH08] show that disjunction cannot in general be ignored in WINE without loss of precision or recall.

Table 7.5: Overview on the degree of completeness and soundness, summarized over all considered queries

query	recall	precision	f-meas
$\exists R.C$	0.66	1	0.79
$\geq n\, R.A$	0.66	1	0.79
$\forall R.A$	1	0.42	0.59
$A \sqcup B$	1	1	1
$A \sqcap B$	1	1	1

copies, showing – as expected – a worse than linear development. When taking WINE with 10 TBox copies, KAON2 in fact was no longer able to translate the TBox.

Table 7.6: Measured performance for quering 100 $\sqcup$ queries over wine ontology with different TBoxes

ontology	t_{online}	t_{kaon2}	$\frac{t_{online}}{t_{kaon2}}$
WINE	243808	277714	0.88
WINE + 4 TBoxes	551101	922626	0.60
WINE + 9 TBoxes	1055881	2058829	0.51

Let us discuss the matter of complex queries, *i.e.,* of queries involving more than one class constructor. Generally speaking, answering complex queries with KAON2 is not more expensive than answering simple queries of the form we have discussed so far. For our approximate approach, however, any additional class constructor in the query causes additional computation, namely the retrieval of one or several approximate extensions, and the combination of these with the result of the remaining query. The effect of this can be expected to be roughly linear in the number of class constructors. In the end, this means that our approach yields less speed-up for more complex queries, but we also see from the figures in Table 7.3 that the approximation can still be worthwhile, in particular for the in-memory offline variant.

In order to investigate the effects of error compensation for complex queries, we consider some complex concepts presented in Table 7.7. Consider first the complex concept C_1. The conventional extension of C_1 contains 37 individuals. In contrast, the approximate extension $\langle C_1 \rangle$ only contains 36 individuals; one individual is not found by the algorithm due to its incompleteness for $\sqcup$ and $\exists$-queries. If we combine this query with a more specific concept WHITEWINE as in C_2, the missing individual disappears, *i.e.,* the error is being compensated for and we have that $\langle C_2 \rangle = |C_2|$.

Table 7.7: Approximate extensions of complex queries

query (C_i)	miss	corr	more	$\langle C_i \rangle$	$\lvert C_i \rvert$
$C_1 = \exists \mathsf{locatedIn}.(\textsc{ItalianRegion} \sqcup \textsc{USRegion})$	1	36	0	36	37
$C_2 = C_1 \sqcap \textsc{WhiteWine}$	0	7	0	7	7
$C_3 = \forall \mathsf{hasSugar}.\textsc{Dry}$	0	47	109	156	47
$C_4 = \forall \mathsf{hasSugar}.\textsc{Dry} \sqcap \textsc{WhiteWine}$	0	18	2	20	18
$C_5 = \forall \mathsf{hasSugar}.\textsc{Dry} \sqcap C_1 \sqcap \textsc{WhiteWine}$	0	6	0	6	6

As for unsoundness, we consider the query C_3 and its approximate extension. There are many (109) individuals that are computed incorrectly. Note that we have had relatively low precision for $\forall$-queries, in Table 7.7. If we further constrain this query by a conjunction with the named concept WHITEWINE as in C_4, then the amount of the individuals computed incorrectly for the query C_3 drastically reduces from 109 to 2. This shows that also an error due to unsoundness can be compensated for by a combination of constructs in complex queries.

Table 7.8: Running times (ms) obtained for complex queries

query (C_i)	$t_{offline}$	t_{db}	t_{kaon2}
$C_1 = \exists \mathsf{locatedIn}.(\textsc{ItalianRegion} \sqcup \textsc{USRegion})$	11	123	1900
$C_2 = C_1 \sqcap \textsc{WhiteWine}$	15	137	1583
$C_3 = \forall \mathsf{hasSugar}.\textsc{Dry}$	5	104	1219
$C_4 = \forall \mathsf{hasSugar}.\textsc{Dry} \sqcap \textsc{WhiteWine}$	9	129	1500
$C_5 = \forall \mathsf{hasSugar}.\textsc{Dry} \sqcap C_1 \sqcap \textsc{WhiteWine}$	23	161	1564

As a final example, we consider the intersection of all the query constructs used above in the query C_5. Here we have full compensation of both kinds of errors as its approximate extension is equal to its conventional one, *i.e.*, $\langle C_5 \rangle = |C_5|$. This suggests that for many practical complex queries our approximation approach yields less errors than indicated for the basic queries in Table 7.3 on page 94 due to the effect of compensation, while speed-up in computation is still significant as given Table 7.8.

7.5 Extensions

Several variants in the AQA system that rely on the Algorithm 7 and the mapping algorithm defined in Table 7.2 have been introduced. This section now explores the weaknesses of each variant and proposes several improvements. The next subsection introduces an approach which aims to improve performance in online processing. The basic idea is here to make use of an approximate knowledge base with better computational properties, instead of the original knowledge base. Subsection 7.5.3 discusses the need of an incremental materialization in offline processing and presents an algorithm which enables incremental computation of a materialisation depending on queries to be posed. Subsection 7.5.2 discusses parallel computation of atomic extensions in online processing.

7.5.1 Combination with Knowledge Approximation

It has been described in Section 7.3 that online processing requires a sound and complete DL reasoner to compute atomic extensions on a given knowledge base. In order to improve the efficiency of this computation, one can make use of a knowledge base pre-compiled by the SCREECH algorithms, instead of the original knowledge base.

Please recall the computational characteristics of the SCREECH variants from Section 6.4, Chapter 6. According to Table 6.1 on page 67, for instance, SCREECH-NONE is a sound and but incomplete approximation. However, it should be noted that the actual completeness of this approximation depends on a particular query and knowledge base. For instance, consider the experimental results from the evaluation of the Semintec ontology illustrated in Figure 6.7. There is 50 % performance gain for 80 % of the queries tested; at the same time, all the SCREECH variants compute the extensions correctly and completely (see Table 6.9).

This suggests to apply SCREECH approximations to knowledge bases handled by the AQA algorithm in online processing. In an ideal case, like for Semintec ontology, soundness and completeness of the extensions of named classes are preserved in online processing and at the same time it will be possible to improve performance.

Another motivation to combine SCREECH and AQA systems is to avoid an expensive TBox translation required for answering complex queries in case of invoking the KAON2 reasoner. Algorithm 8 defines this combination. In lines 2 to 10, the given knowledge base is compiled into a Horn program P_{Horn} by a SCREECH-approximation which is parameterized through variable t. Finally, Algorithm 7 is applied to P_{Horn} in order to compute the approximate extension of the complex query Q.

7.5.2 Parallel Computation of Atomic Extensions

As already stated, ABox reasoning problems are often computational intense. Up to now, only the structure and functionality of approximate instance retrieval algorithms have been explored without paying attention to their potential of parallelization. In general parallelization[3] means to search for pieces of code that can potentially run concurrently and having them executed by different processors.

It has been revealed that AQA does not provide a satisfying performance in online processing. The reason for this is that it computes atomic extensions of a complex query only sequentially. The more complex the query, the longer it takes to compute atomic extensions in online processing.

[3] http://en.wikipedia.org/wiki/Parallelization [accessed 2009-07-03]

```
Algorithm 8: $\psi_{screech}(KB, Q, t)$
Input: $KB$: a $\mathcal{SHIQ}$ knowledge base
Input: $Q$: a complex query concept in NNF
Input: $t$: the required SCREECH approximation
Result: the approximate extension of $Q$
begin
    $\mathsf{D}(KB) \longleftarrow tboxTranslation(KB)$;
    $P_{Horn} \longleftarrow 0$;
    switch $t \in \{screech\_one, screech\_one, screech\_all\}$ do
        case $t$ is SCREECH-ALL
            $P_{Horn} \longleftarrow$ SCREECH-ALL($\mathsf{D}(KB)$);
        case $t$ is SCREECH-ONE
            $P_{Horn} \longleftarrow$ SCREECH-ONE($\mathsf{D}(KB)$);
        case $t$ is SCREECH-NONE
            $P_{Horn} \longleftarrow$ SCREECH-NONE($\mathsf{D}(KB)$);

    return $\psi(P_{Horn}, Q)$;
end
```

This issue is illustrated by the following simple example. Given a complex query C of the form $\exists.r.C_1 \sqcap C_2$, we denote the time needed by the approximate extension method in online processing as $t_{online}(C)$. This time is estimated by the following equation:

$$t_{online}(C) = \underbrace{t_{online}(r) + t_{online}(C_1) + t_{online}(C_2)}_{t} + t_{online}(op)$$

where $t_{online}(op)$ is the time needed for computing the resulting approximate extension and $t_{online}(A_i), A_i \in \{r, C_1, C_2\}$ is the time need for computing an atomic extension. In fact, computations of atomic extensions can run parallel as they are independent of one another. By parallelizing these computations, we can speedup the overal computation in online processing.

Obviously, parallel computation of atomic extension comes with some overheads such as forking and joining threads and thread synchronization. The overhead of such a parallel computation should be insignificant as it is a simple parallelization without complex thread synchronization. In an ideal situation, we might obtain a better time $t(C)$ such that

$$t = \underbrace{\sup\{t_{online}(A_i) \mid A_i \in \{r, C_1, C_2\}\}}_{t_{||}} + t_{overhead} + t_{online}(op)$$

where $t_{overhead}$ is the time introduced by some overhead and $t_{||}$ is the execution time spent in the parallelised computation of atomic extensions.

This approach is defined in Algorithm 9. In line 3 to 4, we parallelise the computation of atomic classes and roles in which the parallelization is presented by an abstract function $||$. This function runs each computation of atomic extensions in parallel and joins the resulting extensions.

Algorithm 9: $\psi_{parallel}(KB,\mathsf{Ext}^C,\mathsf{Ext}^r,Q)$

```
    Input: $KB$: $\mathcal{SHIQ}$ knowledge base
    Input: $Q$: a complex query concept in NNF
    Input: $\mathsf{Ext}^C$: the materialised atomic concept extensions
    Input: $\mathsf{Ext}^r$: the materialised atomic role extensions
    Result: approximate extension of $Q$
1   //determine all atomic concepts and roles occuring in $Q$
2   $(C, R) \longleftarrow findAtomics(Q)$
3   $E^C \longleftarrow ||_{c_i \in C}$ Compute_Ext$(KB, c_i)$
4   $E^R \longleftarrow ||_{r_i \in R}$ Compute_Ext$(KB, r_i)$
5   return $\psi_{incremental}(KB,Ext^C \cup E^C,Ext^r \cup E^R,Q)$;
```

7.5.3 Incremental Maintance of Materialised Knowledge Bases

The database and offline variants require the whole materialisation of atomic extensions. We have shown that materialisation has been successfully applied in these variants to increase performance in computation of approximate extension at query time. The main aim of materialisation was to avoid computations of atomic extensions.

In general, materialisation has been applied successfully in many applications where reading access to data is predominant. For example, data warehouses usually apply materialisation techniques to make online analytical processing possible. Similarly, most web portals maintain cached web pages to offer fast access to dynamically generated web pages. We conjecture that reading access to ontologies is predominant in the Semantic Web and other ontology-based applications, hence materialisation seems to be a promising technique for fast query processing.

However, a materialisation process is time-consuming. Hence, the central problem that arises regarding materialisation is the maintenance of a materialisation when dealing with relatively large ontologies. As the computation of the materialisation is often complex and time consuming, it is desirable to apply more efficient techniques in practice, *i.e.*, to *incrementally* maintain a materialisation.

In order to effectively maintain a materialisation of an ontology and maximize the utility of the offline variant, we extend Algorithm 7 on page 90. The extended algorithm will be able to incrementally compute the materialisation of atomic extensions depending on queries being posed. In practice, this optimisation is important, as it is a particular necessity for handling ontologies having thousands of atomic classes and roles where the whole materialisation is impracticable. To realise this incremental approach, we extend the function Compute_Ext and Algorithm 7; however we pose the following two requirements:

- A materialisation should be only performed depending on queries being posed.
- A instance retrieval algorithm should have the effect that whenever the same query is repeated – it should not be evaluated and instead the previously computed answers should be returned.

To meet requirement 2, we adapt Algorithm 7 and denote it as $\psi_{incremental}$. The incremental approach is then defined in Algorithm 10. In order to meet requirement 1, we need to adapt the function Compute_Ext and the resulting (incremental extension) function $\mathsf{Compute\_Ext}_{incr}$ is defined in Algorithm 11. The main idea realised in this algorithm is first to retrieve the extension of an atomic class or role from the database. If the retrieval fails which means that there is no corresponding extension available, a DL reasoner is needed to compute it, as given in line 5 and 9.

Algorithm 10: $\psi_{incremental}(KB,\mathsf{Ext}^C,\mathsf{Ext}^r,Q)$

```
Input: $KB$: $\mathcal{SHIQ}$ knowledge base
Input: $Q$: a complex query concept in NNF
Input: $t$: a type indicating which Screech variant is to be used
Input: $\mathsf{Ext}^C$: the materialised atomic concept extensions
Input: $\mathsf{Ext}^r$: the materialised atomic role extensions
Result: approximate extension of $Q$
//check if the approximate extension of $Q$ is already computed.
$\langle Q \rangle \longleftarrow \pi_{[ind]}(\sigma_{[class=Q]}(\mathsf{Ext}^C))$
if $\langle Q \rangle = \varnothing$ then
    return $\psi_{mod}(KB,\mathsf{Ext}^C,\mathsf{Ext}^r,Q)$ // this function is the modified version of $\psi$ defined in
    Algorithm 7. It invokes $\mathsf{Compute\_Ext}_{incr}$ instead of Compute_Ext
else
    return $\langle Q \rangle$ //this yields the previously computed extension of $Q$ and thus meets the requirement 2.
return $\varnothing$;
```

To examine the effect of incremental materialisation, we run KAON2 and Algorithm 10 along with the function $\mathsf{Compute\_Ext}_{incr}$ to compute the approximate extensions of the 107 $\exists$-queries. The performance comparison is shown in Figure 7.4. As can be seen in this graphic, the algorithm is much slower than KAON2 at the first 15 queries because there is no materialisation for those queries available and a DL reasoner is to be invoked. The algorithm then starts running faster than KAON2 since from the 77^{th} query onwards, the performance of the algorithm is much better than that of KAON2.

```
Algorithm 11: Compute_Ext_incr (Ext^C, Ext^r, C)
Input: KB: $\mathcal{SHIQ}$ knowledge base
Input: C: an atomic class or role
Input: $\mathsf{Ext}^C$: the materialised atomic concept extensions
Input: $\mathsf{Ext}^r$: the materialised atomic role extensions
Result: conventional extension of C
$E \longleftarrow \emptyset$
if C is an atomic class then
    $E \longleftarrow \pi_{[ind]}(\sigma_{[class=C]}(\mathsf{Ext}^C))$
    if $E = \emptyset$ then
        Compute_Ext($KB$,C)
else if C is an atomic role then
    $E \longleftarrow \sigma_{[role=C]}(\mathsf{Ext}^r)$
    if $E = \emptyset$ then
        Compute_Ext($KB$,C)
return $E$;
```

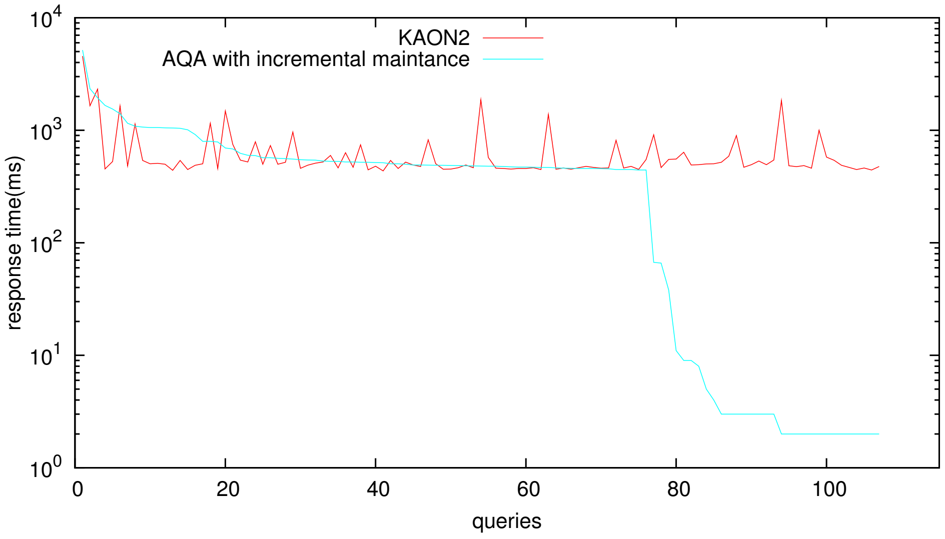


Figure 7.4: Measured performance of KAON2 and AQA with $\psi_{incremental}$ over 107 ∃-queries

7.6 Conclusion

In this chapter, an approach to approximate instance retrieval based on the notion of approximate concept extension has been introduced. This approach with its several variants such as online and offline in-memory and database variants is realised in a system called AQA. Furthermore, several optimisations have been developed to speed up performace in online and offline in-memory variants.

Performing a comprehensive experiment using the well-known, expressive benchmarking ontology WINE, all the variants in the AQA system have been evaluated. The purpose of this experiment was on the one hand to reveal how fast the approximate instance retrieval method is in comparison to the efficient ABox reasoner KAON2. On the other hand, it had to be determined the degree of soundness and completeness of the approximate method. The benchmark suite used for the evaluation contains over several thousand test queries. These queries were repeated at least 10 times in order to obtain proper experimental results. This evaluation has demonstrated that it is possible to achieve a significant performance improvement for ABox reasoning over expressive ontologies with large ABoxes and TBoxes. Moreover, the evaluation has shown that a significant speed-up of around 90% can be obtained. It has also been shown that for many practical complex queries the approximation approach yields fewer errors than indicated for the simple queries while speed-up in computation is still significant.

We conjecture that the approximate instance retrieval method as presented in this chapter will help to effectively reason over expressive ontologies with a large volume of instance data.

Chapter 8

Composed Anytime Algorithms

Chapter 2 provided a basis for analyzing the effects of approximate reasoning. In this chapter, this research will be further developed by analyzing how to construct anytime algorithms through the composition of approximate reasoning algorithms. Section 8.1 presents the concept of anytime algorithms, whereas Section 8.2 introduces the concept of oracle algorithms as a composition of approximate algorithms. Section 8.4 presents an anytime algorithm composed of the SCREECH approximations introduced in Chapter 6 and an anytime algorithm for AQA, which has been presented in Section 7.

8.1 Resource-bounded Reasoning and Anytime Algorithms

Resource-bounded reasoning [Zil96a] is an emerging field within artificial intelligence that is concerned with the construction of intelligent systems that can operate in real–time environments under uncertainty and limited computational resources. The need to employ resource-bounded reasoning techniques is based on a simple, but general, observation. In many complex domains, the computational resources required to reach an optimal decision reduce the overall utility of the result. This observation covers a wide range of applications such as medical diagnosis and treatment, combinatorial optimization, probabilistic inference, mobile robot navigation, and information gathering. What is common to all these problems is that it is not feasible (computationally) or desirable (economically) to compute the optimal answer.

For developing resource-bounded reasoning techniques, there have been proposed a number of approaches, the most popular one amongst others, is anytime algorithms. The term "anytime algorithm" was coined by Tom Dean in the late 1980's in the context of his work on time-dependent planning [BD89, DB88]. In many cases, a satisfying answer, which falls within the range of error tolerance and which is available at a certain point in time, is preferred to correct and best answer requiring arbitrary long time. Anytime algorithms are algorithms designed to conform to this practical requirement. They gradually improve the quality of their results, as computation time increases, and end with providing the whole answer when complete computation is required. Since the work of Dean and Boddy, the context in which anytime algorithms have been applied has broadened from planning and decision making to include problems from sensor interpretation to database manipulation, and the methods for utilising anytime techniques have become more powerful.

Based on anytime algorithms, Zilberstein has introduced a new programming paradigm to construct modular real-time system in artificial intelligence and demonstrated a number of applications[Zil96b, ZR96]. In this paradigm, a modular system is constructed by more flexible anytime algorithms, instead of standard algorithms that are typically implemented by a simple call-return mechanishm and have a fixed quality of the output. The core issue is to determine an optimal composition of anytime algorithms. This is solved by an offline compilation process in which the optimal allocation of time to the anytime components for any given total allocation and a run-time monitoring component that together generate a performance-maximizing profile for the complete system. However, a declarative characterisation of such algorithms is often lacking [Gtv04].

In the setting of logic and knowledge representation, the idea of applying anytime computation, is to define a family of entailment relations that approximate classical entailment, by relaxing soundness or completeness of reasoning. The knowledge base can provide partial solutions even if stopped prematurely; the accuracy of the solution improves with the time used in computing the solution and may eventually converge to the exact answer. From this point of view, anytime reasoning offers a compromise between the time complexity needed to compute answers by means of approximate entailment relations and the quality of these answers. Based on this paradigm, Schaerf and Cadoli [SC95] present a general technique for approximating deduction problems. Their framework includes a set of atomic propositions as a parameter, which captures the quality of approximation. Based on this parameter, the authors define two dual families of entailment relations, which are respectively sound but incomplete and complete but unsound with respect to classical entailment. Inspired by this idea, several extensions to approximating deduction problems have been proposed in the literature, including notably default logic and circumscription [CS96], modal logics [Mas98] and first-order logic [Kor01].

Anytime algorithms are important for the Semantic Web for two reasons. First, although many problems require a lot of resources (*e.g.,* time) to solve them, many systems can already produce good approximate solutions in a short amount of time. A system that can reason about how much time is needed to obtain an adequate result may be more adaptive in complex and changing environments. Second, a technique for reasoning about allocating time need not be restricted to the internals of a system. Web intelligent agents must be able to reason about how fast they and other agents can manipulate and respond to their environment. In realistic Semantic Web applications of expressive ontologies, the user may have to stop the execution of the algorithm, because there is no more time left for continuing the computation. Moreover, the time constraints may be unknown in advance, they can vary from seconds to hours or days in large-scale knowledge applications. In both cases, one may be interested to find a good, but not necessarily the optimal, set of approximate answers as quickly as possible. In this respect, the most important characteristics of anytime algorithms are outlined as follows as given in [GZ96]:

- **Measurable quality**: The quality of an approximate result can be determined precisely.
- **Predictability**: Anytime algorithms also contain statistical information about the output quality given in a certain amount of time as well as information about the data it receives. This information can be used for meta-reasoning about computational resources to eventually construct an anytime algorithm.
- **Recognisable quality**: The quality of an approximate result can easily be determined at run time.
- **Interruptibility and Continuation**: Anytime algorithms can be interrupted and return the partial results they have computed so far. Additionally, they can be continued beyond the contract time they are given.
- **Monotonicity**: Anytime algorithms always improve the output quality of the data they work on as soon as they are granted more time.

8.2 Concept of Composed Algorithms

Our focus is to develop anytime algorithms by combining the approximate algorithms introduced in Chapters 6 and 7 in a way that it gradually improves the quality of results by increasing the computation time.

Unlike standard algorithms that grant a fixed quality of output and completion time, anytime algorithms are meant to be interruptible at any time. In practice, particularly in the context of reasoning, it would not always be beneficial to develop algorithms that continuously generate intermediate answers whose quality is unknown. It would be more beneficial if the quality of the answers were known.

A composed anytime algorithm in this regard cannot be interrupted at any time, however, it can be interrupted at the point of time when one approximate algorithm involved in the composition finishes its computation. The advantage of such compositions is that one can be aware of the quality of intermediate answers, since soundness and completeness of the composed approximate algorithms are known. In the following, the formal definition of such a composed anytime algorithm is given.

Approximate reasoning systems do not have often anytime behaviour. However, it is possible to obtain anytime behaviour by composing approximate reasoning algorithms. Assume that a number of algorithms a_i ($i = 1, \ldots, n$) is given. Furthermore, assume there is an ORACLE ALGORITHM **c** whose behaviour can be described by a function $c : (X \times 2)^n \rightarrow X \times 2$ which combines a vector of outputs

$(a_1(\omega,t),\dots,a_n(\omega,t))$ of the algorithms a_i and yields a single output. Given an input ω, the invocation of all a_i in parallel and the subsequent call of the oracle algorithm yield a new algorithm $c_{a_1,\dots,a_n}$ with IO-function

$$f_{c_{a_1,\dots,a_n}}(\omega,t) = c(a_1(\omega,t),\dots,a_n(\omega,t)).$$

The definition just given is very general in order to allow for a very free combination, depending on the algorithms which are being combined. For the general setting, we impose only the very general constraint that for all $x_1,\dots,x_n \in X$ we have

$$c((x_1,1),\dots,(x_n,1)) = (x,1)$$

for some x, and also that the natural constraint discussed on page 45 on the corresponding IO-function $f_{c_{a_1,\dots,a_n}}$ is satisfied. This is just to ensure $\varrho_{c_{a_1,\dots,a_n}}(\omega) \leq \max\{\varrho_{a_1},\dots,\varrho_{a_n}\}$, i.e. the "combiner" indicates termination at the latest whenever all of the single input algorithms a_i do so.

It is more interesting to look at more concrete instances of oracles. Assume now that $a_1,\dots,a_{n-1}$ are one-answer algorithms and that a_n is an (always terminating) sound and complete algorithm. Let **c** be such that

$$c(a_1(\omega,\varrho_{a_n}(\omega)),\dots,a_{n-1}(\omega,\varrho_{a_n}(\omega)),a_n(\omega,\varrho_{a_n}(\omega))) = (f_{a_n}^{\text{res}}(\omega),1).$$

Then it is easy to see that $c_{a_1,\dots,a_n}$ is anytime.

If we know about soundness or completeness properties of the algorithms $a_1,\dots,a_{n-1}$, then it is also possible to guarantee that $c_{a_1,\dots,a_n}$ is monotonic anytime. This can be achieved in several ways, and we give one specific example based on ABox reasoning in description logics.

Assume that each input consists of a class description C over some description logic L, and each output consists of a set of (named) individuals. For constructing an oracle from such algorithms, we will actually consider as outputs *pairs* (A,B) of sets of individuals. Intuitively, A contains only individuals which are *known* to belong to the extension of C, while B constitutes an individual set which *is known to contain* all individuals in the extension of C. A single output (set) A can be equated with the output pair (A,A). Now let $a_1,\dots,a_n$ be sound[1] but incomplete[2] one-answer algorithms over L, let $b_1,\dots,b_m$ be complete but unsound one-answer algorithms over L and let a be a sound, complete and terminating algorithm over L, i.e. $f_a^{\text{res}}(C,\varrho_a)$ – which we denote by C_a – contains exactly all named individuals that are in the extension of C as a logical consequence of the given knowledge base. Under this assumption, we know that $f_{a_i}^{\text{res}}(C,\varrho_{a_i}) = (C_{a_i},\mathbf{I})$ and $f_{b_j}^{\text{res}}(C,\varrho_{b_j}) = (\varnothing,C_{b_j})$ for some sets C_{a_i} and C_{b_j}, where $\mathbf{I}$ stands for the set of all (known) individuals, and furthermore we know that $C_{a_i} \subseteq C_a \subseteq C_{b_j}$ for all i,j.

[1] We mean soundness in the following sense: If the set I of individuals is the correct answer, then the algorithms yields as output a pair (A,A) of sets with $A \subseteq I$.

[2] We mean completeness in the following sense: If the set I of individuals is the correct answer, then the algorithms yields as output a pair (A,A) of sets with $I \subseteq A$.

The oracle $\mathbf{c}$ is now defined as follows.

$$\mathbf{c}(a_1(C,t),\dots,a_n(C,t),b_1(C,t),\dots,b_m(C,t),a(C,t))$$
$$= \begin{cases} ((f_a^{\text{res}}(C,t), f_a^{\text{res}}(C,t)),1) & \text{for } t \geq \varrho_a(C), \\ ((\mathit{upper},\mathit{lower}),\mathit{term}) & \text{for } t < \varrho_a(C) \\ \quad \text{where } \mathit{lower} = \bigcup_{(A_i,B_i,1)=f_{a_i}(C,t)} A_i, & \\ \qquad \mathit{upper} = \bigcap_{(A_j,B_j,1)=f_{b_j}(C,t)} B_j, & \\ \qquad \mathit{term} = 1 \text{ if } \mathit{lower} = \mathit{upper}, \text{ otherwise } 0. & \end{cases}$$

Note that the empty set union is by definition the empty set, while the empty set intersection is by definition I.

Intuitively, the oracle realizes the following behavior: If the sound and complete sub-algorithm has terminated, display its result. Beforehand, use the lower resp. upper bounds delivered by the sound resp. complete algorithms to calculate one intermediate lower and one intermediate upper bound. If those two happen to coincide, the correct result has been found and may terminate without waiting for a's termination. This *squeezing in* of the correct result now also explains why we have chosen to work with pairs of sets as outputs.

As error function, we might use the sum of the symmetric difference between A and A_0, respectively between B and A_0, i.e.

$$e((A,B),(A_0,A_0)) = |A_0 \setminus A| + |B \setminus A_0|.$$

We could also use a value constructed from similar intuitions like precision and recall in information retrieval, but for our simple example, this error function suffices. It is indeed now straightforward to see that $\mathbf{c}_{a_1,\dots,a_n,b_1,\dots,b_m,a}$ is monotonic anytime. It is also clear that $\mathbf{c}_{a_1,\dots,a_n,b_1,\dots,b_m,a}$ is more precise than any of the a_i and b_j, at all time points.

8.3 An Example of the Composed Algorithms

In this section, we will instantiate the very general framework established in the preceding sections. We will use the presented techniques to compare three approximate reasoning algorithms and compose a (simple) anytime algorithm following the example at the end of Section 8.2.

Consider the three algorithms SCREECH-ALL, SCREECH-NONE and KAON2, as discussed in Section 6.3 on page 60 and Section 6.4 on page 65. It was introduced that SCREECH-ALL is complete but unsound, SCREECH-NONE is sound but incomplete, and KAON2 is sound and complete. Following the general framework, we first have to stipulate the probability space (Ω, P) for our case. Here we introduce the first simplifying assumptions, which are admittedly arguable, but will suffice for the example:

- The task considered here is instance retrieval for named classes.

- The knowledge base is the Wine ontology. For more detailed evaluation data on this knowledge base, consider Table 6.6 on page 75 and Table 6.9 on page 79.
- As queries, we consider only instance retrieval tasks, i.e. given an atomic class description, we query for the set of individuals which can be inferred to be instances of that class. Hence Ω – the query space – consists of named classes $\mathbf{C}$ of the Wine ontology the instances of which are to be retrieved: $\Omega = \mathbf{C}$. Examples for named classes in this ontology are *e.g.,* `Chardonnay`, `StEmilion` or `Grape`.
- All those instance retrieval queries $\omega \in \Omega$ are assumed to be equally probable to be asked to the system, hence

  $$P(\omega) = \frac{1}{|\mathbf{C}|} \text{ for all } \omega \in \Omega.$$

  Obviously, the probability of a query could also be assumed differently, *e.g.,* correlating with the number of instances the respective class has. Nevertheless, for the sake of simplicity we will stick to the equidistributional approach.

Obviously, the output space X consists of subsets of the set of individuals $\mathbf{I}$ from the Wine ontology together with the no-output symbol $\bot$: $X = 2^{\mathbf{I}} \cup \{\bot\}$. As the error function e comparing an algorithm's output I with the correct one I_0, we use the inverted value of the common f-measure, i.e.

$$e(I, I_0) := 1 - \frac{2 \cdot \mathit{precision} \cdot \mathit{recall}}{\mathit{precision} + \mathit{recall}}$$

where (as usual)

$$\mathit{precision} := \frac{|I \cap I_0|}{|I|} \qquad \text{and} \qquad \mathit{recall} := \frac{|I \cap I_0|}{|I_0|}.$$

According to the proposed handling of $\bot$, we stipulate the overall "worst-case distance": $e(\bot, I_0) = 1$ for all $I \subseteq \mathbf{I}$.

As mentioned before, the set $\mathcal{A}$ of considered algorithms comprises three items:

$$\mathcal{A} = \{\text{KAON2, SCREECH-ALL, SCREECH-NONE}\}$$

For each of these algorithms, we carried out comprehensive evaluations: we queried for the class extensions of each named class and stored the results as well as the time needed. By their nature none of the considered algorithms exhibits a genuine anytime behavior, however, instead of displaying the "honest" $\bot$ during their calculation period, they could be made to display an arbitrary intermediate result. It is straightforward to choose the empty set in order to obtain better results: most class extensions will be by far smaller than half of the individual set, hence the distance of the correct result to the empty set will be a rather good guess.

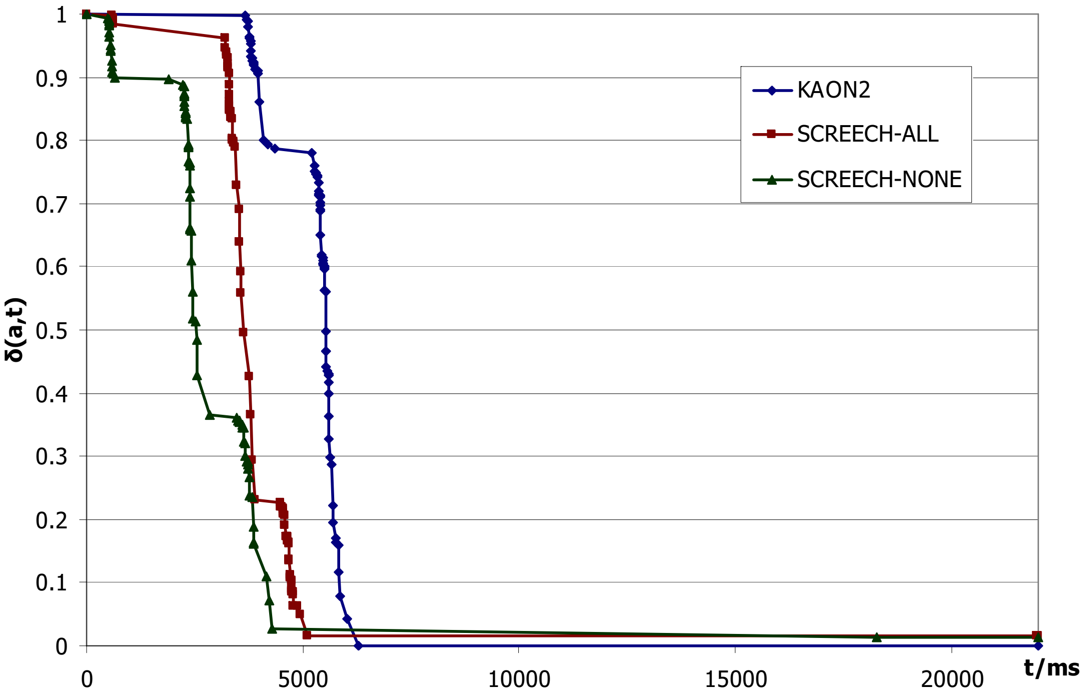


Figure 8.1: Defect over time

Hence, for any algorithm a of the above three and any class name C let I_C denote be the set of retrieved instances and t_C denote the measured runtime for accomplishing this task. Then we can define the IO-function as

$$f_a(C,t) = \begin{cases} (\emptyset, 0) \text{ if } t < t_C \\ (I_C, 1) \text{ otherwise.} \end{cases}$$

The values of the correct output function f_0 can be found via KAON2, as this algorithm is known to be sound and complete. Moreover, the runtime functions $\rho_a(C)$ of course coincide in our case with the runtimes t_C measured in the first place. Since all of the considered algorithms are known to terminate, no ρ_a will ever take the value ∞.

After this preconsiderations, we are ready to carry out some calculations estimating the quality of the considered algorithms. Figure 8.1 shows a plot depicting the decrease of the defect for all the three algorithms. As expected, there is an ultimate defect for the two Screech variants, namely 0.013 for SCREECH-NONE and 0.015 for SCREECH-ALL, i.e. with respect to the terminology introduced earlier, we can say that SCREECH-NONE is more precise than SCREECH-ALL. While the defect of KAON2 is initially greater than those of the Screech variants, it becomes better than them at about 6 seconds and decreases to zero defect after about 7 seconds. In other words, SCREECH-ALL is more precise than KAON2 at all time points less than 6 seconds. A first conclusion from this would be: if a user is willing to wait for 7 seconds for an answer (which then would be guaranteed to be correct) KAON2 would be the best choice, otherwise (if time is crucial and precision not), SCREECH-ALL might be a better choice as it shows the quickest defect decrease. If we now assume a time-critical application where responses coming in later than, say, 5 seconds are ignored, we can describe this by the fact that SCREECH-ALL is better than KAON2 with respect to the density function

$$\bar{f}(x) = \begin{cases} 1 & 0 \leq x \leq 5, \\ 0 & \text{otherwise.} \end{cases}$$

Considering the fact that SCREECH-ALL is complete, SCREECH-NONE is sound, and KAON2 is both, we can now utilise a variant of the oracle given in Section 8.2 in the example. The behaviour of the combined algorithm can in this simple case be described as follows. It indicates termination whenever one of the following occurs:

- KAON2 has terminated. Then the KAON2 result is displayed as solution.
- Both SCREECH-ALL and SCREECH-NONE have terminated with the same result. In this case, the common result will be displayed as the final one.

If none of above is the case, the experimental findings suggest to choose the SCREECH-NONE result as intermediate figure. The algorithm obtained that way is anytime and more (or equally) precise than any of the single algorithms at all time points.

8.4 Composed Anytime Reasoners

This section presents the underlying concept of two anytime reasoners composed of the approximate algorithms developed in Chapters 6 and 7.

8.4.1 SCREECH-Anytime

This section presents a composed anytime reasoner by using the SCREECH variants introduced in Chapter 6 and will show how to improve quality of approximation while preserving completeness.

Before the composed algorithm will be discussed, please recall the properties of the SCREECH variants to be combined. SCREECH-ALL is the approximation that may be unsound, but complete, while SCREECH-NONE is sound, but may be incomplete. In the composed algorithm, two instances of SCREECH-NONE and one instance of SCREECH-ALL are combined. The resulting anytime algorithm will produce answers at three discrete times, and end up with building a complete answer set. In general, this composed algorithm is a non-monotonic anytime algorithm. One cannot predict which variant will terminate at first because it critically depends on the types of knowledge bases and queries. To better understand this idea, let us consider a Venn diagram illustrated in Figure 8.2. This diagram depicts the answer sets that will be

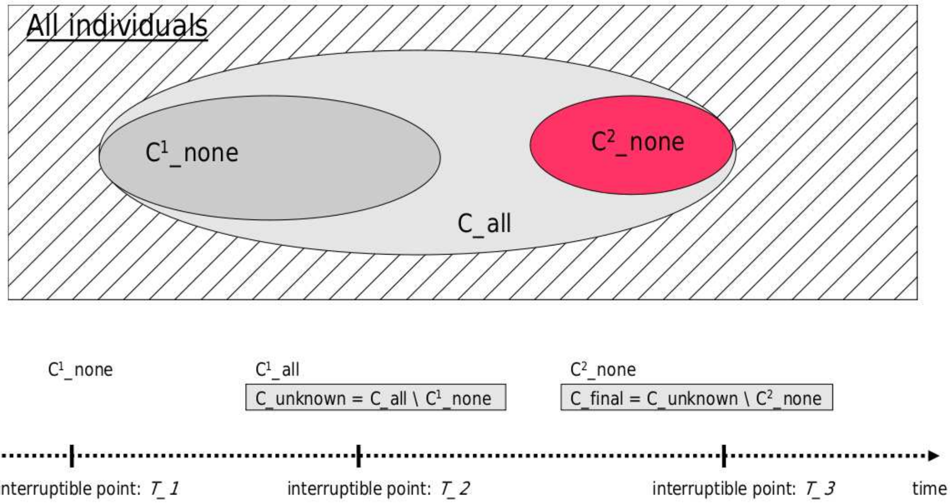


Figure 8.2: A combination of the three SCREECH approximate algorithms

obtained from three SCREECH instances. Here, the rectangle represents the universal set, *i.e.,* the set of all individuals defined in the ABox of the ontology under consideration. The answer set obtained by the first instance of SCREECH-NONE for querying a named class C is denoted C^1_{none}. The answer set obtained by the other instance of SCREECH-NONE for querying $\neg C$ is denoted C^2_{none}, whereas the set obtained by the instance of SCREECH-ALL is denoted C_{all}. Note that the composed algorithm is not interruptible at any time, but they are interruptible at the time when one approximate algorithm involved in the composition finishes its computation and is able to continue.

Assume that the first instance of SCREECH-NONE is faster than the others. Due to the soundness of SCREECH-NONE, the first instance always provides the correct individuals for C, but, its answer set is incomplete. The question that arises is then how to determine other correct individuals that cannot be computed by the above approximation. Assuming that the instance of SCREECH-ALL is the next fast one, it will provide an answer set, which contains all the individuals belonging to the extension of C. The answer set is complete, but can contain more individuals, *e.g.,* incorrect individuals, due to the unsoundness. Furthermore, by excluding the correct individuals from this set, one can determine at that time which elements can be potentially incorrect individuals. The resulting set is denoted $C_{unknown}$, thus:

$$C_{unknown} = C_{all} \setminus C^1_{none}.$$

Now, the main issue is how to determine the remaining correct individuals. In fact, one cannot determine all the correct individuals, so the aim is to compute an answer set with as few incorrect individuals as possible. To achieve this, we have made use of the fact that the complement of a class includes all the individuals that are not members of that class. This means, by using the second instance of Screech-None for querying $\neg C$, one can determine some of the incorrect individuals, so the anytime algorithm improves its quality by excluding those incorrect individuals from $C_{unknown}$. The final answer set C_{final} contains all the individuals that are known to belong the extension of C, thus:

$$C_{final} = C_{unknown} \setminus C^{2}_{none}.$$

This final answer set denotes the consolidated answer of the anytime algorithm after the composed approximate algorithms have finished their computations.

8.4.2 AQA-Anytime

In the previous section, it has been discussed how to construct a composed anytime algorithm without involving a sound and complete reasoner. This has been demonstrated by combining the Screech variants. In the following, we discuss the development of an anytime reasoner called AQA-Anytime, which combines an AQA reasoner with a sound and complete DL reasoner. Consider the soundness and completeness properties of AQA from Section 7.2, Chapter 7. It has been revealed that AQA is fast and its soundness and completeness depends on the type of queries. Moreover, Proposition 7.2 on page 84 states that, for queries that do not make use of the $\forall$, $\geq$ and $\leq$ constructs (after normalisation [3]), the AQA approach of approximating concepts in their negation normal form yields an extension that might miss some instances but has no improper instances in it. In other words, computing the approximate extension is *sound* with respect to the conventional extension.

Regarding this, we introduce a function isSound, which determines whether or not a concept in the negation normal form is built from the $\forall$, $\geq$ and $\leq$ constructs. This function assigns a Boolean value of $\{\mathbf{F}, \mathbf{T}\}$ to each concept expression and is recursively defined as follows:

$$\begin{array}{lcl@{\qquad}lcl}
\mathsf{isSound}(\mathsf{C} \sqcap \mathsf{D}) & = & \mathsf{isSound}(\mathsf{C}) \wedge \mathsf{isSound}(\mathsf{C}) & \mathsf{isSound}(\forall \mathsf{R.C}) & = & \mathbf{F} \\
\mathsf{isSound}(\mathsf{C} \sqcup \mathsf{D}) & = & \mathsf{isSound}(\mathsf{C}) \wedge \mathsf{isSound}(\mathsf{C}) & \mathsf{isSound}(\top) & = & \mathbf{T} \\
\mathsf{isSound}(\exists \mathsf{R.C}) & = & \mathsf{isSound}(\mathsf{C}) & \mathsf{isSound}(\bot) & = & \mathbf{T} \\
\mathsf{isSound}(\geq n\, R.C) & = & \mathbf{F} & \mathsf{isSound}(\mathsf{A}) & = & \mathbf{T} \\
\mathsf{isSound}(\leq n\, R.C) & = & \mathbf{F} & \mathsf{isSound}(\neg \mathsf{A}) & = & \mathbf{T}
\end{array}$$

The AQA-Anytime reasoner is composed of two reasoners, so there is only one point in time to interrupt it when the AQA reasoner responds faster than a DL reasoner. Here, we consider an intermediate result of the AQA-Anytime reasoner. Assuming that an AQA reasoner first responds at time T_{aqa}, one can determine the logical property of AQA by applying isSound to the negation normal form $\mathsf{NNF}(Q)$ of a

[3] Concept expressions are assumed to be negated normal form (NNF) before processing them with AQA

given query Q. If $\mathsf{isSound}(\mathsf{NNF}(\mathsf{Q})) = \mathbf{T}$, all the individuals in the answer set provided by AQA will be correct due to its soundness. However, the answer set may not contain all the individuals known to be correct due to the incompleteness. Since a sound and complete DL reasoner is utilized in the AQA-Anytime reasoner, possibly missing individuals can be found, so the AQA-Anytime reasoner results in a sound and complete answer set. In case the query has been built from the $\forall$, $\geq$ and $\leq$ constructs, however, we cannot determine at the time T_{aqa} which individuals are incorrect ones due to the unsoundness of AQA. Hence, in general, the AQA-Anytime is, like SCREECH-Anytime, a non-monotonic anytime reasoner as well. The implementation of the AQA reasoner and its integration into the reasoner broker system will be further discussed in Chapters 9 and 10.

8.5 Conclusion

Applying anytime computation is an important aspect in the Semantic Web as it is connected with the field of resource-bounded reasoning. In this chapter, we analyzed how to construct anytime algorithms by composing approximate reasoning algorithms. With regard to this, it has been studied how to combine the approximate reasoning approaches presented in Chapters 6 and 7. The main goal of this study has been to investigate the synergy effects of these approximate approaches with different computational characteristics.

Unlike standard algorithms that grant a fixed quality of output and completion time, anytime algorithms are meant to be interruptible at any time. In practice, particularly in the context of reasoning, it would not always be beneficial to develop algorithms that continuously generate intermediate answers whose quality is unknown. It would be more beneficial if the quality of the answers were known. The composed algorithms discussed in this chapter are not interruptible at any time, but they are interruptible at the time when one approximate algorithm involved in the composition finishes its computation and is able to continue. The advantage of such compositions is that one can be aware of the quality of intermediate answers, since soundness and completeness of the composed approximate algorithms are known. The development of composed anytime algorithms requires correct synchronization and maintenance of intermediate answers, and thus, a commonly useful anytime reasoning pattern will be designed and discussed in Chapter 9.

Part III

Implementation and Applications

Chapter 9

Implementation

Having introduced the approximate reasoning algorithms for instance retrieval in the previous chapters, in this chapter, their implementations are presented. Firstly, the general architecture of the framework, which interconnects the approximate reasoning components in a flexible environment, is introduced in Section 9.1. Subsequently, the individual components of the framework are discussed: Section 9.2 describes the design and implementation of the SCREECH knowledge compilation component. Section 9.3 presents the design and implementation of the AQA reasoning component. In Section 9.4, the design and implementation of the anytime reasoners are described. Finally, Section 9.5 introduces the implementation of the framework for the evaluation of approximate reasoning systems.

9.1 General Architecture

The approximate reasoning algorithms for instance retrieval of expressive ontologies described in this work have been implemented in the *Approximate Ontology Reasoning Workbench* (AORW). AORW is an open source tool and hosted at http://sourceforge.net/projects/aorw. Figure 9.1 shows the overall architecture of AORW. The SCREECH component implements the knowledge compilation algorithms from Chapter 6 whereas AQA implements the instance retrieval algorithms from Chapter 7. Furthermore, AORW includes an automated benchmarking tool, which is a concrete implementation of the framework to evaluate approximate reasoning systems discussed in Chapter 5.

As a combination of approaches, two anytime reasoners have been implemented, namely SCREECH-Anytime and AQA-Anytime. They provide an implementation of the composed anytime algorithms discussed in Chapter 8. Composing the SCREECH approximation variants, SCREECH-Anytime reasoner enables anytime instance retrieval for named classes. In contrast, an AQA-Anytime reasoner combines the AQA variants with a DL reasoner, enabling anytime instance retrieval for complex classes.

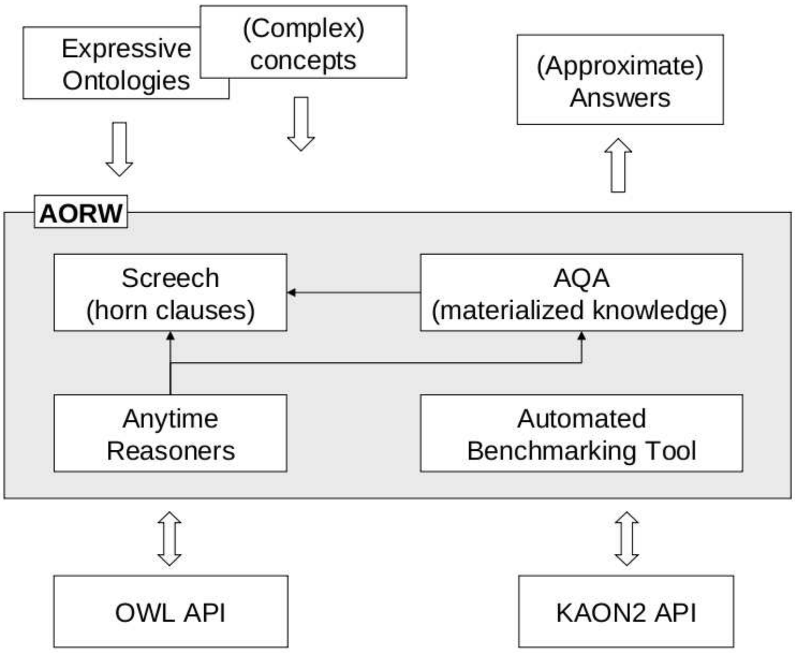


Figure 9.1: AORW system architecture

AORW has been completely implemented in Java and has made use of the KAON2 API to access OWL ontologies and to invoke the reasoning procedures provided by KAON2. The commonly used OWL API has been also used by AQA to handle expressive ontologies, which cannot be processed by KAON2. Its primary goal, however, is to invoke other DL reasoners for the materialization of atomic extensions, which is required by AQA.

A thorough design and implementation is important for developing an approximate reasoning system, since a bad design and its incorrect implementation could affect reasoning performance as well as approximation quality. To ensure a certain level of quality, the whole implementation has been guided by the well-acknowledged design pattern[1] from Software Engineering [Gam08]. Regarding this, we have made use of several well-known design patterns to develop the present framework, which are shown in Table 9.1. They help improving the quality of the software in terms of the software being reusable, maintainable and extensible. In the next sections, we will describe the individual components of AORW in detail and use those design patterns to describe their implementation.

[1] Design pattern are recurring solutions to problems that arise during the life of a software application.

Pattern Name	Description
Abstract Factory	Allows the creation of an instance of a class from a suite of related classes without having a client object to specify the actual concrete class to be instantiated.
Bridge	Allows the separation of an abstract interface from its implementation. This eliminates the dependency between the two, allowing them to be modified independently.
Composite	Allows both individual objects and composite objects to be treated uniformly.
Strategy	Allows each of a family of related algorithms to be encapsulated into a set of different subclasses (strategy objects) of a common superclass. For an object to use an algorithm, the object needs to be configured with the corresponding strategy object. With this arrangement, algortithm implementation can vary without affecting its clients.
Visitor	Allows an operation to be defined across a collection of different objects without changing the class of objects on which it operates.

Table 9.1: The design patterns used for developing the AORW framework

9.2 Implementation of the SCREECH Approach

Figure 9.2 shows a UML class diagram of the Screech component's layout in terms of Java classes and interfaces. The SCREECH component implements three algorithms, which correspond to the three SCREECH approximation variants. The algorithms for these approximations were discussed in Section 6.3 on page 60 and Section 6.4 on page 65.

The implementation of these algorithms follows the Strategy pattern. This pattern suggests keeping the implementation of each of the algorithms in a separate class, which is often referred to as a *strategy* class. In the SCREECH component, three strategy classes, namely `ScreechAll`, `ScreechOne`, `ScreechNone`, were designed, which implement Algorithm 2, Algorithm 4, and Algorithm 5 described in Section 6.3 and Section 6.4, respectively. The `Screech` class is the main reasoner class implementing the `ScreechApproximateReasoner` interface. Due to the Strategy pattern, changing the behavior of a reasoner is simply a matter of changing its Strategy object to the one that implements the required algorithm. To enable a `Screech` object to access different Strategy objects in a seamless manner, all Strategy classes have been designed to offer the same interface named `ScreechApproximation`. Selecting and instantiating an appropriate `ScreechApproximation` class and configurating it with the selected instance

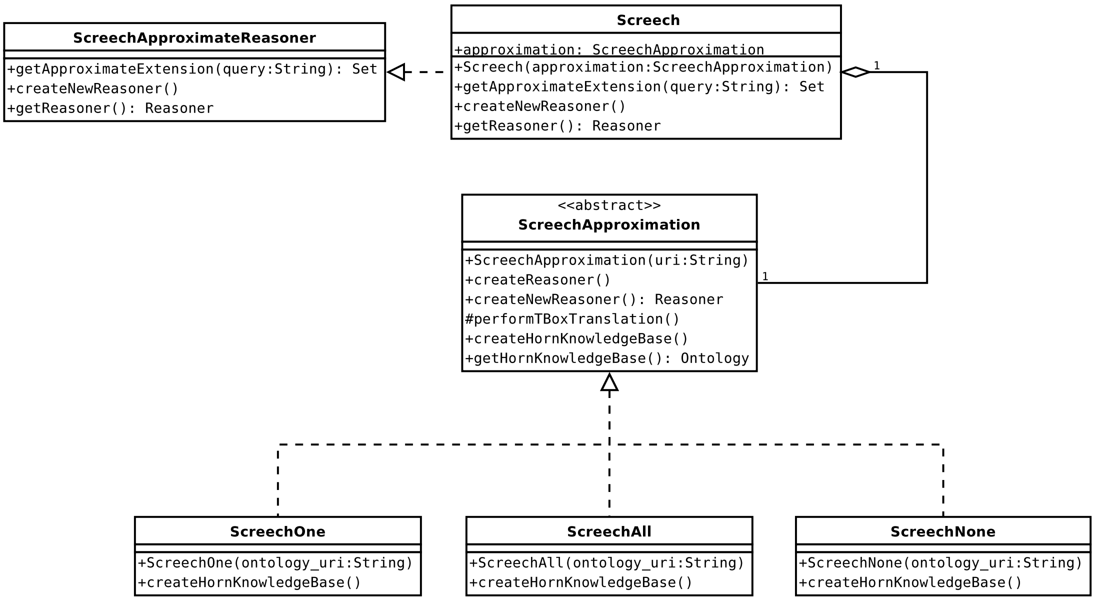

ScreechApproximateReasoner
+getApproximateExtension(query:String): Set
+createNewReasoner()
+getReasoner(): Reasoner
Screech
+approximation: ScreechApproximation
+Screech(approximation:ScreechApproximation)
+getApproximateExtension(query:String): Set
+createNewReasoner()
+getReasoner(): Reasoner
1
<<abstract>>
ScreechApproximation
+ScreechApproximation(uri:String)
+createReasoner()
+createNewReasoner(): Reasoner
#performTBoxTranslation()
+createHornKnowledgeBase()
+getHornKnowledgeBase(): Ontology
1
ScreechOne
+ScreechOne(ontology_uri:String)
+createHornKnowledgeBase()
ScreechAll
+ScreechAll(ontology_uri:String)
+createHornKnowledgeBase()
ScreechNone
+ScreechNone(ontology_uri:String)
+createHornKnowledgeBase()

Figure 9.2: A Class diagram for the SCREECH component

can alter the behavior of a `Screech` object.

This type of arrangement completely separates the implementation of a SCREECH approximation algorithm from the SCREECH reasoner that uses it. As a result, when an existing algorithm implementation is changed or a new approximation algorithm is added, the `Screech` object remains unaffected.

Among the SCREECH approximations, the SCREECH-ONE is the most complex one, which, in general, can apply various heuristics to produce a subset of the split program derived from a disjunctive rule. According to Algorithm 4 on page 66, it chooses only one Horn rule from a split program whose head has a greater extension than the other heads of the Horn rules. To let SCREECH be extendable, an abstract class `Heuristic` has been introduced, which contains the most common code to select Horn rules. To implement more complex algorithms for selecting Horn rules, other resources might be required to read heuristic information. For instance, heuristic information can be read from a file or a database system.

SCREECH component serves two purposes. One is to use it as a knowledge compilation method for other DL reasoners, the other is to use it as an approximate reasoner for instance retrieval. As reasoner, it makes use of the KAON2 reasoner and performs the following tasks to compute approximate extensions of named classes: creating an instance of a strategy class, creating a `Screech` object and configuring it with the object of the selected strategy class, and invoking the `getApproximateExtension` on the reasoner object. As knowledge compilation, it performs the following tasks to compile the given ontology into a Horn knowledge base: creating an instance of a strategy class and invoking `createHornKnowledgeBase` and `getHornKnowledgeBase`. The `getHornKnowledgeBase` method provides an ontology object which can be further serialized as an OWL ontology.

Note that the UML diagram only illustrates some important classes and interfaces, which constitute the SCREECH component. The underlying API includes a number of functionalities for accessing ontologies and a database system.

9.3 Implementation of the AQA Approach

This section presents the AQA implementation details. AQA is an approximate reasoner designed to provide efficient instance retrieval for DL complex concepts. The core functionality of this system is to compute approximate extensions, which has been implemented in several variants according to the algorithms defined in Section 7.3, Chapter 7. Figure 9.3 illustrates a UML class diagram of the AQA component's layout in terms of Java classes and interfaces.

9.3.1 Computation Of Approximate Extensions

In AQA, as discussed in Section 7.3 on page 85, there are two distinguished variants, namely database and in-memory computation, to compute approximate extensions. In the first case, computation is delegated to underlying database operations, whereas in the second case, it is performed in computer memory. The algorithms for these computations were defined in Table 7.2 on page 88 and Algorithm 7 on page 90. The first operation of these algorithms is to break down a given complex query to the atomic concepts and roles occurring in it. KAON2 API[2] already provides a class hierarchy of DL concept expressions, so it is used to implement AQA. From the implementation point of view, breaking a complex concept down to atomic ones means to design an operation across a heterogeneous collection of objects of the class hierarchy. The Visitor pattern is a commonly useful pattern to model such an operation. KAON2 API offers a `KAON2Visitor` interface to traverse the class hierarchy. Moreover, it implements that every DL object (a DL axiom or expression) has a method `accept(KAON2Visitor)`, which makes a call to the respective `visit` method, passing itself as an argument.

The `KAON2Visitor` interface is used to design two classes, `DatabaseMapping` and `InMemoryComputation`, for computing approximate extensions in the database variant as well as in-memory variant. This interface provides them with the means to perform the required computation on every concept. The `KAON2Visitor` interface, however, declares several methods, which are not required by these classes. In order to improve readibility and to provide a better design, an abstract `AbstractDLConceptVisitor` class has been designed to separate the unnecessary

[2] Its current release supports the $\mathcal{SHIQ}$ DL language.

methods declared in the `KAON2Visitor` interface. Moreover, the common functionality for both the `InMemoryComputation` and `DatabaseMapping` classes is placed in this class. Although these classes share the `KAON2Visitor` interface, their implementation is quite different from each other. According to the algorithm defined in Table 7.2, a DL expression is mapped to a SQL expression. In case of a complex DL expression, the corresponding SQL expression would become verbose, and thus make testing and debugging more difficult. Another issue is that the execution time for evaluating a long SQL expression could cause bad performance. Hence, an efficient implementation had to be undertaken. As a result, it has been decided to design the `DatabaseMapping` class such that it yields a list of SQL expressions, instead of generating just one long SQL expression. Each expression in that list corresponds to each sub-concept of the query being processed. The actual approximate extension will be then computed by sequentially evaluating each SQL expression from that list.

Note that SQL expressions are generated such that the result of its evaluation is stored in a temporary database table, and an evaluation reads the result of its previous evaluation accessing the corresponding database table. This implementation has been tested with a number of JUnit[3] tests. Another test strategy for this implementation has been to compare it with the other variants.

In contrast to the implementation of the `DatabaseMapping` class, the `visit` methods in the `InMemoryComputation` class implement Algorithm 7, returning a `Set` object, which represents the approximate extensions of the sub-concepts. For atomic concepts and roles, the `InMemoryComputation` class requires the computation of their extensions. As discussed in Section 7.3, there are two variants for this computation – offline and online. In the offline variant, an atomic extension is calculated by retrieving the database, which stores the materialized knowledge whereas in the online variant, it is calculated by invoking a DL reasoner.

It has been decided to implement this process of computing atomic extensions, *i.e.,* the separation between the online and offline variants, in another class outside the `InMemoryComputation` class. The advantage of this design-related decision is that the `InMemoryComputation` class only contains the computation of approximate extensions without dealing with the case analysis of AQA variants. For such an implementation, the Bridge pattern is a commonly useful pattern. Applying this pattern, an interface named `ExtensionManagerInf` and its implementer class `ExtensionManager` have been designed. The advantage of providing this interface is that one can implement another complex extension manager without affecting the `InMemoryComputation` class. The `ExtensionManager` class controls the variant for computing atomic extensions chosen in AQA. The design has been developed in such a way to make the choice dependent on the given query. Certainly, the variant of AQA can be changed at runtime, moreover, it can be changed depending on individual queries, which allows more flexibility in a sophisticated application of expressive reasoning. For this purpose, the `ExtensionManager` couples a `DLReasoner` as well as an `ExternalDatabase`

[3] `http://www.junit.org/` [accessed 2009-8-13]

object in a seamless manner and also provides functionalities for materializing knowledge bases.

The design has been developed in such a way that an AQA reasoner is configured with a DL reasoner and an external database encaplused into an `ExtensionManager` object. Furthermore, it has been designed that an AQA reasoner can switch its processing variant depending on queries. That means, invoking the `getApproximateExtension` method, the user can specify in which variant the approximate extension ought be computed, without instantiating a new AQA reasoner. To distinguish the different variants in AQA, different types of constant data are used.

In order to compute the approximate extension of a given complex query, a client program can perform the following tasks: creating an instance of the `ExtensionManager` by passing a `DLReasoner` object as well as an `ExternalDatabase` object, creating an `AQAReasoner` and configuring it with the `ExtensionManager` object created above and invoking the `getApproximateExtension` method on the `AQAReasoner` object by specifying in which variant the approximate extension is to be computed.

9.3.2 Embedding DL Reasoners and Databases

A DL reasoner is required in the online variant to compute atomic extensions at query-time. Furthermore, in the offline variant as well as in the database-based variant, the materialization of atomic extensions has to be achieved using a DL reasoner. KAON2 is the standard DL reasoner integrated in AQA. It is, however, practical, to integrate other efficient reasoners like Pellet. In order to make this possible, the common methods for instance retrieval offered by different DL reasoners have been abstracted and declared as a separate interface named `DLReasoner`. In case of using another DL reasoner, this interface can be used by a concrete reasoner class. This enables AQA to use different types of DL reasoner objects in a seamless manner without requiring any changes, so the `ExtensionManager` class does not need to be altered even when integrating a new DL reasoner.

Except for the online variant, other variants require a database system to store atomic extensions as well as experimental results. The relational database system MySQL is used as the default storage system. In order to have other efficient database systems integrated in AQA, an interface named `ExternalDatabase` has been designed, which offers the functionalities required by an `ExtensionManager` object. An implementer object of the `ExternalDatabase` interface can be configured with an object, which contains information, such as how to access the database as well as to store extensions.

9.3.3 Applying Knowledge Compilation

In Section 6.5, Chapter 6, we have discussed that SCREECH speeds up reasoning significantly while preserving soundness and completeness. That has been the case of

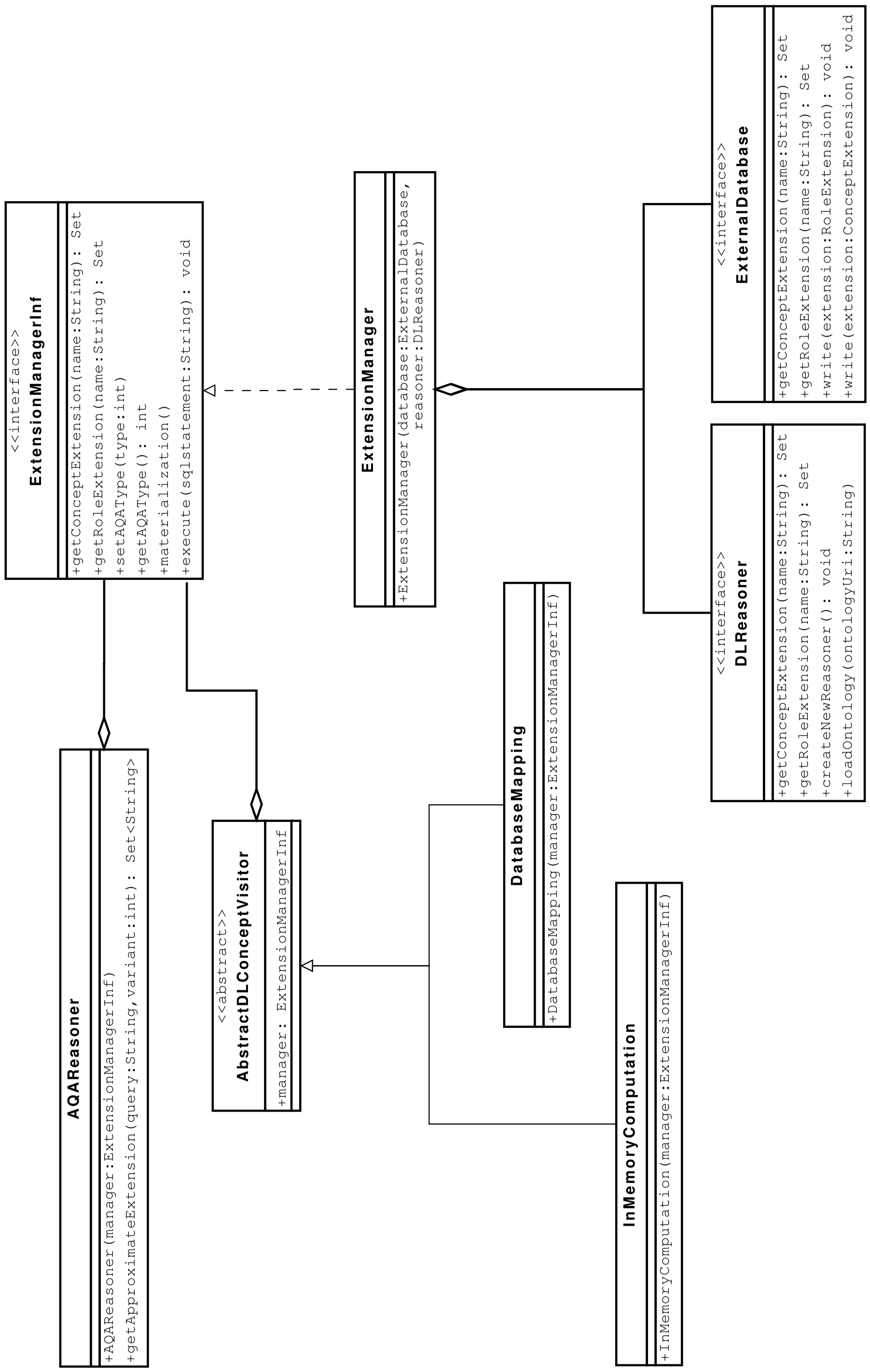

<<interface>>
ExtensionManagerInf
+getConceptExtension(name:String): Set
+getRoleExtension(name:String): Set
+setAQAType(type:int)
+getAQAType(): int
+materialization()
+execute(sqlstatement:String): void
ExtensionManager
+ExtensionManager(database:ExternalDatabase, reasoner:DLReasoner)
<<interface>>
ExternalDatabase
+getConceptExtension(name:String): Set
+getRoleExtension(name:String): Set
+write(extension:RoleExtension): void
+write(extension:ConceptExtension): void
<<interface>>
DLReasoner
+getConceptExtension(name:String): Set
+getRoleExtension(name:String): Set
+createNewReasoner(): void
+loadOntology(ontologyUri:String)
AQAReasoner
+AQAReasoner(manager:ExtensionManagerInf)
+getApproximateExtension(query:String,variant:int): Set<String>
<<abstract>>
AbstractDLConceptVisitor
+manager: ExtensionManagerInf
DatabaseMapping
+DatabaseMapping(manager:ExtensionManagerInf)
InMemoryComputation
+InMemoryComputation(manager:ExtensionManagerInf)

Figure 9.3: A Class diagram for the AQA component

querying the ontologies such as SEMINTEC and VICODI, the corresponding experimental results are shown in Tables 6.7 and 6.8 on pages 76 and 78, respectively. SCREECH can be used as either a preprocessor or an approximate reasoner on top of the KAON2 reasoner. As preprocessor defined in Algorithm 8 in Chapter 7 on page 99, one applies a SCREECH-approximation to the ontology to be processed by AQA. The resulting knowledge is approximate with respect to the original ontology. After having been serialized as an OWL ontology using the XML/RDF syntax by the SCREECH component, the compiled ontology can be processed by AQA. As approximate reasoner, implementing the `DLReasoner` interface like other DL reasoners illustrated in Figure 9.3, SCREECH can be invoked in the online variant of AQA.

9.4 Anytime Reasoning Component

In an anytime reasoning system, several reasoning components have to run simultaneously. Although the behavior of a single component can usually be specified and analyzed relatively easily, the behavior of the system as a whole is often too complex to be specified or analyzed thoroughly. This is primarily due (and inherent) to the parallelism between the system's components.

This leads to the development of a common anytime reasoning pattern, which is general enough to implement the kind of anytime reasoning based on the composition of approximate reasoning algorithms or systems. This section presents such a pattern used to implement the anytime reasoning algorithms described in Section 8.1, Chapter 8. The anytime reasoning pattern enables the creation of a composed anytime reasoner, which may involve various approximate reasoners as well as sound and complete DL reasoners running simultaneously. Recall from Section 8.1 that the term anytime reasoning was originally coined to design a reasoning algorithm in such a way that it provides intermediate results computed continuously, and refines them as time increases. In contrast, the idea of developing an anytime reasoning algorithm here relies on a combination of approximate reasoning algorithms, in some cases combined with a DL reasoner. Hence, this kind of anytime reasoning system does not function like a classical anytime reasoning system that is assumed to be interrupted at any given time. Instead, it is interruptible at the completion time of the individual reasoning component involved.

Figure 9.4 shows a UML class diagram of the generic anytime reasoning pattern. This pattern supports an easy creation of anytime reasoning system that shall be based on a combination of multiple reasoning systems.

The main idea of this pattern is, independent from any specific reasoner, to control reasoners using a shared `AnytimeResult` object. Regarding with this, an interface `ComposedReasoner` has been introduced which can be implemented by a specific reasoner. This interface provides methods for reading queries, and performing reasoning task. A specific reasoner should implement the `querying` method for own reasoning task. An `AnytimeReasoningManager` object runs the involved reasoners in

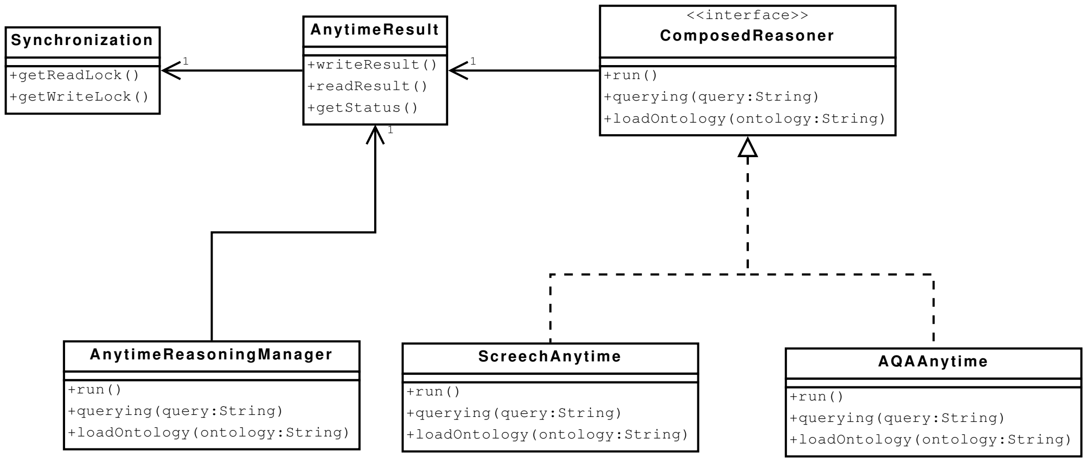


Figure 9.4: A Class diagram for the generic anytime reasoning component

paralell, and forwards the knoweldge base to be queried to each reasoner. The involved reasoners in turn load the knowledge base and wait for queries. Once a query is posed by users, the `AnytimeReasoningManager` object forwards it to the reasoners so they will start performing the reasoning task implemented in the `querying` method. After performing the required reasoning task for the query, each reasoner puts its answer in the shared `AnytimeResult` object using the `writeResult` method. The `AnytimeReasoningManager` object reads the current answer when available, and forwards it to a client's program. The `Synchronization` class is responsible for a proper synchronization between reasoner objects and a `AnytimeReasoningManager` object.

Implementing the `ComposedReasoner` interface for AQA and Screech, two anytime reasoners have been implemented, namely SCREECH-Anytime and AQA-Anytime reasoners, the underlying concepts of these reasoners have been discussed in Section 8.4, Chapter 8. The `ScreechAnyTime` class encapsulates the SCREECH-Anytime algorithm in which the three Screech approximations are composed. The `AQAAnyTime` encapsulates the AQA-Anytime algorithm, which combines an AQA reasoner with an DL reasoner.

9.5 Automated Benchmarking

Providing a foundation of approximate reasoning for the Semantic Web, Chapter 5 presented an abstract framework for evaluating approximate reasoning algorithms. This section discusses the implementation of this framework. In Chapter 5, several functional requirements derived from analyzing the difficulties in benchmarking of reasoning systems were also suggested, which a concrete benchmarking tool has to meet.

In essence, it has been assumed that, due to a very large set of test queries, the tasks of measuring the performance as well as the quality of an approximate reasoning algorithm need to be automated.

The present automated benchmarking tool has been specifically designed to enable full-automated execution for complex benchmarks. Hence, it allows designing a composite benchmark, called benchmark suite, which can be made up of other composite benchmarks, simple benchmarks, or by embedding external tools. In contrast, a (simple) benchmark cannot contain any other simple or composite benchmarks. It contains external tools that will be executed in a sequential manner. In order to provide a flexible design for such hierarchical, composite objects, the Composite pattern has been used. Applying the Composite pattern, the resulting class hierarchy contains classes, such as `AbstractBenchmark`, `BenchmarkSuite`, and `Benchmark` shown in Figure 9.5.

The `AbstractBenchmark` class declares and implements methods that are common for benchmarks, external tools as well as benchmark suites. Due to the Composite pattern, one can view `AbstractBenchmark` objects uniformly and recursively call methods over them. The `exec` method iterates over the collection of `AbstractBenchmark` objects it contains, in a recursive manner, and invokes their `exec` methods. The `setParameter` configures an `AbstractBenchmark` object with required parameters.

The `BenchmarkSuite` class represents a benchmark suite, implementing methods, such as `addBenchmark` and `getBenchmark`, to deal with the components it contains. The `addBenchmark` method is used to add different `Benchmark` objects to a benchmark suite object. The `BenchmarkSuite` stores the other `Benchmark` objects inside a list. The `getBenchmark` is used to retrieve one such object stored in the specified location. In contrast, the `Benchmark` class represents a benchmark and implements methods for loading test queries and ontologies as well as adding and executing external tools. Moreover, the `runTest` method loads an ontology and test queries from a file, runs the embedded reasoners for each query on the given ontology and stores the run times as well as the answers using the `TimeLogger` class. To design this tool extendable and generic as possible for embedding external tools, an abstract class named `ExternalTool` has been provided.

In order to programmatically use the present benchmarking tool, a typical client program would first create a set of `AbstractBenchmark` objects. Using the `addBenchmark` method and adding those to a `BenchmarkSuite` object, a composite benchmark object can be created. When the client program wants to run the composite benchmark, it can simply invoke the `exec` method, which iterates over the collection of `AbstractBenchmark` objects in a recursive manner, and invokes their `exec` methods. An alternative way is to load a composite benchmark from a XML file by using the `loadBenchmark` method implemented in the `BenchmarkFactory` abstract class.

As part of the implementation of this benchmarking tool, several tools have been developed to support the evaluation of approximation reasoning algorithms. These

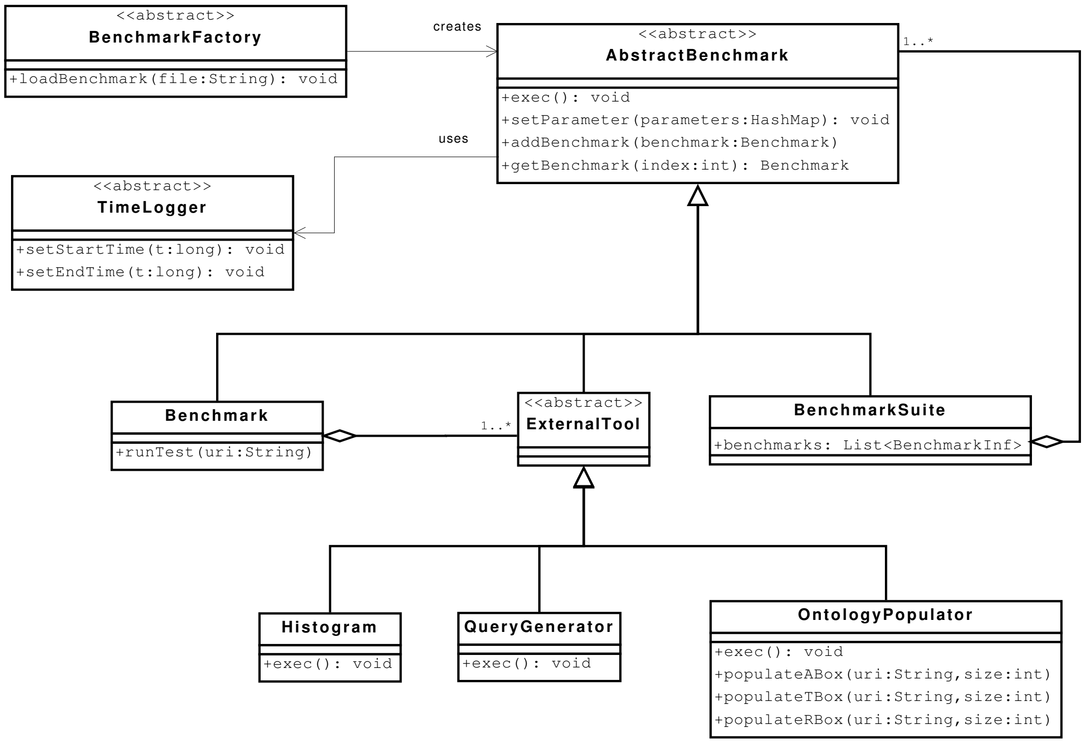

<<abstract>>
BenchmarkFactory
+loadBenchmark(file:String): void
creates
<<abstract>>
AbstractBenchmark
+exec(): void
+setParameter(parameters:HashMap): void
+addBenchmark(benchmark:Benchmark)
+getBenchmark(index:int): Benchmark
1..*
uses
<<abstract>>
TimeLogger
+setStartTime(t:long): void
+setEndTime(t:long): void
Benchmark
+runTest(uri:String)
1..*
<<abstract>>
ExternalTool
BenchmarkSuite
+benchmarks: List<BenchmarkInf>
Histogram
+exec(): void
QueryGenerator
+exec(): void
OntologyPopulator
+exec(): void
+populateABox(uri:String,size:int)
+populateTBox(uri:String,size:int)
+populateRBox(uri:String,size:int)

Figure 9.5: A Class diagram for the automated benchmarking tool

include tools for generating query as well as populating ontologies regarding their ABox and TBox.

9.5.1 Query Generation

The evaluation of the approximate reasoning systems developed in this thesis has required various kinds of test queries. The `QueryGenerator` class provides several algorithms for generating complex concept expressions, which are needed for the evaluation. Several criteria had to be met to generate such queries. In the following, we will first define two basic sets of test queries and then describe the corresponding algorithms that generate them. For the sake of compactness, we will make use of the notation of primitive function.

The simplest query generation algorithm is to generate concepts of depth 2 from a given set Nc of (atomic) concepts using concept connectives such as $\sqcup$ and $\sqcap$. There are two requirements for the definition of this query generation function: (1) to avoid generating concepts, such as $A \sqcup A$ for a concept A, and (2) to generate only one combination $A \sqcup B$ from concepts A and B, while the alternative combination $B \sqcup A$ is apparently without any sense. The resulting algorithm is defined as follows:

$$genConceptQuery(op, Nc) = \begin{cases} genConcept(op, c_1, Nc) \\ \cup\, genConceptQuery(op, Nc \setminus \{c_1\}) \ \textbf{if } Nc \neq \varnothing \\ \varnothing \ \textbf{otherwise} \end{cases}$$

, where $op \in \{\sqcup, \sqcap\}$ and Nc is a set of concepts. The function *genConcept* is defined to be

$$genConcept(op, c_1, Nc) = \begin{cases} \{(c_1 \ op \ c_2)\} \\ \cup\, genConcept(op, c_1, Nc \setminus \{c_2\}) \ \textbf{if } Nc \neq \varnothing \\ \varnothing \ \textbf{otherwise.} \end{cases}$$

A further basic query generation function is to generate role concepts from a given set R of roles and a set Nc of concepts that are disjoint. This function is defined as follows:

$$genRoleQuery(\circ, R, Nc) = \begin{cases} genRoleConcept(\circ, r, Nc) \\ \cup\, genRoleQuery(\circ, R \setminus \{r\}, Nc) \ \textbf{if } R \neq \varnothing \\ \varnothing \ \textbf{otherwise} \end{cases}$$

, where $\circ \in \{\forall, \exists\}$, Nc is a set of concepts and Nc is a set of roles. The function *genRoleConcept* is defined to be

$$genRoleConcept(\circ, r, Nc) = \begin{cases} \{(\circ \ r \ c)\} \cup genRoleConcept(\circ, r, Nc \setminus \{c\}) \\ \qquad\qquad\qquad\qquad \textbf{if } Nc \neq \varnothing \\ \varnothing \ \textbf{otherwise.} \end{cases}$$

Up to this point, we have considered the generation of basic queries that are built from using the DL concept connectives $\sqcup$ and $\sqcap$ as well as concepts with roles. Particularly when dealing with large knowledge bases, such basic queries as considered above could be even larger, so it is impracticable to test all the queries. In the following, we introduce several query generation algorithms that can be used to restrict those basic queries in some way.

Sound selection. Especially for testing sound reasoning algorithms, it is adequate to take into account only those queries, for which a sound and complete DL reasoner provides answers, *i.e.,* a non-empty set. Hence, for any set Q of basic queries, the following function determines a subset of queries, for which a sound and complete DL reasoner provides a non-empty set:

$$soundQuery(KB, Q) = \begin{cases} soundQuery(KB, Q \setminus \{q\}) \\ \qquad\qquad\qquad \textbf{if } Res_{dl}(KB, q) = \emptyset \wedge Q \neq \emptyset \\ \{q\} \cup soundQuery(KB, Res_{dl}, Q \setminus \{q\}) \\ \qquad\qquad\qquad \textbf{if } Res_{dl}(KB, q) \neq \emptyset \wedge Q \neq \emptyset \\ \emptyset \ \textbf{otherwise}. \end{cases}$$

, where *KB* is a knowledge base and Res_{dl} is a DL reasoner.

Random selection. Another way of restricting basic queries is to randomly select some queries. Assuming a random function *rand*, the following function *randomSel* generates the set of randomly selected queries of a certain size *n* from a given set:

$$randomSel(Q) = \begin{cases} \{q\} \cup randomSel(Q \setminus \{q\}) \ \textbf{if } Q \neq \emptyset, \text{where } q = rand(Q) \\ \emptyset \ \textbf{otherwise}. \end{cases}$$

Timed selection. In some cases, it is useful to avoid testing certain queries, for which a sound and complete DL reasoner under test needs much time. Assuming a function *time* for measuring run time, the following function *timedQuery* generates certain queries from a given set of queries, to which a sound and complete DL reasoner responds in a specified time *t*:

$$timedQuery(KB, Res_{dl}, Q, t) = \begin{cases} \emptyset \ \textbf{if } Q = \emptyset \\ \{q\} \cup timedQuery(KB, Res_{dl}, Q \setminus \{q\}, t) \\ \qquad\qquad\qquad \textbf{if } time(Res_{dl}(KB, q) \leq t) \\ timedQuery(KB, Res_{dl}, Q \setminus \{q\}, t) \\ \qquad\qquad\qquad\qquad \textbf{otherwise}. \end{cases}$$

In the previous sections, we have discussed various kinds of test queries that were utilized for the evaluation of the approximate instance retrieval methods developed in this thesis.

9.5.2 Ontology Population

In addition to defining test queries, an important aspect of testing approximate reasoning algorithms with regard to performance is to construct artificial ontologies that can be used for exploring the strengths and weaknesses of an approximate reasoning algorithm. For this purpose, we have devised several algorithms that can be used to populate OWL-DL ontologies.

Table 9.2: A population algorithm for $\mathcal{SHOIN}$: KB is a $\mathcal{SHOIN}$ knowledge base. $KB_{\mathcal{T}}$, $KB_{\mathcal{R}}$ and $KB_{\mathcal{A}}$ are the respective TBox, RBox and ABox in KB. α stands for axioms in KB

	Populating $\mathcal{SHOIN}$ KB	
$pop(\alpha, K)$	$=$	$\bigcup_{0<i<K}\{rn(\alpha, i)\} \cup \{\ \alpha\}$
$pop(KB_{\mathcal{R}}, K)$	$=$	$\bigcup_{\alpha \in KB_{\mathcal{R}}} pop(\alpha, K)$
$pop(KB_{\mathcal{T}}, K)$	$=$	$\bigcup_{\alpha \in KB_{\mathcal{T}}} pop(\alpha, K)$
$pop(KB_{\mathcal{A}}, K)$	$=$	$\bigcup_{\alpha \in KB_{\mathcal{A}}} pop(\alpha, K)$
$pop(KB, K)$	$=$	$pop(KB_{\mathcal{R}}, K) \bigcup pop(KB_{\mathcal{T}}, K) \bigcup pop(KB_{\mathcal{A}})$

Note that populating OWL ontologies regarding the TBox as well as the ABox is not the primary focus of this thesis, so an in-depth treatment of this topic goes beyond its scope. However, in order to compare and contrast the performance of the approximate reasoning systems presented in this thesis with other sound and complete DL reasoners, several algorithms have had to be developed. These algorithms are encapsulated in the `OntologyPopulator` class, as illustrated in Figure 9.5. The class provides three `populateTBox`, `populateRBox` and `populateABox` methods for populating TBox, RBox and TBox in a knowledge base, taking as arguments the URI of the ontology to be populated as well as its population size regarding how many copies should be generated. The corresponding algorithms are presented as a recursive function *pop* in Table 9.2.

The function *pop* takes two variables, K and KB, as arguments, with K referring to the number of copies for populating the knowledge base KB. Starting with K, the general idea of this population function is to recursively traverse the axioms in $KB_{\mathcal{T}}$, $KB_{\mathcal{R}}$ and $KB_{\mathcal{A}}$ as well as all concept expressions until atomic concepts and roles are reached. Moreover, the function *pop* is defined by being applied to TBox, RBox, and ABox, respectively. It is defined by being applied to each of their axioms. Subsequently, K copies of each axiom are generated. All the copies are renamed using the function *rn* according to the current value of K, as given in the first row of Table 9.2.

In Table 9.3, the function *rn* is recursively defined for TBox, RBox and ABox axioms. For RBox and ABox axioms, it changes atomic concept, role, and individual names by concatenating the current value of K to them. For TBox axioms, *rn* is recursively applied to axioms of $\sqsubseteq$ and $\equiv$ as well as complex concept expressions.

9.5.3 Histogram Generation

To work with a large set of test queries, and to illustrate their corresponding results graphically, an illustration of displaying all results becomes easily unclear. In this case, advanced methods of data summary, such as histogram, is practicable. A histogram is a summary graph showing distribution of data points measured that falls

Table 9.3: A renaming algorithm for $\mathcal{SHOIN}$: $\circ$ is a generic concatenate function. a_1 and a_2 are individuals. C and D stand for complex concepts while r, r_1 and r_2 stand for roles. K stands for a natural number. The formal syntax of $\mathcal{SHOIN}$ is discussed in Chapter 3

Renaming Concepts		
$rn(\top, K)$	$=$	$\top$
$rn(\bot, K)$	$=$	$\bot$
$rn(A, K)$	$=$	$A \circ K$
$rn(\neg C, K)$	$=$	$\neg rn(C, K)$
$rn(C \sqcap D, K)$	$=$	$rn(C, K) \sqcap rn(D, K)$
$rn(C \sqcup D, K)$	$=$	$rn(C, K) \sqcup rn(D, K)$
$rn(\forall\, r\,.C, K)$	$=$	$\forall\, r \circ K\,.rn(C, K)$
$rn(\exists\, r\,.C, K)$	$=$	$\exists\, r \circ K\,.rn(C, K)$
$rn((\geq n\, r.), K)$	$=$	$\geq n\, r \circ K.$
$rn((\leq n\, r.), K)$	$=$	$\leq n\, r \circ K.$
Renaming TBox, RBox, and ABox Axioms		
$rn(C \sqsubseteq D, K)$	$=$	$rn(C, K) \sqsubseteq rn(D, K)$
$rn(C \equiv D, K)$	$=$	$rn(C, K) \equiv rn(D, K)$
$rn(r_1 \sqsubseteq r_2, K)$	$=$	$r_1 \circ K \sqsubseteq r_2 \circ K$
$rn(r_1 \equiv r_2, K)$	$=$	$r_1 \circ K \equiv r_2 \circ K$
$rn(\mathsf{Trans}(r), K)$	$=$	$\mathsf{Trans}(r \circ K)$
$rn(C(a), K)$	$=$	$C \circ K(a \circ K)$
$rn(R(a_1, a_2), K)$	$=$	$R \circ K(a \circ K, b \circ K)$
$rn(a_1 \approx a_2, K)$	$=$	$a_1 \circ K \approx a_2 \circ K$
$rn(a_1 \not\approx a_2, K)$	$=$	$a_1 \circ K \not\approx a_2 \circ K$

within various class-intervals. A class interval (also called bin) is a division of a range of values into sets of non-overlapping intervals for plotting a histogram.

In our work, histograms are used to compare and contrast performances between approximate reasoning systems und DL reasoning systems. They illustrate the distribution of performance gains for test queries. The `Histogram` class provides functionalities for generating histograms for analyzing performances. A `Histogram` object reads its configurations from the `parameters` object and experimental data, performs statistic calculations on them, such as frequency distribution and cumulative percentage, and yields a series of gnuplot[4] scripts. The generation of scripts can be customized by using an predefined template. It also enables the automatic detection of the appropriate number of bins in experimental data. Sometimes, the shape of the histogram is particularly sensitive to the number of bins. If the bins are too wide, important

[4] It is a popular comprehensive tool for plotting data in various forms.

information might get omitted. For example, the data may be bimodal, but this characteristic may not be evident if the bins are too wide. On the other hand, if the bins are too narrow, what may appear to be meaningful information really may be due to random variations that appear because of the small number of data points in a bin. To determine whether the bin width is set to an appropriate size, different bin widths should be used and the results ought to be compared to determine the sensitivity of the histogram shape with respect to bin size. Generating meaningful histograms aims at the automatic detection of the number of bins, analyzing the distribution of results. It can be also configured with a certain number of bins.

9.6 Conclusion

This chapter has described the implementation details of the approximate reasoning algorithms having been presented in this thesis. The resulting system, AORW, includes the individual components for approximate instance retrieval for OWL ontologies as well as the automated benchmarking tool.

To provide an efficient implementation for this system, a careful design has been conducted and a number of implementation tests have been carried out to ensure the correctness of the implementation. Such a careful development was required, since a bad design and its incorrect implementation could affect the performance of reasoning as well as the quality of approximation.

Applying the Bridge pattern and Visitor pattern, a comprehensive design for AQA has been realized. This design allows a flexible implementation so that AQA can switch between its variants at runtime without creating any new reasoner object. It furthermore makes testing and debugging easier, which is very important for the development of an approximate reasoning algorithm. The mainteance of the materialization for ABox assertions and accessing them as well as invoking DL reasoners is completely separated from computing approximate extensions.

SCREECH has been implemented by applying the strategy pattern due to its different knowledge compilation strategies. A functional requirement for the development of SCREECH has been to design it such that it can be used for two purposes: as a knowledge compiler or as an approximate reasoner. This has been achieved by separating the functionalities required by a SCREECH approximation from the functionalities required from an approximate reasoner for instance retrieval. The Strategy pattern allows to access different SCREECH approximation strategies in a seamless manner and enables an easy extension when implementing other new approximation strategies.

Applying the Strategy pattern, the benchmarking tool has been developed, since there was no special tool available to automate benchmarking of approximate reasoning systems. The resulting tool is generic for automated benchmarking, which supports the comparison of system performance as well as measuring the degree of soundness and completeness for over several thousands of complex concept queries.

Its generic architecture allows integrating external tools under a common interface. The tool takes care of the time-consuming details of the execution, management of experimental results, and their visual interpretation. Script-based editing of benchmarks and the ability to integrate external tools makes the benchmarking tool inherently extendable and allows it to run more complex benchmarks fully automatically.

Chapter 10

Applications

This chapter presents how the approximate reasoning solutions developed in this work and presented in the previous chapters have been used in a concrete setting given by the research project we have been involved in. First, the research project THESEUS will be introduced in Section 10.1. Section 10.2 then presents the main concept of the reasoning broker having been developed in THESEUS, whereas Section 10.3 describes its implementation, and the integration of the anytime reasoners, which have been presented in Chapter 8, into the reasoning broker. Finally, in Section 10.4, we will discuss possible applications of approximate reasoning.

10.1 The THESEUS Project

Currently, Semantic Web technologies are maturing and moving out from academic applications into the industrial sector. This is demonstrated by the strong and growing interest of various business sectors like human resources and employment, health care and life sciences, transport and logistics, and on the other hand, public institutions like the European Commission, which has supported the development and transfer of these technologies into the business world in various European projects like Knowledge Web[1], NEON[2] and LarKC[3]. A common goal of these projects is to develop practicable, scalable ontology reasoning systems for emerging Semantic Web applications of expressive ontologies based on various reasoning paradigms and technologies. Apart from these European projects, other national projects, like the German project THESEUS,[4] aim at bridging the gap of pure research and practical application of semantic technologies.

The focus of THESEUS lies on semantic technologies for the next generation Internet, which will recognize the meaning and content of information and will be able

[1] `http://knowledgeweb.semanticweb.org` [accessed 2009-9-12]

[2] `http://www.neon-project.org` [accessed 2009-9-12]

[3] `http://www.larkc.eu/` [accessed 2009-9-12]

[4] Supported by the German Federal Ministry of Economics and Technology.

to classify it – irrespective of whether it be words, photos, sounds, 2D or 3D image data. With these technologies, computer programs will be able to intelligently understand in which context data should be stored as well as draw logical conclusions and establish correlations.

Reasoning Objectives in the THESEUS Project

The THESEUS project consists of several cases and a cross cutting Content Technology Cluster (CTC). CTC aims at solving the most fundamental scientific and technological challenges in the THESEUS project. Moreover, the core working group within CTC, which we have been involved in, focuses on the development of an ontology management system which is supposed to lay the basis for advanced Semantic Web technologies to be developed in the THESEUS Use Cases. Having analyzed the functional and non-functional requirements posed by the THESEUS Use Cases, it has been identified that the scalable ontology reasoning system will be a key component in the expected ontology management system. In the following, we will present some of the identified objectives, which are directly related to the work presented in this thesis.

Size and scale of semantic data: The use of ontologies and semantic data can vary in size and scale regarding the amount of accessed and processed data. In some cases, the knowledge involved in an application can be rather restricted in terms of quantity providing only supplementary support to use case solutions. In other cases, both ontologies and semantic data pools can be very large in size. A (typical) special case occurs when manually created ontologies have a maintainable size, while semantic (meta) data formulated with respect to these ontologies comes in great amounts.

Expressivity of ontology language: Ontologies can vary in their expressivity. While simple (light-weight) ontologies may include subsumption hierarchies of concepts or properties, higher axiomatized (heavy-weight) ontologies include the use of Boolean concept expressions, existential and universal property restrictions, cardinality restrictions, etc. Use Cases may need ontologies of different expressivity, depending on the ontology-based functionality they exploit. This again depends on the modeling capabilities of users and support by provided tools, *e.g.,* for providing semantic annotation, as well as the added value expected from exploiting semantic features by the system, *e.g.,* in assisting semantic search or browsing, consistency checking, or explanation.

10.2 Reasoning Brokerage

This section presents a reasoning broker system, which aims at providing scalable ontology reasoning services for the advanced Semantic Web applications being developed in the THESEUS Use Cases.

The main idea behind the reasoning broker system HERAKLES is to provide a unifying platform for various specific reasoners, and to act as an abstract reasoning service that hides the concrete type of reasoning used for a particular request from

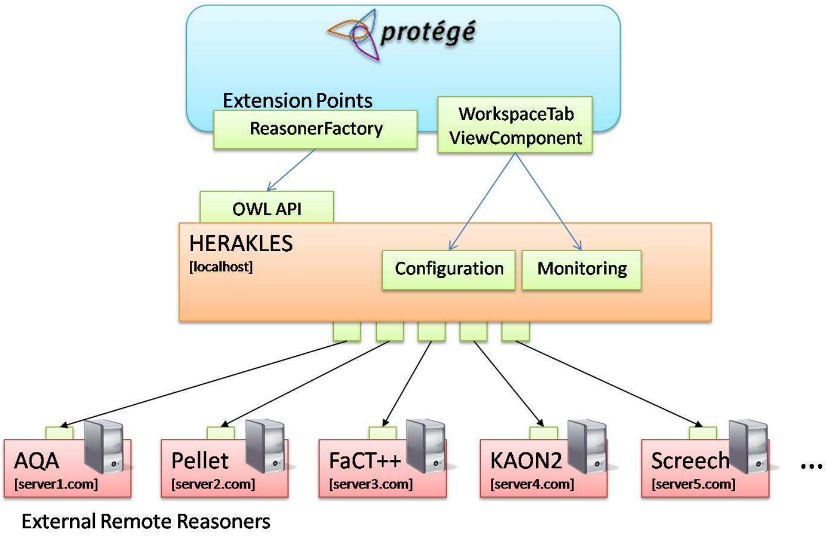


Figure 10.1: Overview of the HERAKLES plug-in

the client. Behind this facade, different types of reasoners serve as components for answering the request next to components for supplementary tasks like normalisation or modularisation of ontologies and strategies for caching and other forms of optimisation on top of these. In the best case, a specific reasoning task might be split in several parts that are independently evaluated by different reasoners in parallel according to the nature of the query and ontology modules involved. HERAKLES implements the reasoner interface provided by the OWL API and can thus be used as an ordinary OWL reasoner by any semantic application. However, behind the scenes HERAKLES manages several *external remote reasoners*, which can again be any reasoner that is accessible via the OWL API's reasoner interface. Currently there are connections to the standard reasoners Pellet[5], FaCT++[6], and HermiT[7], as well as to KAON2[8] via a newly developed adapter bridging the KAON2 API and the OWL API.

Figure 10.1 illustrates the general architecture of HERAKLES and its integration with Protégé. Apart from using HERAKLES as a reasoner via the OWL API, it can also be monitored and configured via tabs and views added to Protégé. The external remote reasoners are connected to HERAKLES via a remote interface, and can hence be located on different servers. The centralised control of various external remote reasoners enables the implementation and combination of different features that provide added value compared to traditional reasoner systems. The following list of features can be implemented by the use of a strategy concept, which controls the behaviour of the reasoning broker system. However, not all of those features have been imple-

[5] http://clarkparsia.com/pellet

[6] http://owl.man.ac.uk/factplusplus/

[7] http://www.hermit-reasoner.com/

[8] http://kaon2.semanticweb.org/

mented yet. Concrete strategies that are in fact implemented are mentioned at the end of this section.

Parallel Reasoner Invocation. Predicting the exact run-time of reasoning systems for a given ontology and a given query is hardly possible in practise. Even prediction of the first reasoner to finish a given query is not possible in many cases, which motivates the parallel execution of reasoning tasks on a set of reasoning systems. The reasoner which finishes first can then propagate the results of the query, assuming correctness of all reasoners invoked.

Reasoner Selection. Due to benchmarks and knowledge about the implementations of different reasoning systems, some of the available reasoners can be selected prior to actually executing the reasoning tasks. This selection will keep reasoners unsuitable for a given ontology/query combination idle and available for other reasoning tasks they are more appropriate for.

Query Decomposition. Queries containing complex class or property expressions can be analysed if those expressions decompose into several subexpressions which can be answered by different reasoners in parallel. It must be ensured, though, that the combination of the results delivered by different reasoners does not bear an unacceptable overhead compared to the performance gained by parallel computation. A naive example of such a decomposition is to split a conjunctive class expression into its operands and answer each subexpression by different reasoners in parallel. The answer in this simple case would be the intersection of the answers delivered by the different reasoners.

Partitioning of Ontologies. The notions of *conservative extensions* and *locality* provide means to partition an ontology into several semantically independent modules [GHKS07]. Executing a query on such a module instead of the whole ontology can result in run-time performance improvements for reasoning requests. Moreover, in combination with intelligent query decomposition more complex queries can be executed by different reasoners on different ontology modules in parallel.

Load Balancing and Scheduling. In a scenario where a sequence of queries is to be answered, or multiple applications are using the same instantiation of a reasoning broker, it is necessary to balance the workload of each remote reasoner in order to provide optimal overall run-time performance. To this end an asynchronous reasoner interface would allow for acceptance of more than a single query at once from an application. Query answering can then be scheduled to be processed by the different remote reasoners according to their strengths and language conformance, in order to ensure maximum throughput of queries.

Real-time Benchmarking. Assuming that the reasoning systems have been correctly implemented, existing benchmarks for ontology reasoning basically focus on performance measurements. It is also important to perform correctness tests for the implementation of the reasoning systems [GHT06, GTH06], in particular, the evaluation of emerging approximate reasoning systems requies measuring the quality of answers which is a special case of correctness tests. The reasoning broker provides an ideal infrastructure for both performance and correctness tests.

10.3 Implementation Of Reasoning Brokerage

The previous section described the underlying concept of the reasoning broker system. This section presents its current implementation.

The reasoning broker has been implemented as the HERAKLES system[9] in the Java™ programming language in accordance with the OWL API [HBN07]. HERAKLES is implemented in a client/server architecture to ensure modular decoupling of remote reasoners, *i.e.,* the HERAKLES server, and the broker layer, *i.e.,* the HERAKLES client. The HERAKLES client implements the `OWLReasoner` interface of the OWL API and can thus be used like any standard reasoner from within an OWL API based application. The HERAKLES client furthermore maintains a reasoner registry to record attached remote reasoners that can be used by the broker. Remote reasoners are wrapped into a remote reasoner adapter, which allows them to be run as reasoning servers connected to the HERAKLES client. This adaptation has not only been realised for OWL API compliant reasoners, but also for the KAON2 reasoner with its own API, and the KAON2 based approximate reasoning systems SCREECH and AQA. The communication between client and servers has been realised using Java™ RMI[10]. Hence reasoning servers can be run on remote machines, which allows for exclusive provision of computational resources for each reasoner.

10.3.1 Broker Strategies

The behaviour of the broker and thus the implementation of the features discussed in Section 10.2 is controlled by exchangeable broker *strategies*. More precisely there is a load strategy to control the loading of ontologies into the different remote reasoners, and an execution strategy to control the execution of reasoning tasks by those reasoners. The strategy concept allows for easy substitution of both load and execution strategy by different implementations depending on the usage scenario of the reasoning broker. Furthermore the strategy concept allows for the implementation and use of customised strategies for specific use cases. Implementation of strategies in HERAKLES is simplified by several *strategy components*, which encapsulate core broker tasks such as parallelisation, reasoner selection, partitioning, or ontology analysis.

[9] `http://herakles.sourceforge.net` [accessed 2009-9-12]

[10] Remote Method Invocation.

These strategy components can then be used and combined to assemble new broker strategies. The following paragraphs describe interfaces and currently available implementations of strategy components and strategies for HERAKLES.

Paralleliser. This component invokes the execution of a reasoning task on a selection of reasoners in parallel. It will most likely be the final component involved in a strategy, possibly after some partitioning and selection steps. Currently there are two implementations: a competing paralleliser, which delivers the result of the reasoner that finishes first, and a blocking paralleliser, which waits until all reasoners have finished. The former will most likely be the default implementation in order to gain the best run-time performance of the broker, while the latter could be used for benchmarking tasks.

Selector. This component selects a set of reasoners out of the ones registered by the broker. Different implementations can apply different selection criteria, such as ontology properties, reasoning task to be executed, or query properties[11]. Currently there are two implementations: an ontology selector, which selects reasoners according to properties of the ontology, and a task selector, which selects reasoners according to the reasoning task to be performed. Selection of reasoners in this way requires knowledge about the capabilities of the different reasoners, which are currently recorded by the remote reasoner adapters.

Modulariser. This component provides means to partition an ontology into several modules, which can ideally be processed by different reasoners concurrently. There is currently no implementation available, but there are plans for realising partitioning as discussed in Section 10.2.

Analyser. This component is supposed to be used in load strategies which perform an analysis of the ontologies to be loaded. The information gained by this analysis can then be used *e.g.*, by selectors in the execution phase.

Basic/Analysing Load Strategy. This load strategy loads the ontology into all available remote reasoners. The *analysing load strategy* extends the basic load strategy by additionally analysing the ontologies and recording their properties.

Basic/Fault-tolerant Parallelisation Strategy. This execution strategy performs a reasoning request on all available (idling) remote reasoners, which have loaded the ontologies. A *fault-tolerant parallelisation strategy* extends the basic parallelisation strategy by being insensitive to failing remote reasoners. In case of a failure, it waits

[11] Query properties could be for instance the language features used in a class description in an instance query.

for more reasoners to become available and fails on a particular query only if all remote reasoners fail.

(Fault-tolerant) Task Selection Strategy. This execution strategy selects remote reasoners according to the reasoning task requested. Selection is carried out by a *task selector* strategy component, which can be configured in order to map reasoning tasks to reasoners having certain characteristics. Selected reasoners are then invoked in parallel using the *competing paralleliser* strategy component. A *fault-tolerant task selection strategy* extends the task selection strategy by being insensitive to failing remote reasoners. In the case all selected reasoners fail on a particular query, it also selects reasoners not matching the selection criteria in order to try and have the query succeed[12].

Anytime Strategy. This execution strategy simulates anytime reasoning behaviour by using approximate reasoning systems developed in this thesis. It selects distinct sets of remote reasoners respecting soundness, completeness, and both soundness and completeness. All reasoners are invoked in parallel using the *competing paralleliser* strategy component, where each set of reasoners is invoked by a different paralleliser. Results are delivered from the fastest reasoner of each set, characterising results as *sound*, *complete*, or *sound/complete* rsp. Anytime behaviour arises from the faster runtimes of the approximate reasoners and thus from the early delivery of (potentially) unsound or incomplete answers.

Benchmark Strategy. This execution strategy can be used for simple run-time performance benchmarking of reasoners. It invokes all available remote reasoners in parallel without any prior selection. The strategy component for parallel execution is the *blocking paralleliser* to enable time measurement of each reasoner for each reasoning task. The blocking characteristic of this strategy component ensures availability of all reasoners for each reasoning task out of a test series.

10.3.2 User interface

Access to the HERAKLES system via the OWL API reasoner interface is primarily useful when it is supposed to serve as a reasoner in an application that needs support for ontology reasoning. Another aspect is to integrate an ontology reasoner in an ontology editor to help knowledge engineers detect erroneous modeling. Since there are already a number of sophisticated ontology-editing tools available, our aim is to incorporate our developed reasoning solutions into an existing ontology editing tool.

Among others, Protégé[13] is a popular, open-source ontology engineering environment with a large community. It provides functionality for editing and reasoning

[12] This behaviour assumes that the selection in the first place was only based on expected run-time performance and not due to language conformance.

[13] `http://protege.stanford.edu/` [accessed 2009-9-9]

OWL ontologies, and is highly extensible and customizable. Originally, it is a frame-based knowledge model with support for metaclasses. The latest version, namely Protégé 4.0 is now built on top of the OWL API, providing more flexible development and a straightforward migration path for OWL-based applications. Hence, our decision is to use Protégé as a general purpose ontology engineering tool as well as a general reasoning frontend.

In what follows, we will describe it briefly as far as we need for our purpose, in particular, for the use of our approximate reasoning methods. Protégé's user interface consists of several screens, called tabs, each of which displays a different aspect of the ontology in a specialized view. Each of the tabs can include arbitrary Java components. Most of the existing tabs provide an explorerstyle view of the model, with a tree on the left hand side and details of the selected node on the right hand side. The details of the selected object are typically displayed by means of forms. The forms consist of configurable components, called widgets. Typically, each widget displays one property of the selected object. There are standard widgets for the most common property types, but ontology developers are free to replace the default widget Protégé's architecture makes it possible to add and activate plugins dynamically, so that the default system's appearance and behavior can be completely adapted to a project's needs.

Protégé is a flexible, configurable platform for the development of arbitrary model-driven applications and components. Protégé has an open architecture that allows programmers to integrate plug-ins, which can appear as separate tabs, specific user interface components, or perform any other task on the current model. The Protégé-OWL editor provides many editing and browsing facilities for OWL models, and therefore can serve as an attractive starting point for rapid application development.

Writing class expression

Our anytime solutions require class expressions. The syntax of the OWL specification for modeling ontologies is clearly not convenient for writing OWL class expressions for query answering. The OWL Abstract Syntax[14] is much more user-friendly, however, rather verbose. Hence, a new Anytime Query plug-in has been implemented based on the DL Query plugin, which allows writing class expressions in the Manchester OWL Syntax [HDG$^+$06].

This syntax is a new syntax which borrows ideas from the OWL Abstract Syntax and the DL style syntax. Special logical symbols such as $\exists$, $\forall$, $\neg$ and have been replaced by keywords such as "some", "only", and "not". Furthermore, the Manchester OWL Syntax uses an infix notation rather than a prefix notation for keywords. All these features make the syntax quicker to write and easier to read and understand even for non-logicians. An overview of the Manchester OWL Syntax for using the

[14] `http://www.w3.org/TR/owl-semantics/` [accessed 2009-9-9]

Table 10.1: The Manchester OWL Syntax for writing class expression

OWL	**DL Symbols**	**Syntax Keyword**	**Example**
someValuesFrom	$\exists$	**some**	`hasChild` **some** `Man`
allValuesFrom	$\forall$	**only**	hasSibling **only** Woman
hasValue	$\ni$	**value**	`hasCountryOfOrigin` **value** Mongolia
minCardinality	$\geq$	**min**	hasChild **min** 3
cardinality	$=$	**exactly**	hasChild **exactly** 3
maxCardinality	$\leq$	**max**	hasChild **max** 3
intersectionOf	$\sqcap$	**and**	Doctor **and** Female
unionOf	$\sqcup$	**or**	Man **or** Woman
complementOf	$\neg$	**not**	**not** Child

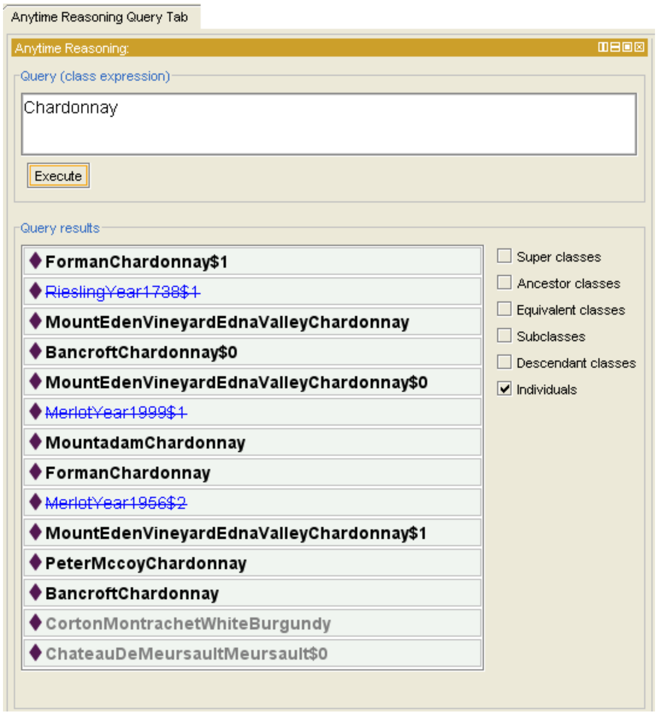
Anytime Query plug-in is given in Table 10.1. Readers interested in the full description of the syntax might refer to [HDG$^+$06].

Displaying Anytime Results

A Reasoning Result plug-in provided in Protégé is used to display the results returned by a DL reasoner. Yet, the results obtained by an approximate reasoning system require a different interpretation compared to those obtained by a DL reasoner. Hence, a new Anytime Result plug-in has been implemented by extending the Reasoning Result plug-in. In order to differentiate the results with respect to the reasoners involved in an anytime reasoner, the Anytime Result plug-in allows displaying the results in different colors.

Recall from Section 8.4.1, Chapter 8 that the SCREECH-Anytime reasoner consists of three reasoners, namely two instantiations of SCREECH-NONE and one instantiation of SCREECH-ALL. It has been designed that one instantiation of SCREECH-NONE is executed on the negation of a given query in order to determine possibly incorrect individuals. The SCREECH-Anytime reasoner informs the Anytime Result plug-in which of the three reasoners provides its result. The Anytime Result plug-in then displays the obtained result in different colors depending on the soundness and completeness characteristics of the reasoner that provided it.

This means, when the instantiation of SCREECH-ALL provides the result at first, the Anytime Result plug-in displays it in gray, since it might contain some possibly incorrect individuals, that cannot be determined regarding which specific one is correct or incorrect at that time. Consequently, when the first instantiation of SCREECH-NONE provides the result, due to its soundness, it is clear that the result contains only correct individuals, but may contain not all of the correct ones. In this case, the Anytime Result plug-in displays results of the first instantiation of SCREECH-NONE



Query (class expression)

Chardonnay

Query results

- ♦ FormanChardonnay$1
- ♦ ~~RieslingYear1738$1~~
- ♦ MountEdenVineyardEdnaValleyChardonnay
- ♦ BancroftChardonnay$0
- ♦ MountEdenVineyardEdnaValleyChardonnay$0
- ♦ ~~MerlotYear1999$1~~
- ♦ MountadamChardonnay
- ♦ FormanChardonnay
- ♦ ~~MerlotYear1956$2~~
- ♦ MountEdenVineyardEdnaValleyChardonnay$1
- ♦ PeterMccoyChardonnay
- ♦ BancroftChardonnay
- ♦ CortonMontrachetWhiteBurgundy
- ♦ ChateauDeMeursaultMeursault$0

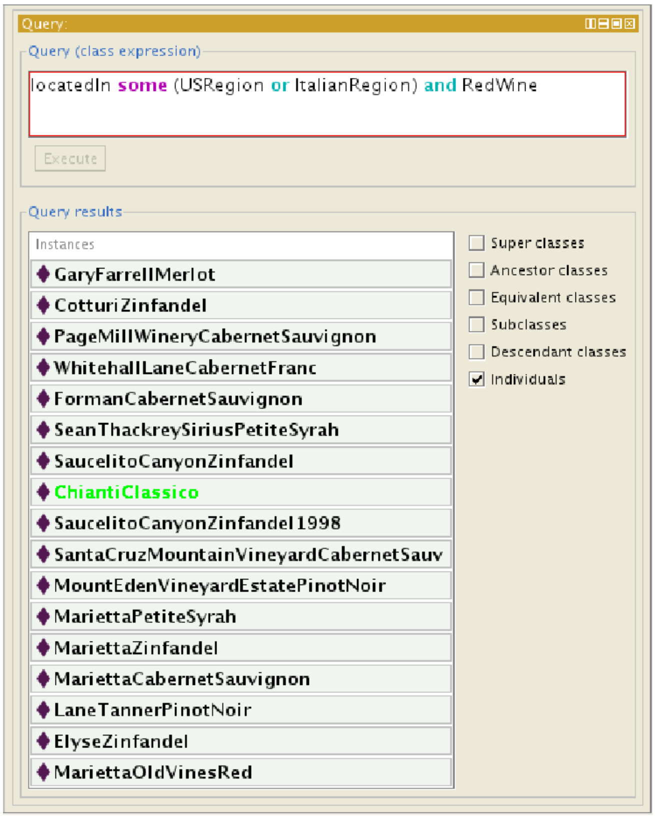
□ Super classes
□ Ancestor classes
□ Equivalent classes
□ Subclasses
□ Descendant classes
☑ Individuals

Figure 10.2: The result for the class named *Chardonnay* obtained by the SCREECH-Anytime reasoner

in black indicating that they are definitely correct ones. Individuals obtained from the second instantiation of SCREECH-NONE are crossed out, since they are definitely incorrect individuals.

Recalling from Section 8.4.1, Chapter 8 that soundness and completeness of AQA algorithms depend on the type of queries, AQA algorithms are sound for those queries, in which no universal quantifier occurs. The Anytime Result plug-in displays the correct individuals obtained from AQA algorithms in black, whereas those obtained from a DL reasoner are displayed in green so that one can examine which ones have been obtained either from a DL reasoner or an AQA reasoner.

Consider, for instance, a complex concept expression $\text{REDWINE} \sqcap \exists \mathsf{locatedIn}.(\text{ITALIANREGION} \sqcup \text{USREGION})$ to query all red wines, which are produced in the USA or Italy. Figure 10.3 illustrates the final results obtained from the AQA-Anytime reasoner, displaying the individuals obtained by the AQA reasoner in black and those from a DL reasoner in green. In the case of queries with the universal quantifier, the AQA algorithms are unsound so the incorrect individuals obtained from an AQA reasoner are crossed out, as it is the case for SCREECH-Anytime.



locatedIn some (USRegion or ItalianRegion) and RedWine

Instances

- GaryFarrellMerlot
- CotturiZinfandel
- PageMillWineryCabernetSauvignon
- WhitehallLaneCabernetFranc
- FormanCabernetSauvignon
- SeanThackreySiriusPetiteSyrah
- SaucelitoCanyonZinfandel
- ChiantiClassico
- SaucelitoCanyonZinfandel1998
- SantaCruzMountainVineyardCabernetSauv
- MountEdenVineyardEstatePinotNoir
- MariettaPetiteSyrah
- MariettaZinfandel
- MariettaCabernetSauvignon
- LaneTannerPinotNoir
- ElyseZinfandel
- MariettaOldVinesRed

☐ Super classes
☐ Ancestor classes
☐ Equivalent classes
☐ Subclasses
☐ Descendant classes
☑ Individuals

Figure 10.3: The result for the complex query obtained by the AQA-Anytime reasoner

10.3.3 Integration of Anytime Reasoners

In the previous sections, we have introduced the underlying concept of HERAKLES system and the integration of the anytime reasoners developed in this work. This section describes how these anytime reasoners, namely the SCREECH-Anytime and AQA-Anytime reasoners, work in HERAKLES. In particular, we demonstrate this with the SCREECH-Anytime reasoner, as the AQA-Anytime reasoner follows the same design. The underlying concepts of these anytime reasoners have been described in Section 8.4, Chapter 8.

In the previous sections, we have argued that HERAKLES allows running reasoners in the remote mode using the JAVA RMI technology. RMI enables a client object to access remote objects and invoke methods on them as if they were local objects. Figure 10.4 illustrates the different SCREECH reasoners working together to enable anytime reasoning for approximate instance retrieval with named classes.

Suppose that the SCREECH reasoners composed in the SCREECH-Anytime reasoner are located on different servers. The RMI registry on each server stores the

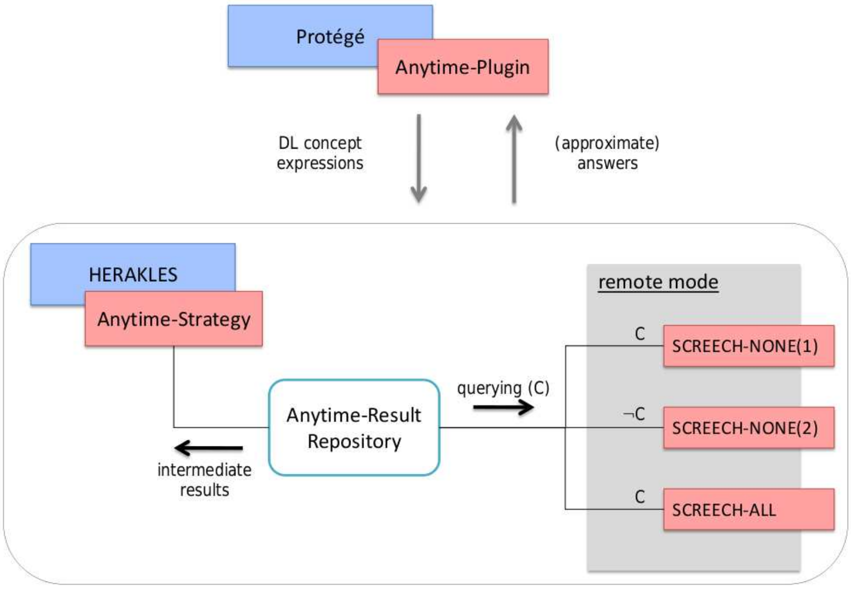


Figure 10.4: Remote Architecture of the SCREECH-Anytime reasoner within the reasoning broker system HERAKLES

remote reasoner along with a name reference for HERAKLES to be able to access it. When setting up, HERAKLES reads the connection details from a configuration file and searches for the required remote reasoners using their name references in the corresponding RMI registry and then attempts to connect to them. The configuration file includes communication details about which reasoners need to be invoked, on which server they are located as well as which ports have to be used to communicate with them. When the user invokes HERAKLES within Protégé, according to the selected loading strategy, HERAKLES loads the ontology being edited to the connected remote reasoners. This task is performed by the Screech Anytime Strategy. In the context of SCREECH-Anytime, each remote SCREECH reasoner loads the ontology and transforms it into a Horn program, invoking the TBox-translation of KAON2. When a query in the Anytime Query plug-in is posed, it is parsed, checked for syntax errors and finally sent to HERAKLES. Consequently, the Screech Anytime Strategy component receives the query and retrieves the remote reasoner references from the RMI registry using the reference names. Once the remote reasoner references are retrieved from the registry, the component implements the anytime-functionality and can invoke operations on the remote reasoner references.

The Screech Anytime Strategy component forwards the query to the Screech Anytime Manager, which in turn forwards it to the remote reasoners and waits for their response. Note that the Screech Anytime Manager is integrated into the Screech Anytime Strategy component and at the same time responsible for synchronizing the remote Screech reasoners. Each remote reasoner receives the query and performs the instance retrieval-reasoning task for the query. When a remote reasoner responds, its result will then be stored in the Anytime Result Repository. The Screech Anytime Manager analyzes each intermediate result from the repository and determines its quality due to the soundness and completeness characteristics of the remote reasoner that provided it. Finally, the Screech Anytime Strategy sends the analyzed result to the Anytime Result plug-in, which displays it according to the color-principle discussed in the previous section.

10.4 Further Potential Applications

A broader application area, for which approximation reasoning methods can be beneficial, is Information Retrieval (IR). This is the science of (full-text) searching for documents, for information within documents, and for metadata about documents. In order to improve the effectiveness of purely text-based search, several issues have to be considered such as semantic relations and handling vaguely defined abstract concepts and time specifications.

Regarding these issues, most of the present IR systems can successfully handle various inflection forms of words using stemming algorithms, it seems that a lot of heuristics and ranking formulas using text-based statistics that were developed in the course of classical IR research during the last decades [BYRN99] cannot master the issues mentioned above [Nag07]. One of the reasons is that term co-occurrence that is used by most statistical methods to measure the strength of the semantic relation between words, is not valid from a linguistic-semantical point of view [Kur05].

Besides term co-occurrence-based statistics another way to improve search effectiveness is to incorporate background knowledge into the search process. The IR community concentrated so far on using background knowledge expressed in the form of thesauri. Thesauri define a set of standard terms that can be used to index and search a document collection (controlled vocabulary) and a set of linguistic relations between those terms, thus promise a solution for the vagueness of natural language, and partially for the problem of high-level concepts. Unfortunately, while intuitively one would expect to see significant gains in retrieval effectiveness with the use of thesauri, experience shows that this is usually not true [Sal86, Nag07]. One of the major cause is the “noise” of thesaurus relations between thesaurus terms. Linguistic relations, such as synonyms are normally valid only between a specific meaning of two words, but thesauri represent those relations on a syntactic level, which usually results in false positives in the search result.

To improve search quality, modern approaches, namely ontology-based information systems, make use of a more sophisticated representation of background knowledge than classical thesauri. Particularly, in recent years, ontology languages, such as RDF and OWL, are of great interest for IR researchers. Using a more expressive language like OWL can improve search quality, however, search efficiency can decrease due to its high computational complexity. Another issue is that the performance of state-of-the-art ontology reasoning systems is not comparable with other well-established technologies, such as relational databases and full-text search engines. Therefore, it is a very challenging task in the development of ontology-based information systems to maintain the good efficiency of current IR systems, while improving their effectiveness at the same time by using expressive ontologies.

Hence, it is very unlikely that solutions using solely sound and complete ontology reasoning for information retrieval will scale so well that they can be used in large web information systems due to the worst case time complexity of ontology reasoning [Nag07]. As the approximation instance retrieval methods developed in this work deal with lowering the worst case time complexity, it would be useful to apply such methods for ontology-based information systems.

In order to illustrate the usefulness of approximate reasoning on information retrieval, consider, for instance, the VICODI system, which was developed within the EU IST VICODI project[15]. The mission of VICODI is to enhance people's understanding of the digital content on the Internet. This is achieved by introducing a novel set of contextualization mechanisms for digital content. The main idea of visual contextualization is to visualize the spatial and temporal aspects of the document context to make the document content more comprehensible for users. The context visualization was based on semantic metadata, which had been semi-automatically generated for the documents, using an ontology of European history, namely the VICODI ontology.

In Section 6.5, Chapter 6, we have evaluated the knowledge compilation method SCREECH using the VICODI ontology revealing that a significant reasoning speed can be achieved. Moreover, there has been a time saving of 50% for all named classes with 100% precision and recall. This fact indicates that approximate reasoning solutions can be useful for ontology-based information retrieval systems like the VICODI system.

10.5 Conclusion

This chapter has discussed the application of the approximate reasoning solutions developed in this work in the context of the research project THESEUS. Addressing the reasoning objectives based on the requirements posed by the THESEUS Use Cases, the role of scalable reasoning of expressive ontologies has been highlighted.

The variety of language features available in the W3C standard for OWL 2 and its profiles as well as the multitude of available OWL reasoners poses the challenge to choose the best performing reasoner on a given ontology and reasoning task. To approach this problem, the reasoning broker system HERAKLES has been introduced, in whose development we have been involved in. Furthermore, it has been described

[15] `http://www.vicodi.org` [accessed 2009-9-9]

how our approximate solutions have been integrated into the reasoning broker system HERAKLES in order to enable it for scalable instance retrieval. To realize the integration, a novel anytime reasoning interface supplementary to the existing most commonly used OWL API has been designed and applied.

Finally, we have highlighted a possible application of approximate reasoning solutions in an ontology-based information retrieval system where a proper tradeoff between search performance and quality is required.

Part IV

Finale

Chapter 11

Related Work

There have been a number of approaches, which addresss the scalability of expressive reasoning for the Semantic Web. Those range from distributed and parallel reasoning to ontology modularization. Since the goal of this thesis is to develop scalable reasoning methods using logical approximation, those works related to approximate reasoning for the Semantic Web will be discussed and compared with the approaches developed in this thesis.

11.1 Approximate Reasoning in the Semantic Web

Approximations are used for dealing with problems that are hard, usually NP-hard or coNP-hard. A significant number of approaches in the approximate reasoing field have been proposed to address intractability of reasoning while staying within the framework of symbolic, formal logic. In this thesis we are concerned only with approximate reasoning for the Semantic Web, and therefore approximating different logics are not within the scope of our examination. However, for completeness in providing a review of related work on approximate reasoning, we start with a brief presentation of the family S of approximate logics [Cad95]. For a critique of these logics, and for further developments in complete but unsound reasoning, the interested reader is referred to the work of Finger and Wassermann [FW02, FW06].

Approximating Logics. The work of Cadoli and Schaerf in [Cad95, SC95] represents a significant milestone in the development of approximate logics. Two families of logics are proposed; S-3 is classically sound but incomplete, while S-1 provides for complete but unsound reasoning. Together these two families can be used to provide varying levels of approximation to classical propositional logic.

For both the S-3 and S-1 logics, a set of propositional atoms S is maintained and serves to identify individual logics. This set is referred to as the parameter set, or more informally, the relevance set. The parameter set serves to control inference and also defines the complexity for a logic. The complexity of reasoning is determined by the size of this set. With a small parameter set the available inferences are severely limited but the complexity of reasoning is of a low order polynomial. As the size of the parameter set approaches the set of all propositional atoms the logics become increasingly classical, both in their inferences and complexity. These logical developments are based on earlier work by Levesque, on logics of limited inference [Lev84], which in turn, draw from work on multi-valued logics [ABD92]. However, the innovation of Cadoli and Schaerf has been to consider a parameterized family of logics, where each individual logic is determined by membership of the parameter set. This has allowed more fine-grained reasoning behaviour, with gradations of logical and computational properties.

Classical approximate reasoning methods have rarely been considered in the context of the Semantic Web. In the following, we will review some work of the approaches dealing with approximate reasoning for the Semantic Web.

Approximate Satisfiability. In addition to the family of approximate logics, Cadoli and Schaerf also investigated approximation of DL reasoning in [SC95]. Regarding this, they proposed a syntactic manipulation of concept expressions that simplifies the task of checking their satisfiability. The method generates two sequences of approximations, one sequence containing weaker concepts and the other sequence containing stronger concepts. The sequences of approximations are obtained by substituting a subconcept D in a concept expression C by a simpler concept.

More precisely, for every subconcept D, this method defines the depth of D to be the number of universal quantifiers occurring in C and having D in its scope. The scope of $\forall R.\phi$ is ϕ which can be any concept term containing D. A sequence of weaker (stronger) approximated concepts can be defined, denoted by $C_i^\top$ ($C_i^\perp$), by replacing every existentially quantified subconcept, *i.e.,* $\exists R.\phi$ where ϕ is any concept term, of depth greater or equal than i by $\top$ ($\perp$). Concept expressions are assumed to be in negated normal form (NNF) before approximating them. The sequences $C^\top$ and $C^\perp$ can be used to gradually approximate the satisfiability of a concept expression. In [SC95], subconcepts $D \equiv \exists R.C$ are replaced as the worst case complexity depends on the nesting of existential and universal quantifiers. The Theorem 1 given in [SC95] leads to the following for $C^\perp$-approximation:

$$\begin{array}{llcll} (I \sqsubseteq Q)_i^\perp & \text{is not satisfiable} & \Leftrightarrow & (I \sqcap \neg Q)_i^\perp & \text{is satisfiable} \quad \Rightarrow \\ (I \sqcap \neg Q) & \text{is satisfiable} & \Leftrightarrow & (I \sqsubseteq Q) & \text{is not satisfiable.} \end{array}$$

Therefore, we are only able to reduce complexity when approximated subsumption tests are not satisfiable. When an approximated subsumption test $(I \sqsubseteq Q)_i^\perp$ is satisfiable, nothing can be concluded and the approximation continues to level $i+1$ until no more approximation is applicable, *i.e.,* the original concept term is obtained. Analogously, from the Theorem 1 one obtains that when $(I \sqsubseteq Q)_i^\top$ is satisfiable this implies that $(I \sqsubseteq Q)$ is satisfiable. When $(I \sqsubseteq Q)_i^\perp$ is not satisfiable nothing can be deduced and the approximation continues to level $i+1$. Research on this kind of

DL approximation is quite limited. This approach is the only method that deals with approximation of satisfiability in DLs.

In contrast to the approach dealing with the approximation of satisfiability in DLs, the focus of our approaches is on the approximation of assertional reasoning over expressive ontologies. Comparing it with our knowledge compilation approach, the this approach deals with a syntactic manipulation of concept expressions while our approach concentrates on a manipulation of knowledge bases. More technically, this method suggests substituting some sub-concepts of a complex concept expression by a simpler concept, whereas our method deals with the replacement of disjunctions. Comparing the AQA approach with this method, our approach defines an approximate semantics and computes the extension of concept expressions by breaking down complex expressions into atomic concepts, instead of substituting certain sub-concepts. Another crucial difference is the logic under consideration. Our solutions deal with more expressive description logics, while the logic considered in the first approach is a less expressive description logic, such as $\mathcal{ALC}$.

Approximation Of Classifying Concepts. In [GSW05], Groot and his colleagues have investigated whether approximation reasoning methods known from the knowledge representation literature can help to simplify OWL reasoning. Concretely, it has been studied how to apply the approximate deduction approach of Cadoli and Schaerf in [SC95] to the problem of classifying new concept expressions.

Classifying a concept expression Q into the concept hierarchy requires a number of subsumption tests for comparing the query concept with other concepts C_i in the hierarchy. As the classification hierarchy is assumed to be known, the number of subsumption tests can be reduced by starting at the highest level of the hierarchy and to move down to the children of a concept only if the subsumption test is positive. The most specific concepts w.r.t. the subsumption hierarchy which passed the subsumption test are collected for the results. Finally, it is to check if the result is subsumed by Q as this implies that both are equal. The idea of Groot here is to approximate the subsumption tests using the method of Cadoli and Schaerf *e.g.,* to approximate the subsumption tests by sequences of weaker and stronger subsumptions. Although this approximation method can contribute to the efficiency of query classification, the experiments taken by Groot have shown that a direct application does not lead to an improvement and that it also suffers from two fundamental problems. One is the collapsing of concept expressions leading to many unnecessary approximation steps. The other is that only in some cases the approximate method is able to successfully replace subsumption tests by cheaper approximations.

Scalable Instance Retrieval. In the following, we describe an approach to ABox reasoning that restricts the language and deals with role-free ABoxes, *i.e.,* ABoxes that do not contain any axioms asserting role relationships between pairs of individuals. Instance Store developed in [LTHB04] addresses the scalability of reasoning with very large ABoxes so is partly close to our approach presented in Chapter 7. To better apprepriciate the difference, the approach is described in detail.

In the case of a role-free ABox, the instances of a concept D could be retrieved simply by testing for each individual x in $\mathcal{A}$ if $KB \models x : D$. This would, however, clearly be very inefficient if $\mathcal{A}$ contained a large number of individuals. An alternative approach is to add a new axiom $C_x \sqsubseteq D$ to $\mathcal{T}$ for each axiom $x : D$ in $\mathcal{A}$, where C_x is a new atomic concept; such concepts will be called pseudo-individuals. Classifying the resulting TBox is equivalent to performing a complete realisation of the ABox: the most specific atomic concepts that an individual x is an instance of are the most specific atomic concepts that subsume C_x and that are not themselves pseudo-individuals. Moreover, the instances of a concept D can be retrieved by computing the set of pseudo-individuals that are subsumed by D. The problem with this latter approach is that the number of pseudo-individuals added to the TBox is equal to the number of individuals in the ABox, and if this number is very large, then TBox reasoning may become inefficient or even break down completely (*e.g.,* due to resource limits).

The basic idea behind the Instance Store is to overcome this problem by using a DL reasoner to classify the TBox and a database to store the ABox, with the database also being used to store a complete realisation of the ABox, *i.e.,* for each individual x, the concepts that x realises (the most specific atomic concepts that x instantiates). The realisation of each individual is computed using the DL (TBox) reasoner when an axiom of the form $x : C$ is added to the Instance Store ABox. A retrieval query to the Instance Store (*i.e.,* computing the set of individuals that instantiate a query concept) can be answered using a combination of database queries and TBox reasoning. Given an Instance Store containing a $KB\langle\mathcal{T}, \mathcal{A}\rangle$ and a query concept Q, the instances of Q are computed as follows. First, the DL reasoner is used to compute C, the set of most specific atomic concepts in $\mathcal{T}$ that subsume Q, and D, the set of all atomic concepts in $\mathcal{T}$ that are subsumed by Q. Next, the database is used to compute A_Q , the set of individuals in A that realise some concept in D, and A_C, the set of individuals in A that realise every concept in C. Then, the DL reasoner is used to compute $A^{'}_{Q}$, the set of individuals $x \in A_C$ such that $x : B$ is an axiom in A and B is subsumed by Q, finally returning the answer $A_Q \cup A^{'}_{Q}$.

On the whole, our approach AQA, in contrast, addresses scalability of reasoning, by giving up soundness and completness, whereas Instance Store preserves soundness and completeness. Moreover, our approach materializes not only atomic extensions, but also materializes atomic role extensions and computes extensions of complex concept expressions based on the simplified semantics, as described in Section 7.2, Chapter 7.

Approximating Queries. In [Sv02], Stuckenschmidt and van Harmelen have introduced a method for approximating conjunctive queries. The method computes a sequence $Q^1, ..., Q^n$ of queries such that: (1) $i < j \Rightarrow Q^i \sqsupseteq Q^j$ and (2) $Q^n \equiv Q$. The first property ensures that the quality of the results of the queries does not decrease. The second property ensures that the last query computed returns the desired exact result. The proposed method can easily be adapted for instantiation checks. The computed sequence $Q^1, ..., Q^n$ is used to generate the sequence $C_1^\Delta, ..., C_n^\Delta$ with $C_i^\Delta = a : Q^i$. Assuming that less complex queries can be answered in less time, instantiation checks can then be speeded up using the following implication:

$$(I \not\sqsubseteq Q') \wedge (Q \sqsubseteq Q') \Rightarrow I \not\sqsubseteq Q.$$

In [Sv02] the sequence of subsuming queries $Q^1, ..., Q^n$ is constructed by stepwise adding a conjunct (of the original query) starting with the universal query. A problem that remains to be solved in this approach is a strategy for selecting the sequence of queries to be checked successively. This problem boils down to ordering the conjuncts of the query which should balance the two factors "smoothness" and "time complexity". As described in [Sv02] the smoothness of the approximation can be guaranteed by analyzing the dependencies between variables in the query. After translating the conjunctive query to a DL expression, these dependencies are reflected in the nesting of subexpressions. As the removal of conjuncts from a concept expression is equivalent to substitution by $\top$, this nesting provides a selection strategy to determine a sequence of approximations S_i where all subexpressions at depth greater or equal than i are replaced by $\top$. Hence, this method is somewhat similar to $C^\top$-approximation except that it is restricted to the conjunctive query, *i.e.,* the instance description is not approximated, and it can replace any conjunct in the query with $\top$, not only existentially quantified conjucts.

In [WGS05], Wache and his colleagues have observed that queries have typically a very flat structure. Considering this, they have proposed an improved strategy extending the above approach with a heuristic for subconcept selection. Moreover, to overcome the flatness of queries typically found in ontologies, the improved strategy provides an order for subexpressions at the same level of depth. A possible ordering is the expected time contribution of a conjunct to the costs of the subsumption test. As measuring the actual time is practically infeasible, a heuristic to determine a suitable measure of complexity for expression is used. The basic idea here is to unfold the conjuncts using the definitions of the concepts from the ontology occurring in the conjunct. In order to determine a suitable measure of complexity for expressions, the standard proof procedure for DLs has been considered. Most existing DL reasoners are based on tableau methods, which determine the satisfiability of a concept expression by constructing a constraint system based on the structure of the expression. As the costs of checking the satisfiability of an expression depends on the size of the constraint system, this size has been considered to be used as a measure of complexity.

As determining the exact size of the constraint system requires to run the tableau method, heuristics are used for estimating the size. Based on this estimated size, one can determine the order in which conjuncts at the same level of depths are considered. This strategy was implemented in the Instance Store system and evalutated on the Gene ontology[1]. As reported in [WGS05], the experiment has shown some potential for speeding up instance retrieval. However, a comprehensive evaluation is still required so questions on the applicability of this approach to more expressive knowledge bases are still open.

In contrast to our AQA approach, the focus of the approaches discussed above is to approximate conjunctive queries, thus, proposing a rewriting technique of conjunctive queries into concept expressions. Moreover, the focus of our AQA approach is to provide scalable instance retrieval for complex concept expressions by applying approximate semantics. This means, our focus is not on the approximation of conjunctive queries, but on complex concept expressions.

Anytime Classification Instead of answering an approximated query or approximating deduction, an approach taken by Schlobach et al. in [SBK$^+$07] addresses the scalability of terminological subsumption by approximating ontologies. Moreover, a method for approximate subsumption has been proposed which is based on ignoring a selected part of the set of concepts in an ontology. By incrementally increasing this set, one can construct an anytime algorithm for approximate subsumption, which yields sound but possibly incomplete answers w.r.t. classical subsumption queries. The behaviour of this approximate algorithm is dependent on the strategy for selecting the subset of concepts taken into account by the approximation.

Based on the formal definitions such as lower and upper S-approximations for interpreting $\mathcal{ALC}$ concepts and approximate subsumption, the authors have shown how this approximate subsumption on an ontology can be computed in terms of a classical subsumption check on a rewritten version of the ontology. The approximate subsumption method has been applied to a set of 8 benchmark ontologies, comparing its performance against classical subsumption in terms of runtime and recall. An important aspect in applying this method is to define an approximated vocabulary, *i.e.,* to build an increasing sequence of approximation sets of vocabulary by adding new concepts. Regarding this, three different strategies have been suggested to define an approximated vocabulary, based on the occurrence frequency of the concepts in the axioms. The RANDOM strategy simply chooses concepts randomly, whereas MORE adds the most occurring concept first. LESS adds the least occurring concepts first.

The experiments as reported in [SBK$^+$07] have shown that the gain in runtime outweights the loss in recall on 5 out of 8 cases. Furthermore, the gain is larger for ontologies where classical subsumption queries are expensive. Hence, it has been revealed that the approximation algorithm works best when it is most needed, namely on complex ontologies. The experiments have also shown that of the three strategies considered, the most-frequent-first strategy performed best, in particalur on those ontologies with a skewed occurrence frequency of the concepts in the axioms. The test ontologies were relatively large and expressive, however, they were simplified to corresponding $\mathcal{ALC}$ versions since the approximate method has been only designed for

[1] `http://www.geneontology.org/` [accessed 2009-9-12]

handling $\mathcal{ALC}$ ontologies. To this end, the effect of this approximation method on more expressive OWL ontologies is still unknown.

Although this method applies approximation in a similar way like our approximate solutions, the reasoning task considered is quite different. The reasoning task targeted in this approach is the problem of subsumptions, while our approaches deal with the approximation of assertional reasoning.

11.2 Evaluation of Reasoning Systems

There have been a number of efforts made to evaluate DL reasoning systems. The main concern of those approaches is to develop benchmarks for testing reasoning performance. The development of automated benchmarking frameworks has been largely neglected. Measuring degree of query soundness and completeness and its difficulty in testing a large set of queries have been rather sparsely addressed so far. The related work in this context has been reviewed focusing on the development of automated benchmarking tools as well as the measurement of the degree of query soundness and completeness. Two frameworks have been identified which strive for a similar goal like that of our automated benchmarking framework introduced in Section 9.5, Chapter 9.

LUBM. A benchmarking framework called LUBM (Lehigh University Benchmark), has been proposed in [GQPH07]. It suggests a collection of requirements for developing benchmark for knowledge representation systems, adapting the requirements used in the database community. The LUBM provides an automated approach which allows to measure degree of the soundness and correctness of reasoning systems, analyzing how many of the correct answers are returned and how many of the returned answers are correct. It also provides a benchmark ontology about the university domain along with some test queries. Furthermore, it presents a synthetic data generation approach for OWL ontologies that is scalable and models the real world data. The data-generation algorithm [WGQH05] learns from real domain documents, and generates test data based on the extracted properties relevant for benchmarking. In this framework, a collection of performance metrics have been considered such as loading time, repository size and query response time including query soundness and completeness. Loading time is simply the time taken to load benchmark ontology while repository size measures the size of the repository of systems with persistent storage. It measures the degree of completeness of each query answer as the percentage of the entailed answers that are returned by the system while the degree of soundness is measured as the percentage of the answers returned by the system that are actually entailed.

The framework includes interpretation methods generating various charts. Three alternative interpretation ways of trading query response time and query completeness at the same time are introduced. Using clustered columns, the first method simply demonstrates the relationship between query response time and the degree of query completeness on each query. The second method uses a scatter chart to show query response time and query completeness for individual queries in different systems. Compared to the former one, the method is better in illustrating a flavor of the overall performance of the systems. The third method shows the generalisation of the results, *i.e.,* average query time and average completeness per system. On the whole, this framework is not designed to tackle the issues related to measuring soundness and completeness in the presence of a large set of queries.

In [MYQ$^+$06], a benchmarking framework called UOBM standing for University Ontology Benchmark, has been introduced which extends the LUBM framework in terms of inference and scalability testing. The framework provides both OWL Lite and OWL DL ontologies covering a complete set of OWL Lite and DL constructs, respectively. Furthermore, it includes necessary properties to construct effective instance links and improve the LUBM data generation methods to make the scalability testing more convincing. Moreover, the ABox of an ontology is enriched by interrelations between individuals of formerly separated units, which then requires ABox reasoning to answer the set of given UOBM queries.

In terms of measurements, such as measuring response time of reasoning systems as well as the degree of soundness and completeness, the aim of the LUBM framework as well as UOBM is comparable to that of the automated framework developed in this work. However, these frameworks provide a benchmark with a given ontology and a fixed number of queries, which are manually constructed. In contrast, the focus of our benchmarking framework described in Chapter 9 is on (1) automated execution of benchmarks, which can embed arbitrary OWL ontologies, and (2) automated generation of test queries, especially for instance retrieval. Another significant difference is the scalability of the frameworks. Our framework is able to deal with several thousands of queries to benchmark reasoning systems, whereas the current implementation of these frameworks is designed to benchmark queries that are manually constructed.

BENCHEE. Another framework we have analysed is the Benchee framework [KS04] developed by the Racer group[2]. Benchee is a benchmark-testing infrastructure that supports and standardizes the creation and execution of test benchmarks for the DL reasoning system Racer. It provides a simple, easy-to-use, intuitive GUI for editing benchmarks and monitoring benchmark executions. Benchmarks are specified in a Benchee specific format which can be further processed and plotted by the gnuplot tool[3]. Currently, Benchee does not have support for benchmarking other reasoning systems. However, it enables to benchmark different versions of the Racer system, using Racer servers and detect changes in their performances.

[2] `http://www.racer-systems.com` [accessed 2009-09-22]

[3] Gnuplot is a portable command-line driven graphing utility.

A unique feature of this framework is the use of *nRQL* [HMS$^+$04] language for specfying benchmark commands and telling Racer servers what they should perform. As a matter of fact, this framework cannot be used for benchmarking approximate reasoning systems due to two reasons. First, it is designed to benchmark only Racer systems. Second, it does not address the need of measuring the degree of query soundness and completeness.

Chapter 12

Conclusion and Outlook

This chapter concludes the thesis by summarizing the results and outlining possible directions for further research. In Section 12.1, we summarize the contents of this work and accentuate the major contributions. Subsequently, Section 12.2 reviews open research questions that are directly related to our work and gives an outlook on how those could be addressed in future work. Additionally, an outlook on a broader area of approximate reasoning for the Semantic Web is given as a concluding remark.

12.1 Summary of the Thesis

In this work we have posed the hypothesis: *Using approximation, scalable reasoning over expressive ontologies can be achieved in a controlled and well-understood way.* This hypothesis has been investigated by means of contributions in the areas of knowledge compilation, query answering and resource-bounded reasoning reflected by the structure of this thesis.

In this regard, the goal of this thesis has been to develop approximate reasoning methods whose computational properties can be established in a well-understood way, namely in terms of soundness and completeness, and whose quality can be analyzed in terms of statistical measurements, namely recall and precision. As a result, an approximate reasoning framework for scalable instance retrieval for expressive ontologies has emerged with the support of resource-bounded reasoning by composed anytime algorithms as well as automated benchmarking.

12.1.1 State of the Art of Expressive Reasoning Techniques

It has been extensively studied how to efficiently reason over expressive knowledge bases with large TBoxes. A number of reasoning systems for expressive description logics, namely $\mathcal{SHIQ}$, have been developed based on tableau algorithms, which run in 2NExpTime. Although existing DL reasoning systems promise an efficient implementation, the underlying worst-case complexity is discouraging, in particular, when reasoning with a large volume of instance data. Towards scalable reasoning with large ABoxes, a significant effort has been made, which is based on resolution principles [Mot06]. Its aim is to lower the high computational complexity of query answering; it has been illustrated that by translating expressive ontologies into disjunctive datalog programs and by applying efficient deductive database techniques, query answering in expressive description logic $\mathcal{SHIQ}(\mathbf{D})$ is possible with the worst data complexity of NP. The overall complexity is however still EXPTIME. Although considerable progress has been made in the last ten years in realizing scalable reasoning, an important issue is how to lower the high computational complexities of reasoning in expressive languages, and enable scalable reasoning techniques without the expressivity power needed by emerging applications in the Semantic Web.

As a field that has emerged in logic and artificial intelligence, approximate reasoning addresses the intractable problems of logical reasoning and deals with lowering the high computational complexities. Its basic idea is to speed up reasoning by trading off the quality of reasoning results against increased speed.

Based on the ideas and techniques developed in this field, the main goal of this thesis has been to develop practicable, approximate reasoning methods for assertional reasoning, aiming to lower high computational complexities that a state-of-the-art DL reasoning system will have to face in emerging large-scale semantic applications.

In this regard, this thesis has examined the requirements of achieving scalable reasoning over expressive knowledge bases with large and complex TBoxes as well as large ABoxes. Several results have been achieved that are not yet to be found in related work. We will summarize the contributions by describing the individual components of the resulting framework.

12.1.2 Knowledge Compilation

The hypothesis we have dealt with in this context was: *Scalable reasoning in large and expressive knowledge bases of expressive (intractable) ontology languages can be achieved by compiling them into less expressive ones.* Supporting this hypothesis, we have presented the knowledge compilation method SCREECH in Chapter 6. Based on the fact that data complexity is polynomial for non-disjunctive datalog, SCREECH provides several approximation strategies that compile disjunctive datalog programs into Horn programs, thus enabling tractable ABox reasoning over expressive ontologies with large data sets. Consequently, a Horn program generated by SCREECH is approximate with respect to the original one. In general, it is conceivable that one can generate many different Horn programs from a disjunctive program. Hence, an important issue in developing SCREECH has been to determine practicable approximations for generating Horn programs. On the one hand, such approximations are supposed to provide good approximate answers. On the other hand, they ought to speed up reasoning. Another requirement has been to explore those approximations whose logical properties can be determined in terms of soundness and completeness. After having analyzed several approximations, three approximation strategies, called SCREECH-ALL, SCREECH-ONE, and SCREECH-NONE, have been formed and their logical properties have been analyzed. As a result, SCREECH-ALL is a complete, yet unsound approximation. SCREECH-NONE is sound, but incomplete, whereas SCREECH-ONE is unsound and incomplete. SCREECH can be applied to the DL reasoning system KAON2, since it transforms knowledge bases of the expressive description logic $\mathcal{SHIQ}$ to disjunctive datalog programs. In order to estimate the practicability of SCREECH, well-known benchmarking ontologies have been used. The quality of each approximation as well as its reasoning performance has been compared with those of KAON2. We conjecture that a knowledge compilation method like SCREECH is practicable in emerging Semantic Web applications deploying expressive ontologies with large data sets when their knowledge bases do not significantly change over time.

12.1.3 Approximate Query Answering

The hypothesis we have evaluated in this context was: *Scalable reasoning in large and expressive knowledge bases of intractable ontology languages can be also achieved by query approximation.* As a result, a fast approximate reasoning system for instance retrieval, AQA, has been conceived and implemented. Central to this approach is the approximate semantics for computing the approximate extension of a complex concept expression based on the atomic extensions as introduced in Section 7.2, Chapter 7. Based on this semantics, several algorithms have been introduced to effectively compute approximate extensions, namely database, offline and online variants.

In the database variant, approximate extensions of a complex concept are computed by mapping them into a relational algebra expression and evaluated by using the materialization of atomic extensions. In the offline variant, approximate extensions of a complex concept are computed in computer memory based on the materialization of atomic extensions. Materialization plays a major role in these variants. It has been illustrated that it is possible to achieve a speedup of about factor 10, while the number of introduced errors varies depending on the query, but is within reasonable bounds. These variants have proved to be very useful when dealing with large data sets.

Although materialization appears to be a promising technique for fast query answering, a materialization process is rather time-consuming. In some applications that deal with relatively large ontologies, it would be impracticable to materialize the whole of atomic extensions. Hence, the online variant is intended to access atomic extensions by invoking a sound and complete DL reasoner at query-time as well as computing approximate extensions in computer memory. Furthermore, the combination with the knowledge compilation method SCREECH has extended this variant. This means, SCREECH provides it with approximate knowledge bases, in which reasoning can be sped up in some application cases.

Regarding an effective dealing with time-consuming materialization, an efficient strategy has been developed and implemented, which enables AQA to maintain materialization incrementally and on-demand. Finally, a comprehensive empirical analysis of AQA reporting positive results has been conducted by using the well-known expressive benchmarking ontology WINE. One purpose of this experiment has been to reveal how fast the AQA variants are in comparison to the efficient ABox reasoner KAON2. Another purpose has been to measure the quality of the approximate instance retrieval. The evaluation has demonstrated that it is possible to achieve a significant performance improvement for ABox reasoning over expressive ontologies with large ABoxes and TBoxes. Moreover, the evaluation has shown that a significant speed-up of about factor 10 can be obtained. It has also been shown that the approximate instance retrieval yields fewer errors for many practical complex queries than that for simple queries.

12.1.4 Evaluating Approximate Reasoning Systems

The hypothesis we have dealt with in this context was: *The development of approximate reasoning methods needs well-defined methodological guidelines and tool support to measure correctness and performance.* Clearly, the ultimate goal of approximate reasoning research is to develop approximate reasoning systems that are both effective and efficient. The performance of an approximate reasoning algorithm can be easily evaluated by measuring response time. Evaluating the quality of an approximate reasoning algorithm, however, is a highly complex issue. Chapter 5 has provided a solid mathematical foundation for the assessment and comparison of approximate reasoning algorithms with respect to correctness, run-time and anytime behavior. This general framework can serve as a means to classify algorithms with respect to their respective characteristics and help in deciding which algorithm best matches the demands of a concrete reasoning scenario. With regard to a concrete implementation of that general framework, we have analyzed the difficulty in benchmarking approximate reasoning systems and identified that tool support is important. Concerning this, we have derived a number of requirements, which an automated tool has to meet for evaluating approximate reasoning systems. The implementation of the automated benchmarking framework has been described in Section 9.5, Chapter 9.

12.1.5 Development of Composed Anytime Algorithms

Applying anytime computation is an important aspect in the Semantic Web as it is linked to the field of resource-bounded reasoning. Unlike standard algorithms that grant a fixed quality of output and completion time, anytime algorithms for resource-bounded reasoning are meant to be interruptible at any time. In practice, it would not always be beneficial to develop algorithms that continuously generate intermediate answers whose quality is unknown. Instead it would be much more beneficial if the quality of the answers were known.

In Chapter 8, we have analyzed how to construct anytime algorithms by composing approximate reasoning algorithms. With regard to this, we have investigated how to combine the approximate reasoning approaches presented in Chapters 6 and

7. The main goal of this investigation had been to improve the quality of approximate algorithms with different computational characteristics. As a result, two anytime reasoners, namely SCREECH-Anytime and AQA-Anytime, have been designed and implemented. The resulting reasoners are not interruptible at any time, but at the time when one approximate algorithm involved in the composition finishes its computation and is able to continue. The advantage of such compositions is that the quality of intermediate answers can be clearly distinguished, since soundness and completeness of the composed approximate algorithms are known.

Furthermore, introducing a novel anytime interface designed for anytime computation in the OWL API, we have integrated the anytime reasoners into the reasoning broker system HERAKLES, which is being developed in the research project THESEUS. HERAKLES aims at providing scalable reasoning services for sophisticated Semantic Web applications of the future of the Internet, which are being developed in THESEUS. We conjecture that extending the reasoning broker with anytime behavior for scalable instance retrieval will be useful for emerging web-scaled Semantic Web applications of expressive, complex ontologies.

12.2 Further Work

The work in this thesis has opened up many possibilities for future work. In this section, we will sketch open research issues that can be derived from the results achieved in this work as next steps in regard to various aspects.

Development of Approximate Reasoning Systems — Since the focus of this thesis is to develop approximate reasoning algorithms for instance retrieval, the corresponding benchmarking framework has been developed to support benchmarking for instance retrieval. As an initial future step, it would be valuable to extend the benchmarking framework such that it also supports other reasoning tasks such as satisfiability, subsumption checking as well as classification.

A key aspect to show the advantages of approximate reasoning solutions in contrast to the correct one is to develop artificial ontologies in addition to existing real ontologies. Such artificial ontologies will be crucially helpful to explore the advantages and disadvantages of approximate solutions, thus improving them. Hence, it is very valuable to develop methodological guidelines and appropriate tools to support the modeling of such benchmarking ontologies.

Besides measuring reasoning performance, measuring the degree of completeness and soundness plays an important role in developing approximate solutions. Yet, without an explanation of how and why such answers have been derived, it is hard to improve approximate solutions. Explanation is mostly provided in the context of ontology design to help an ontology designer to rectify problems identified by reasoning support, or to explain to a user why an application behaves in an unexpected manner. In developing an approximate reasoning algorithm, explanation is very important and needs to be investigated. Analyzing ontologies and queries, and attempting to explain the output is not sufficient, and in most cases, it is not even possible to derive correct explanations. Thus, it is valuable to develop methods to visualize and explain inferences that produce approximate answers. This will help understanding the behavior of approximate solutions, thus improving them. In this context, an important future topic is the development of explanation tools that do not only support the ontology design lifecycle, but also the development of approximate reasoning algorithms.

Scalable Conjunctive Query Answering — The development of a decision procedure for conjunctive query answering in expressive DLs is a recent topic of great interest within the DL community. On the other hand, scalable conjunctive query answering is an important requirement for many large-scale Semantic Web applications.

Conjunctive query answering in OWL-DL ontologies is, however, intractable in the worst case. Moreover, it has been illustrated that conjunctive query answering in expressive DLs between $\mathcal{ALCI}$, the fragment of $\mathcal{SHIQ}$ with inverse roles, and $\mathcal{SHIQ}$ is 2EXPTIME-complete. On the other hand, Calvanese et al. have argued that true scalability of conjunctive query answering in DL ontologies can only be achieved by making use of standard relational database management systems [CDL$^+$07a]. Hence, it would be of interest to extend the AQA approach for conjunctive query answering. A starting point for this extension would be to make use of the so-called rolling-up technique based on the idea of rewriting conjunctive queries into equivalent concept expressions [Tes01].

Another more comprehensive treatment would be to make use of the recent theoretical results obtained in [GHLS07]. It has been shown that all $\mathcal{SHIQ}$ conjunctive queries can be rewritten into concept expressions that contain role conjunctions. Reasoning with role conjunctions is, however, computationally harder (2EXPTIME-complete) than reasoning without them. Hence, it would be of great interest to apply these novel theoretical results and to extend AQA with regard to conjunctive queries.

Parameterized Knowledge Compilation — While slightly more speculative, there are a number of possibilities for the development of alternative compilation techniques based on various approximate logics. A strong theme in knowledge compilation is that there is always a balance to be made between space and computation. That is, it is not possible to compile an arbitrary knowledge base into a structure that is guaranteed to be polynomial in the size of the original knowledge base yet also offers polynomial time query answering. Compilation techniques based on approximate logics offers a useful framework for balancing the opposing forces of space and time. A particularly exciting area for the application of approximate logics to knowledge compilation is the development of relevant knowledge compilation techniques. A relevant knowledge compilation technique would allow a knowledge base to be separated into domain partitions. Querying within a particular domain would be subject to computational guarantees, while querying across domains would have no such guarantees. The various partitions could be defined with reference to particular parameter sets of an approximate logic, where the parameter sets are determined through the use of machine learning techniques or by a knowledge base designer. Consequently, it would be worth exploring the extent to which these techniques can be applied as a means towards achieving relevant knowledge compilation.

In summary, this work has focused on the development of approximate query answering techniques and showed how scalable instance retrieval can be performed over expressive ontologies with large and complex TBoxes as well as large ABoxes. This is a step towards realizing the vision of approximate reasoning on the Semantic Web. Although additional research questions have to be addressed to fully reach this goal, the presented approximate ontology reasoning framework provides a solid basis for further deployment as well as further research.

Part V

Appendix

Chapter 13

Appendix A

13.1 Used Data Structures

This section describes the abstract data type **list** and its access functions that are used to describe the algorithms defined in Chapters 6 through 9.

Definition 13.1 (List). *Listsare abstract data types which can be regarded as special tuples. They are sequences where every item is of the same type. We introduce functions that will add elements to or remove elements from lists; that sort lists or search within them. Like tuples, lists can be defined using parenthesis in this thesis. The single elements of a list are accessed by their index written in brackets ($(a, b, c)[1] = b$) where the first element has the index 0 and the last element has the index $n - 1$ (while n is the count of elements in the list: $n = \mathrm{len}((a, b, c)) = 3$) . The empty list is abbreviated with nil.*

Definition 13.2 (createList). *The $l = \mathrm{createList}(n, q)$ method creates a new list l of the length n filled with the item q.*

$$l = \mathrm{createList}(n, q) \Leftrightarrow \mathrm{len}(l) = n \wedge \ \forall 0 \leq i < n \Rightarrow l[i] = q \tag{13.1}$$

Definition 13.3 (insertListItem). *The function $m = \mathrm{insertListItem}(l, i, q)$ creates a new list m by inserting one element q in a list l at the index $0 \leq i \leq \mathrm{len}(l)$. By doing so, it shifts all elements located at index i and above to the right by one position.*

$$\begin{aligned} m = \mathrm{insertListItem}(l, i, q) \Leftrightarrow \quad & \mathrm{len}(m) = \mathrm{len}(l) + 1 \wedge m[i] = q \wedge \\ & \forall j : 0 \leq j < i \Rightarrow m[j] = l[j] \wedge \\ & \forall j : i \leq j < \mathrm{len}(l) \Rightarrow m[j+1] = l[j] \end{aligned} \tag{13.2}$$

Definition 13.4 (addListItem). *The* addListItem *function is a shortcut for inserting one item at the end of a list:*

$$\mathrm{addListItem}(l, q) \equiv \mathrm{insertListItem}(l, \mathrm{len}(l)\,, q) \tag{13.3}$$

Definition 13.5 (deleteListItem). *The function $m = \text{deleteListItem}(l, i)$ creates a new list m by removing the element at index $0 \leq i < \text{len}(l)$ from the list l ($\text{len}(l) \geq i + 1$).*

$$\begin{aligned} m = \text{deleteListItem}(l, i) \Leftrightarrow \quad & \text{len}(m) = \text{len}(l) - 1 \wedge \\ & \forall j : 0 \leq j < i \Rightarrow m[j] = l[j] \\ & \forall j : i < j < \text{len}(l) \Rightarrow m[j-1] = l[j] \end{aligned} \tag{13.4}$$

Definition 13.6 (deleteListRange). *The method $m = \text{deleteListRange}(l, i, c)$ creates a new list m by removing c elements beginning at index $0 \leq i < \text{len}(l)$ from the list l ($\text{len}(l) \geq i + c$).*

$$\begin{aligned} m = \text{deleteListRange}(l, i, c) \Leftrightarrow \quad & \text{len}(m) = \text{len}(l) - c \wedge \\ & \forall j : 0 \leq j < i \Rightarrow m[j] = l[j] \wedge \\ & \forall j : i + c \leq j < \text{len}(l) \Rightarrow m[j-c] = l[j] \end{aligned} \tag{13.5}$$

Definition 13.7 (countOccurences). *The function $\text{countOccurences}(x, l)$ returns the number of occurrences of the element x in the list l.*

$$\text{countOccurences}(x, l) = |\{i \in 0 \ldots \text{len}(l) - 1 : l[i] = x\}| \tag{13.6}$$

Chapter 14

Appendix B

14.1 Detailed Evaluation Results

In this section, we will present the detailed experimental results conducted by comparing the AQA system with KAON2, a summary of which has already been presented in Chapter 7 on page 81.

Tables 14.1 through 14.4 show the detailed experimental results regarding the running times required by AQA and KAON2 for querying concept queries built from the constructors, such as $\exists$, $\forall$, $\sqcup$ and $\sqcap$, for instance retrieval on the ontology WINE.

t_{db} illustrates the running times measured for the database variant, $t_{offline}$ displays those measured for the offline variant, whereas t_{kaon2} shows the running times measured for KAON2. The measured times are given in milliseconds. The last row of each table then displays the sum of the respective values of each column.

As regards measuring the quality of approximation, ext_{aqa} gives the extensions for each query computed by AQA, whereas ext_{kaon2} reports the extensions computed by KAON2. *miss* indicates the elements of the conventional extensions, which were not found by AQA, *corr* indicates those, which were correctly found, and finally, *more* indicates those individuals which were incorrectly computed by AQA.

Table 14.1: Detailed experimental results for ∃-queries.

id	C_i	t_{db}	$t_{offline}$	t_{kaon2}	ext_{aqa}	ext_{kaon2}	*miss*	*corr*	*more*
1	∃locatedIn.NOM23	300	211	3113	205	369	164	205	0
2	∃madeFromGrape.NOM8	113	4	2496	0	41	41	0	0
3	∃madeFromGrape.NOM62	132	3	2887	0	123	123	0	0
4	∃madeFromGrape.NOM29	99	4	2319	0	246	246	0	0
5	∃madeFromGrape.WineGrape	116	12	2970	41	2173	2132	41	0
6	∃locatedIn.NOM10	209	129	2513	82	82	0	82	0
7	∃locatedIn.NOM41	248	138	2891	656	656	0	656	0
8	∃locatedIn.Region	968	83	2814	2665	3362	697	2665	0
9	∃adjacentRegion.Region	154	7	2921	82	82	0	82	0
10	∃hasMaker.Winery	1026	68	2232	2132	2173	41	2132	0
11	∃hasBody.WineBody	338	9	2961	1681	2173	492	1681	0
12	∃hasWineDescriptor.WineBody	521	36	2472	1681	2173	492	1681	0
13	∃hasBody.NOM40	375	11	3003	1681	2173	492	1681	0
14	∃hasWineDescriptor.NOM40	567	74	2696	1681	2173	492	1681	0
15	∃hasSugar.WineTaste	397	10	2943	1640	2173	533	1640	0
16	∃hasFlavor.WineTaste	396	11	2455	1763	2173	410	1763	0
17	∃hasBody.WineTaste	413	12	2898	1681	2173	492	1681	0
18	∃hasWineDescriptor.WineTaste	675	44	2644	1804	2173	369	1804	0
19	∃locatedIn.NOM38	356	46	2943	1435	1435	0	1435	0
20	∃hasBody.NOM59	245	12	2465	1148	1189	41	1148	0
21	∃hasWineDescriptor.NOM59	398	44	2859	1148	1189	41	1148	0
22	∃hasBody.NOM51	301	11	2443	1558	1722	164	1558	0
23	∃hasWineDescriptor.NOM51	397	44	2890	1558	1722	164	1558	0
24	∃hasBody.NOM49	222	10	2528	1025	1107	82	1025	0
25	∃hasWineDescriptor.NOM49	277	29	2978	1025	1107	82	1025	0
26	∃hasBody.NOM45	181	8	2493	533	656	123	533	0

Detailed experimental results for ∃-queries (continued)

id	C_i	t_{db}	$t_{offline}$	t_{kaon2}	ext_{aqa}	ext_{kaon2}	*miss*	*corr*	*more*
27	∃hasWineDescriptor.nom45	282	39	2990	533	656	123	533	0
28	∃hasSugar.nom7	288	9	2544	1517	1517	0	1517	0
29	∃hasWineDescriptor.nom7	414	41	3108	1517	1517	0	1517	0
30	∃hasSugar.nom18	306	11	2387	1640	2173	533	1640	0
31	∃hasWineDescriptor.nom18	538	42	2932	1640	2173	533	1640	0
32	∃hasSugar.nom12	275	8	2381	1435	1763	328	1435	0
33	∃hasWineDescriptor.nom12	361	62	2993	1435	1763	328	1435	0
34	∃hasSugar.WineSugar	526	9	2531	1640	2173	533	1640	0
35	∃hasWineDescriptor.WineSugar	603	45	3047	1640	2173	533	1640	0
36	∃madeFromGrape.nom25	130	1	2361	0	82	82	0	0
37	∃hasColor.nom9	103	16	2827	41	2173	2132	41	0
38	∃hasWineDescriptor.nom9	372	41	2497	41	2173	2132	41	0
39	∃hasColor.nom32	99	1	2971	0	41	41	0	0
40	∃hasWineDescriptor.nom32	225	40	2483	0	41	41	0	0
41	∃hasColor.WineColor	106	1	3100	41	2173	2132	41	0
42	∃hasWineDescriptor.WineColor	356	57	3109	41	2173	2132	41	0
43	∃locatedIn.nom2	199	50	2362	82	164	82	82	0
44	∃hasColor.nom28	99	2	3079	0	1066	1066	0	0
45	∃hasWineDescriptor.nom28	187	48	2423	0	1066	1066	0	0
46	∃locatedIn.nom19	235	36	2985	164	287	123	164	0
47	∃locatedIn.nom31	246	35	2557	0	41	41	0	0
48	∃hasSugar.WineDescriptor	399	10	2900	1640	2173	533	1640	0
49	∃hasFlavor.WineDescriptor	422	12	3012	1763	2173	410	1763	0
50	∃hasBody.WineDescriptor	467	11	2533	1681	2173	492	1681	0
51	∃hasColor.WineDescriptor	117	3	3345	41	2173	2132	41	0
52	∃hasWineDescriptor.WineDescriptor	901	56	2733	1804	2173	369	1804	0

Detailed experimental results for ∃-queries (continued)

id	C_i	t_{db}	$t_{offline}$	t_{kaon2}	ext_{aqa}	ext_{kaon2}	*miss*	*corr*	*more*
53	∃madeFromGrape.NOM6	138	2	3205	41	287	246	41	0
54	∃hasColor.NOM5	138	1	3021	41	984	943	41	0
55	∃hasWineDescriptor.NOM5	298	42	2527	41	984	943	41	0
56	∃locatedIn.NOM17	201	50	2991	82	82	0	82	0
57	∃locatedIn.NOM42	195	43	2482	0	41	41	0	0
58	∃hasFlavor.NOM48	211	17	3230	1066	1148	82	1066	0
59	∃hasWineDescriptor.NOM48	291	64	3244	1066	1148	82	1066	0
60	∃madeFromGrape.NOM3	108	2	2463	0	41	41	0	0
61	∃locatedIn.NOM16	220	44	3131	0	41	41	0	0
62	∃hasFlavor.NOM64	268	9	2496	1271	1312	41	1271	0
63	∃hasWineDescriptor.NOM64	416	78	2936	1271	1312	41	1271	0
64	∃hasFlavor.NOM50	290	9	3092	1558	1640	82	1558	0
65	∃hasWineDescriptor.NOM50	439	40	2444	1558	1640	82	1558	0
66	∃hasFlavor.NOM35	297	9	3087	1763	2173	410	1763	0
67	∃hasWineDescriptor.NOM35	486	41	3142	1763	2173	410	1763	0
68	∃hasFlavor.WINEFLAVOR	315	10	2405	1763	2173	410	1763	0
69	∃hasWineDescriptor.WINEFLAVOR	562	42	3223	1763	2173	410	1763	0
70	∃madeFromGrape.NOM33	142	2	2975	0	41	41	0	0
71	∃locatedIn.NOM57	301	43	2327	0	41	41	0	0
72	∃locatedIn.NOM47	197	73	3009	41	82	41	41	0
73	∃madeFromGrape.NOM63	105	2	2468	41	82	41	41	0
74	∃madeFromGrape.NOM11	106	1	2926	41	246	205	41	0
75	∃madeFromGrape.NOM43	103	1	2877	0	123	123	0	0
76	∃hasBody.NOM56	141	18	2338	123	246	123	123	0
77	∃hasWineDescriptor.NOM56	221	33	2996	123	246	123	123	0
78	∃madeFromGrape.NOM60	99	1	2986	0	41	41	0	0

Detailed experimental results for $\exists$-queries (continued)

id	C_i	t_{db}	$t_{offline}$	t_{kaon2}	ext_{aqa}	ext_{kaon2}	*miss*	*corr*	*more*
79	$\exists$locatedIn.NOM46	220	40	2321	0	41	41	0	0
80	$\exists$locatedIn.NOM61	235	85	3281	0	41	41	0	0
81	$\exists$locatedIn.NOM14	234	50	2917	0	41	41	0	0
82	$\exists$hasSugar.NOM54	136	8	2516	82	164	82	82	0
83	$\exists$hasWineDescriptor.NOM54	220	35	3152	82	164	82	82	0
84	$\exists$hasFlavor.NOM55	150	9	2827	205	410	205	205	0
85	$\exists$hasWineDescriptor.NOM55	258	35	2326	205	410	205	205	0
86	$\exists$hasFlavor.NOM58	181	9	3072	492	615	123	492	0
87	$\exists$hasWineDescriptor.NOM58	253	27	2464	492	615	123	492	0
88	$\exists$locatedIn.NOM15	323	45	3106	1271	1271	0	1271	0
89	$\exists$madeFromGrape.NOM24	124	1	2987	0	328	328	0	0
90	$\exists$locatedIn.NOM30	251	43	2461	0	41	41	0	0
91	$\exists$locatedIn.NOM52	215	48	3022	0	41	41	0	0
92	$\exists$madeFromGrape.NOM27	103	2	3069	41	123	82	41	0
93	$\exists$hasSugar.NOM1	152	6	2560	123	246	123	123	0
94	$\exists$hasWineDescriptor.NOM1	212	40	3193	123	246	123	123	0
95	$\exists$madeFromGrape.NOM34	107	2	3139	0	82	82	0	0
96	$\exists$locatedIn.NOM26	236	48	2452	0	41	41	0	0
97	$\exists$locatedIn.NOM44	198	53	2996	82	246	164	82	0
98	$\exists$locatedIn.NOM36	193	41	3122	0	41	41	0	0
99	$\exists$madeFromGrape.NOM39	99	1	2433	0	164	164	0	0
100	$\exists$hasSugar.NOM53	171	8	3094	205	246	41	205	0
101	$\exists$hasWineDescriptor.NOM53	330	40	3090	205	246	41	205	0
102	$\exists$hasVintageYear.VintageYear	104	2	2175	41	41	0	41	0
103	$\exists$madeFromGrape.NOM37	100	2	3064	0	164	164	0	0
104	$\exists$madeFromGrape.NOM21	102	2	2315	0	41	41	0	0

Detailed experimental results for ∃-queries (continued)

id	C_i	t_{db}	$t_{offline}$	t_{kaon2}	ext_{aqa}	ext_{kaon2}	*miss*	*corr*	*more*
105	∃madeFromGrape.NOM20	99	1	2940	0	205	205	0	0
106	∃locatedIn.NOM22	238	68	2897	0	41	41	0	0
107	∃madeFromGrape.GRAPE	119	3	2443	41	2173	2132	41	0
	$\sum_{k=1}^{107} C_i$	29331	3187	298472	68347	104099	35752	68347	0

Table 14.2: Detailed experimental results for ∀-queries.

id	C_i	t_{db}	$t_{offline}$	t_{kaon2}	ext_{aqa}	ext_{kaon2}	*miss*	*corr*	*more*
1	∀madeFromGrape.nom8	256	8	2587	160	1	0	1	159
2	∀hasBody.nom45	512	9	1052	133	16	0	16	117
3	∀hasMaker.Winery	354	15	637	161	53	0	53	108
4	∀hasBody.WineBody	234	12	568	161	161	0	161	0
5	∀hasBody.nom49	422	9	1340	145	27	0	27	118
6	∀hasSugar.WineTaste	309	12	501	161	161	0	161	0
7	∀hasSugar.WineSugar	211	9	523	161	161	0	161	0
8	∀hasColor.nom9	233	9	494	161	161	0	161	0
9	∀hasSugar.nom1	432	8	927	124	6	0	6	118
10	∀madeFromGrape.nom29	369	8	22466	160	6	0	6	154
11	∀hasVintageYear.VintageYear	378	7	475	161	161	0	161	0
12	∀locatedIn.Region	322	17	560	161	161	0	161	0
13	∀hasFlavor.nom64	334	9	851	149	32	0	32	117
14	∀madeFromGrape.nom62	766	8	5534	160	5	0	5	155
15	∀madeFromGrape.WineGrape	379	9	748	161	161	0	161	0
16	∀hasSugar.nom7	233	9	1590	158	37	0	37	121
17	∀hasBody.nom59	298	9	51798	148	29	0	29	119
18	∀hasFlavor.nom58	677	8	676	130	15	0	15	115
19	∀hasBody.nom51	557	9	1849	158	42	0	42	116
20	∀hasColor.WineColor	401	8	1800	161	161	0	161	0
21	∀hasSugar.nom12	290	9	934	156	47	0	47	109
22	∀madeFromGrape.nom25	233	8	1550	160	2	0	2	158
23	∀madeFromGrape.nom3	144	8	1721	160	1	0	1	159
24	∀hasBody.nom56	156	39	1744	123	6	0	6	117
25	∀hasSugar.WineDescriptor	289	10	528	161	161	0	161	0
26	∀madeFromGrape.nom6	286	8	1486	161	7	0	7	154

Detailed experimental results for ∀-queries (continued).

id	C_i	t_{db}	$t_{offline}$	t_{kaon2}	ext_{aqa}	ext_{kaon2}	*miss*	*corr*	*more*
27	∀hasFlavor.WineTaste	135	24	517	161	161	0	161	0
28	∀hasColor.nom5	156	9	978	161	24	0	24	137
29	∀adjacentRegion.Region	113	9	538	161	161	0	161	0
30	∀hasBody.nom40	144	12	551	161	161	0	161	0
31	∀hasColor.nom32	122	8	28775	160	1	0	1	159
32	∀hasFlavor.nom50	188	9	676	156	40	0	40	116
33	∀hasSugar.nom54	293	8	1751	123	4	0	4	119
34	∀hasFlavor.nom55	231	8	1811	123	10	0	10	113
35	∀hasFlavor.WineFlavor	112	8	500	161	161	0	161	0
36	∀hasFlavor.nom35	134	8	516	161	161	0	161	0
37	∀madeFromGrape.nom60	104	9	1335	160	1	0	1	159
38	∀madeFromGrape.nom27	101	8	9749	160	2	0	2	158
39	∀hasColor.nom28	124	8	947	160	28	0	28	132
40	∀madeFromGrape.nom20	115	7	1214	160	5	0	5	155
41	∀madeFromGrape.nom21	221	8	878	160	1	0	1	159
42	∀hasFlavor.nom48	133	9	1714	144	28	0	28	116
43	∀madeFromGrape.nom24	122	8	1394	160	8	0	8	152
44	∀madeFromGrape.nom11	154	7	900	160	5	0	5	155
45	∀madeFromGrape.nom39	178	8	4545	160	4	0	4	156
46	∀madeFromGrape.nom33	143	8	1513	160	1	0	1	159
47	∀madeFromGrape.nom63	154	8	621	160	1	0	1	159
48	∀madeFromGrape.nom43	102	7	932	160	3	0	3	157
49	∀madeFromGrape.Grape	192	8	492	161	161	0	161	0
50	∀hasSugar.nom53	133	8	781	126	6	0	6	120
51	∀madeFromGrape.nom37	155	7	1113	160	4	0	4	156
52	∀madeFromGrape.nom34	214	8	747	160	2	0	2	158

Detailed experimental results for ∀-queries (continued).

id	C_i	t_{db}	$t_{offline}$	t_{kaon2}	ext_{aqa}	ext_{kaon2}	$miss$	$corr$	$more$
53	∀hasColor.WineDescriptor	139	8	522	161	161	0	161	0
54	∀hasFlavor.WineDescriptor	154	9	593	161	161	0	161	0
55	∀hasWineDescriptor.WineDescriptor	117	9	491	161	161	0	161	0
56	∀hasBody.WineTaste	146	8	524	161	161	0	161	0
57	∀hasBody.WineDescriptor	243	8	544	161	161	0	161	0
	$\sum_{k=1}^{57} C_i$	13847	545	173088	8839	3730	0	3730	5109

Table 14.3: Detailed experimental results for $\sqcup$-queries.

id	C_i	t_{db}	$t_{offline}$	t_{kaon2}	ext_{aqa}	ext_{kaon2}	*miss*	*corr*	*more*
1	nom46 $\sqcup$ nom53	53	10	3032	123	123	0	123	0
2	nom6 $\sqcup$ PinotNoir	76	9	2556	246	246	0	246	0
3	WineBody $\sqcup$ Chianti	83	5	3252	164	164	0	164	0
4	Wine $\sqcup$ ItalianWine	302	53	2996	2173	2173	0	2173	0
5	nom33 $\sqcup$ nom53	88	3	2677	123	123	0	123	0
6	nom2 $\sqcup$ RedWine	161	19	3114	1107	1107	0	1107	0
7	WineTaste $\sqcup$ CotesDOr	132	7	3201	410	410	0	410	0
8	PinotBlanc $\sqcup$ Vintage	50	4	2346	82	82	0	82	0
9	nom36 $\sqcup$ Margaux	56	2	3066	82	82	0	82	0
10	PotableLiquid $\sqcup$ RedBurgundy	324	65	3092	2173	2173	0	2173	0
11	Medoc $\sqcup$ nom26	107	3	2444	123	123	0	123	0
12	Riesling $\sqcup$ FrenchWine	130	3	3162	164	164	0	164	0
13	nom38 $\sqcup$ nom36	78	5	2962	82	82	0	82	0
14	nom54 $\sqcup$ nom30	53	3	2465	82	82	0	82	0
15	nom38 $\sqcup$ CotesDOr	64	3	3145	82	82	0	82	0
16	TexasWine $\sqcup$ RoseWine	57	2	3000	82	82	0	82	0
17	nom7 $\sqcup$ Chardonnay	89	5	2514	410	410	0	410	0
18	nom25 $\sqcup$ WhiteNonSweetWine	149	9	3086	779	779	0	779	0
19	nom54 $\sqcup$ FrenchWine	56	2	3137	41	41	0	41	0
20	Sauternes $\sqcup$ WhiteBordeaux	55	2	2630	41	41	0	41	0
21	SauvignonBlanc $\sqcup$ nom27	69	3	3186	205	205	0	205	0
22	WineFlavor $\sqcup$ CheninBlanc	66	3	3039	205	205	0	205	0
23	EarlyHarvest $\sqcup$ Margaux	52	2	3309	41	41	0	41	0
24	nom8 $\sqcup$ Meritage	65	3	2576	246	246	0	246	0
25	WhiteWine $\sqcup$ SemillonOrSauvignonBlanc	148	81	3260	984	984	0	984	0
26	WineColor $\sqcup$ Medoc	90	2	3196	205	205	0	205	0

Detailed experimental results for ⊔-queries (continued).

id	C_i	t_{db}	$t_{offline}$	t_{kaon2}	ext_{aqa}	ext_{kaon2}	*miss*	*corr*	*more*
27	nom58 ⊔ Meritage	70	2	2548	82	82	0	82	0
28	PotableLiquid ⊔ nom50	359	8	3169	2255	2255	0	2255	0
29	Bordeaux ⊔ GermanWine	92	2	3068	246	246	0	246	0
30	LateHarvest ⊔ nom37	73	1	3262	123	123	0	123	0
31	nom62 ⊔ Merlot	70	2	2566	205	205	0	205	0
32	nom18 ⊔ nom9	65	1	3349	246	246	0	246	0
33	nom41 ⊔ Gamay	49	2	2992	82	82	0	82	0
34	nom48 ⊔ WineFlavor	61	2	3206	123	123	0	123	0
35	nom38 ⊔ WhiteLoire	70	1	2491	82	82	0	82	0
36	Chardonnay ⊔ nom14	107	3	3372	369	369	0	369	0
37	DryRedWine ⊔ RedBordeaux	195	6	3097	1025	1025	0	1025	0
38	nom7 ⊔ nom3	75	2	3413	123	123	0	123	0
39	DessertWine ⊔ Zinfandel	103	3	2447	369	369	0	369	0
40	nom29 ⊔ CaliforniaWine	201	4	3318	984	984	0	984	0
41	nom41 ⊔ FullBodiedWine	253	15	3077	697	697	0	697	0
42	Port ⊔ Margaux	92	1	3341	82	82	0	82	0
43	nom19 ⊔ CheninBlanc	101	1	2530	123	123	0	123	0
44	nom43 ⊔ nom27	97	2	3243	82	82	0	82	0
45	WineTaste ⊔ nom45	78	2	3238	369	369	0	369	0
46	nom9 ⊔ DryRiesling	60	2	3521	164	164	0	164	0
47	Loire ⊔ Chardonnay	117	2	2609	451	451	0	451	0
48	GermanWine ⊔ nom20	77	2	3403	123	123	0	123	0
49	Semillon ⊔ Burgundy	122	2	3189	246	246	0	246	0
50	nom48 ⊔ nom30	73	2	3100	82	82	0	82	0
51	nom22 ⊔ WhiteNonSweetWine	168	4	2606	779	779	0	779	0
52	nom63 ⊔ nom43	86	1	3342	164	164	0	164	0

Detailed experimental results for $\sqcup$-queries (continued).

id	C_i	t_{db}	$t_{offline}$	t_{kaon2}	ext_{aqa}	ext_{kaon2}	*miss*	*corr*	*more*
53	nom45 $\sqcup$ nom47	96	1	3152	82	82	0	82	0
54	DryWhiteWine $\sqcup$ nom52	213	3	3346	779	779	0	779	0
55	WineColor $\sqcup$ Chianti	109	1	3330	164	164	0	164	0
56	nom59 $\sqcup$ nom14	110	1	3337	123	123	0	123	0
57	FrenchWine $\sqcup$ RedBordeaux	104	2	2563	123	123	0	123	0
58	Merlot $\sqcup$ nom21	108	4	3264	164	164	0	164	0
59	nom19 $\sqcup$ Sauternes	87	3	3109	82	82	0	82	0
60	Wine $\sqcup$ nom64	319	11	3247	2255	2255	0	2255	0
61	DryRedWine $\sqcup$ PinotBlanc	256	5	3313	1066	1066	0	1066	0
62	nom5 $\sqcup$ GermanWine	91	4	2640	123	123	0	123	0
63	WineBody $\sqcup$ nom53	64	3	3418	205	205	0	205	0
64	DryWine $\sqcup$ WineTaste	325	11	3393	2132	2132	0	2132	0
65	nom31 $\sqcup$ RedBordeaux	116	3	3116	164	164	0	164	0
66	nom62 $\sqcup$ AmericanWine	283	4	3438	1066	1066	0	1066	0
67	nom51 $\sqcup$ nom60	140	1	3366	123	123	0	123	0
68	nom10 $\sqcup$ nom34	52	3	2565	82	82	0	82	0
69	RedBurgundy $\sqcup$ nom1	70	1	3287	82	82	0	82	0
70	nom38 $\sqcup$ RedBurgundy	92	1	2987	82	82	0	82	0
71	nom29 $\sqcup$ Region	182	8	3256	1517	1517	0	1517	0
72	WhiteWine $\sqcup$ WhiteBurgundy	223	6	3263	984	984	0	984	0
73	nom45 $\sqcup$ nom26	55	3	3277	82	82	0	82	0
74	nom7 $\sqcup$ nom32	60	3	2680	123	123	0	123	0
75	nom33 $\sqcup$ Vintage	68	3	3135	82	82	0	82	0
76	nom31 $\sqcup$ ItalianWine	71	3	3373	82	82	0	82	0
77	Sancerre $\sqcup$ Port	74	1	3399	82	82	0	82	0
78	DryRiesling $\sqcup$ Margaux	49	2	3310	82	82	0	82	0

Detailed experimental results for $\sqcup$-queries (continued).

id	C_i	t_{db}	$t_{offline}$	t_{kaon2}	ext_{aqa}	ext_{kaon2}	*miss*	*corr*	*more*
79	nom47 $\sqcup$ AmericanWine	151	6	3405	1025	1025	0	1025	0
80	DryWhiteWine $\sqcup$ WhiteWine	164	11	3418	984	984	0	984	0
81	Merlot $\sqcup$ nom50	85	3	3429	205	205	0	205	0
82	nom49 $\sqcup$ WhiteTableWine	162	4	3317	779	779	0	779	0
83	WineBody $\sqcup$ WhiteLoire	78	1	3212	164	164	0	164	0
84	WineColor $\sqcup$ TableWine	272	7	3380	1886	1886	0	1886	0
85	FullBodiedWine $\sqcup$ nom56	187	5	3240	697	697	0	697	0
86	nom28 $\sqcup$ nom44	53	2	3319	82	82	0	82	0
87	nom30 $\sqcup$ nom37	70	1	3368	82	82	0	82	0
88	nom23 $\sqcup$ Muscadet	71	1	3434	82	82	0	82	0
89	SweetRiesling $\sqcup$ nom58	73	1	3657	123	123	0	123	0
90	nom4 $\sqcup$ Meritage	88	2	3158	82	82	0	82	0
91	nom62 $\sqcup$ nom10	58	1	3301	123	123	0	123	0
92	RedWine $\sqcup$ nom11	208	4	3386	1107	1107	0	1107	0
93	DessertWine $\sqcup$ WhiteNonSweetWine	200	5	3195	902	902	0	902	0
94	nom40 $\sqcup$ Medoc	88	3	3494	205	205	0	205	0
95	EarlyHarvest $\sqcup$ CaliforniaWine	175	3	3409	943	943	0	943	0
96	Merlot $\sqcup$ nom1	79	1	3493	164	164	0	164	0
97	WhiteWine $\sqcup$ nom11	203	5	3363	1025	1025	0	1025	0
98	nom62 $\sqcup$ nom17	76	9	3392	123	123	0	123	0
99	nom62 $\sqcup$ RoseWine	82	2	3429	123	123	0	123	0
100	Margaux $\sqcup$ Meritage	74	1	3374	82	82	0	82	0
	$\sum_{k=1}^{100} C_i$	11681	552	312677	43050	43050	0	43050	0

Table 14.4: Detailed experimental results for $\sqcap$-queries.

id	C_i	t_{db}	$t_{offline}$	t_{kaon2}	ext_{aqa}	ext_{kaon2}	*miss*	*corr*	*more*
1	nom46 $\sqcap$ nom53	41	289	3319	0	0	0	0	0
2	nom6 $\sqcap$ PinotNoir	50	48	2813	0	0	0	0	0
3	WineBody $\sqcap$ Chianti	48	40	2397	0	0	0	0	0
4	Wine $\sqcap$ ItalianWine	77	88	3072	41	41	0	41	0
5	nom33 $\sqcap$ nom53	41	23	2351	0	0	0	0	0
6	nom2 $\sqcap$ RedWine	55	51	2858	0	0	0	0	0
7	WineTaste $\sqcap$ CotesDOr	55	45	2982	0	0	0	0	0
8	PinotBlanc $\sqcap$ Vintage	51	40	2151	0	0	0	0	0
9	nom36 $\sqcap$ Margaux	47	37	2915	0	0	0	0	0
10	PotableLiquid $\sqcap$ RedBurgundy	80	77	2435	41	41	0	41	0
11	Medoc $\sqcap$ nom26	51	41	2872	0	0	0	0	0
12	Riesling $\sqcap$ FrenchWine	50	37	2703	0	0	0	0	0
13	nom38 $\sqcap$ nom36	52	20	2253	0	0	0	0	0
14	nom54 $\sqcap$ nom30	58	36	3054	0	0	0	0	0
15	nom38 $\sqcap$ CotesDOr	40	2	2418	0	0	0	0	0
16	TexasWine $\sqcap$ RoseWine	41	37	2970	0	0	0	0	0
17	nom7 $\sqcap$ Chardonnay	49	38	2422	0	0	0	0	0
18	nom25 $\sqcap$ WhiteNonSweetWine	47	43	3002	0	0	0	0	0
19	nom54 $\sqcap$ FrenchWine	39	2	2880	0	0	0	0	0
20	Sauternes $\sqcap$ WhiteBordeaux	46	37	2395	41	41	0	41	0
21	SauvignonBlanc $\sqcap$ nom27	43	44	3117	0	0	0	0	0
22	WineFlavor $\sqcap$ CheninBlanc	44	37	2345	0	0	0	0	0
23	EarlyHarvest $\sqcap$ Margaux	40	20	2930	0	0	0	0	0
24	nom8 $\sqcap$ Meritage	42	41	2886	0	0	0	0	0
25	WhiteWine $\sqcap$ SemillonOrSauvignonBlanc	118	112	2400	287	287	0	287	0
26	WineColor $\sqcap$ Medoc	45	23	2979	0	0	0	0	0

Detailed experimental results for ⊓-queries. (continued).

id	C_i	t_{db}	$t_{offline}$	t_{kaon2}	ext_{aqa}	ext_{kaon2}	*miss*	*corr*	*more*
27	nom58 ⊓ Meritage	42	20	2295	0	0	0	0	0
28	PotableLiquid ⊓ nom50	93	27	2972	0	0	0	0	0
29	Bordeaux ⊓ GermanWine	45	38	2806	0	0	0	0	0
30	LateHarvest ⊓ nom37	42	37	2502	0	0	0	0	0
31	nom62 ⊓ Merlot	48	38	3026	0	0	0	0	0
32	nom18 ⊓ nom9	44	35	2969	0	0	0	0	0
33	nom41 ⊓ Gamay	39	37	2318	0	0	0	0	0
34	nom48 ⊓ WineFlavor	63	23	2970	41	41	0	41	0
35	nom38 ⊓ WhiteLoire	49	20	2275	0	0	0	0	0
36	Chardonnay ⊓ nom14	53	52	2897	0	0	0	0	0
37	DryRedWine ⊓ RedBordeaux	97	60	2919	123	123	0	123	0
38	nom7 ⊓ nom3	49	20	2358	0	0	0	0	0
39	DessertWine ⊓ Zinfandel	59	37	2956	0	0	0	0	0
40	nom29 ⊓ CaliforniaWine	63	40	3059	0	0	0	0	0
41	nom41 ⊓ FullBodiedWine	58	23	2379	0	0	0	0	0
42	Port ⊓ Margaux	50	17	3016	0	0	0	0	0
43	nom19 ⊓ CheninBlanc	54	19	2320	0	0	0	0	0
44	nom43 ⊓ nom27	43	21	2880	0	0	0	0	0
45	WineTaste ⊓ nom45	50	52	2981	41	41	0	41	0
46	nom9 ⊓ DryRiesling	41	20	2481	0	0	0	0	0
47	Loire ⊓ Chardonnay	51	21	3295	0	0	0	0	0
48	GermanWine ⊓ nom20	40	25	2471	0	0	0	0	0
49	Semillon ⊓ Burgundy	43	40	3101	0	0	0	0	0
50	nom48 ⊓ nom30	42	1	2894	0	0	0	0	0
51	nom22 ⊓ WhiteNonSweetWine	55	20	2474	0	0	0	0	0
52	nom63 ⊓ nom43	52	22	3005	0	0	0	0	0

Detailed experimental results for ⊓-queries. (continued).

id	C_i	t_{db}	$t_{offline}$	t_{kaon2}	ext_{aqa}	ext_{kaon2}	$miss$	$corr$	$more$
53	nom45 ⊓ nom47	58	18	2855	0	0	0	0	0
54	DryWhiteWine ⊓ nom52	72	41	2357	0	0	0	0	0
55	WineColor ⊓ Chianti	49	1	3045	0	0	0	0	0
56	nom59 ⊓ nom14	39	19	2873	0	0	0	0	0
57	FrenchWine ⊓ RedBordeaux	52	1	2476	0	0	0	0	0
58	Merlot ⊓ nom21	41	17	3037	0	0	0	0	0
59	nom19 ⊓ Sauternes	41	1	3134	0	0	0	0	0
60	Wine ⊓ nom64	90	41	2490	0	0	0	0	0
61	DryRedWine ⊓ PinotBlanc	55	7	3113	0	0	0	0	0
62	nom5 ⊓ GermanWine	41	19	2993	0	0	0	0	0
63	WineBody ⊓ nom53	40	2	2353	0	0	0	0	0
64	DryWine ⊓ WineTaste	165	25	3228	0	0	0	0	0
65	nom31 ⊓ RedBordeaux	42	20	2820	0	0	0	0	0
66	nom62 ⊓ AmericanWine	70	21	2461	0	0	0	0	0
67	nom51 ⊓ nom60	49	40	3040	0	0	0	0	0
68	nom10 ⊓ nom34	47	35	2826	0	0	0	0	0
69	RedBurgundy ⊓ nom1	49	20	2410	0	0	0	0	0
70	nom38 ⊓ RedBurgundy	52	1	3150	0	0	0	0	0
71	nom29 ⊓ Region	72	22	2909	0	0	0	0	0
72	WhiteWine ⊓ WhiteBurgundy	90	21	2406	123	123	0	123	0
73	nom45 ⊓ nom26	41	1	3164	0	0	0	0	0
74	nom7 ⊓ nom32	39	19	2870	0	0	0	0	0
75	nom33 ⊓ Vintage	39	2	2445	0	0	0	0	0
76	nom31 ⊓ ItalianWine	42	1	3095	0	0	0	0	0
77	Sancerre ⊓ Port	40	49	3098	0	0	0	0	0
78	DryRiesling ⊓ Margaux	40	1	2501	0	0	0	0	0

Detailed experimental results for $\sqcap$-queries. (continued).

id	C_i	t_{db}	$t_{offline}$	t_{kaon2}	ext_{aqa}	ext_{kaon2}	*miss*	*corr*	*more*
79	nom47 $\sqcap$ AmericanWine	52	4	3028	0	0	0	0	0
80	DryWhiteWine $\sqcap$ WhiteWine	255	6	3003	738	738	0	738	0
81	Merlot $\sqcap$ nom50	56	1	2505	0	0	0	0	0
82	nom49 $\sqcap$ WhiteTableWine	47	38	2901	0	0	0	0	0
83	WineBody $\sqcap$ WhiteLoire	41	1	3040	0	0	0	0	0
84	WineColor $\sqcap$ TableWine	81	22	2535	0	0	0	0	0
85	FullBodiedWine $\sqcap$ nom56	47	23	3038	0	0	0	0	0
86	nom28 $\sqcap$ nom44	46	36	2963	0	0	0	0	0
87	nom30 $\sqcap$ nom37	40	1	2472	0	0	0	0	0
88	nom23 $\sqcap$ Muscadet	39	37	3177	0	0	0	0	0
89	SweetRiesling $\sqcap$ nom58	42	20	2988	0	0	0	0	0
90	nom4 $\sqcap$ Meritage	39	19	2578	0	0	0	0	0
91	nom62 $\sqcap$ nom10	42	1	3044	0	0	0	0	0
92	RedWine $\sqcap$ nom11	51	23	2957	0	0	0	0	0
93	DessertWine $\sqcap$ WhiteNonSweetWine	64	4	2509	0	0	0	0	0
94	nom40 $\sqcap$ Medoc	43	19	3040	0	0	0	0	0
95	EarlyHarvest $\sqcap$ CaliforniaWine	44	3	2993	0	0	0	0	0
96	Merlot $\sqcap$ nom1	42	1	2489	0	0	0	0	0
97	WhiteWine $\sqcap$ nom11	59	4	3111	0	0	0	0	0
98	nom62 $\sqcap$ nom17	42	18	3164	0	0	0	0	0
99	nom62 $\sqcap$ RoseWine	47	1	2411	0	0	0	0	0
100	Margaux $\sqcap$ Meritage	43	1	3172	0	0	0	0	0
	$\sum_{k=1}^{100} C_i$	5432	2801	278805	1476	1476	0	1476	0

References

[ABD92] A. R. Anderson, N. D. Belnap Jr., and J. M. Dunn. *Entailment – the Logic of Relevance and Necessity*, volume 2. Princeton University Press, 1992.

[ABW88] K. R. Apt, H. A. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, Los Altos, CA, 1988.

[AKPS94] H. Ait-Kaci, A. Podelski, and G. Smolka. A Feature Constraint System for Logic Programming with Entailment. *Theoretical Computer Science*, 122(1&2):263–283, 1994.

[Ari08] Aristotle. Metaphysics. In W. D. Ross, editor, *The Works of Aristotle translated into English, Volume VIII*. Oxford University Press, Oxford, UK, 1908.

[BBL05] F. Baader, S. Brandt, and C. Lutz. Pushing the $\mathcal{EL}$ Envelope. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK*, pages 364–369. Professional Book Center, 2005.

[BCM$^+$03] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook*. Cambridge University Press, January 2003.

[BD89] M. S. Boddy and T. Dean. Solving Time-Dependent Planning Problems. In N. S. Sridharan, editor, *IJCAI*, pages 979–984, Detroit, Michigan, USA, August 1989. Morgan Kaufmann Publisher.

[BG01] L. Bachmair and H. Ganzinger. Resolution Theorem Proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 2, pages 19–99. Elsevier Science, 2001.

[BG04] D. Brickley and R.V. Guha. RDF Vocabulary Description Language – RDF Schema. `http://www.w3.org/TR/rdf-schema/`, 2004.

[BL84] R. J. Brachman and H. J. Levesque. The Tractability of Subsumption in Frame-Based Description Languages. In Ronald J. Brachman, editor, *Proceedings of the National Conference on Artificial Intelligence (AAAI'84)*, pages 34–37, Austin, USA, August 6–10 1984. AAAI Press.

[BLHL01] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, May 2001.

[BN03] F. Baader and W. Nutt. Basic Description Logics. In F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation, and Applications*, pages 43–95. Cambridge University Press, 2003.

[Bor06] W. N. Borst. *Construction of Engineering Ontologies for Knowledge Sharing and Reuse*. PhD thesis, University of Enschede, Enschede, The Netherlands, 2006.

[BTX$^+$09a] J. Bock, T. Tserendorj, Y. Xu, J. Wissmann, and S. Grimm. A Reasoning Broker Framework for OWL. In P. F. Patel-Schneider R. Hoekstra, editor, *Proceedings of the 6th International Workshop on OWL: Experiences and Directions (OWLED 2009)*, CEUR Workshop Proceedings, Chantilly, VA, United States, October 23–24 2009.

[BTX$^+$09b] J. Bock, T. Tserendorj, Y. Xu, J. Wissmann, and S. Grimm. A Reasoning Broker Framework for Protégé. 11th International Protégé Conference, Amsterdam, June 23–26 2009.

[BVL03] S. Bechhofer, R. Volz, and P. Lord. Cooking the Semantic Web with the OWL API. In *Proceedings of the 2nd International Semantic Web Conference (ISWC'03)*, LNCS, pages 659–675, Sanibel Island, Florida, USA, October 20–23 2003. Springer.

[BYRN99] R. A. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Cad95] M. Cadoli. *Tractable Reasoning in Artificial Intelligence*. Springer, Secaucus, NJ, USA, 1995.

[CD97] M. Cadoli and F. M. Donini. A survey on knowledge compilation. *AI Communications*, 10(3,4):137–150, 1997.

[CDL$^+$07a] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. EQL-Lite: Effective First-Order Query Processing in Description Logics. In Manuela M. Veloso, editor, *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 274–279, Hyderabad, India, January 6-12 2007. Morgan Kaufmann Publishers.

[CDL^{+}07b] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tractable Reasoning and Efficient Query Answering in Description Logics: The DL-Lite Family. *Journal of Automated Reasoning*, 39(3):385–429, 2007.

[CFGL04] C. Cumbo, W. Faber, G. Greco, and N. Leone. Enhancing the Magic-Set Method for Disjunctive Datalog Programs. In B. Demoen and V. Lifschitz, editors, *Proceedings of the 20th International Conference on Logic Programming (ICLP 2004)*, volume 3132 of *LNCS*, pages 371–385, Saint-Malo, France, September 6–10 2004. Springer.

[CJB99] B. Chandrasekaran, J. R. Josephson, and V. R. Benjamins. What Are Ontologies, and Why Do We Need Them? *IEEE Intelligent Systems*, 14(1):20–26, 1999.

[Cod83] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Journal of the ACM*, 26(1):64–69, 1983.

[CS96] M. Cadoli and M. Schaerf. On the complexity of Entailment in Propositional Multivalued Logics. *Annals of Mathematics and Artificial Intelligence*, 18(1):29–50, 1996.

[DB88] T. Dean and M. S. Boddy. An Analysis of Time-Dependent Planning. In T. M. Mitchell and Reid G. Smith, editors, *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 49–54. AAAI Press, 1988.

[DM02] A. Darwiche and P. Marquis. A Knowledge Compilation Map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

[DP01] N. Dershowitz and D. A. Plaisted. Rewriting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 9, pages 535–610. Elsevier Science, 2001.

[EGM97] T. Eiter, G. Gottlob, and H. Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–418, 1997.

[Fen03] D. Fensel. *Ontologies: : A Silver Bullet for Knowledge Management and Electronic Commerce*. Springer, December 2003.

[FH07] D. Fensel and F. Van Harmelen. Unifying Reasoning and Search to Web Scale. *IEEE Internet Computing*, 11(2):94–96, March/April 2007.

[Flo03] L. Floridi, editor. *Blackwell Guide to the Philosophy of Computing and Information*. Blackwell Publishers, Inc., Cambridge, MA, USA, 2003.

[FW02] M. Finger and R. Wassermann. Logics for Approximate Reasoning: Approximating Classical Logic From Above. In G. Ramalho and G. Bittencourt, editors, *SBIA '02: Proceedings of the 16th Brazilian Symposium on*

Artificial Intelligence, pages 21–30, London, UK, November 11–14 2002. Springer.

[FW06] M. Finger and R. Wassermann. The universe of propositional approximations. *Theorectical Computer Science*, 355(2):153–166, 2006.

[Gam08] E. Gamma. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, 2008.

[GHKS07] B. C. Grau, I. Horrocks, Y. Kazakov, and U. Sattler. Just the Right Amount: Extracting Modules from Ontologies. In *Proceedings of the 16th International Conference on World Wide Web (WWW)*, pages 717–726, New York, NY, USA, May 8–12 2007. ACM Press.

[GHLS07] B. Glimm, I. Horrocks, C. Lutz, and U. Sattler. Conjunctive Query Answering for the Description Logic SHIQ. In M. M. Veloso, editor, *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 157–204, Hyderabad, India, January 6–12 2007. Morgan Kaufmann Publishers.

[GHM$^+$08] B. C. Grau, I. Horrocks, B. Motik, B. Parsia, P. F. Patel-Schneider, and U. Sattler. OWL 2: The next step for OWL. *Web Semantics.*, 6(4):309–322, 2008.

[GHT06] T. Gardiner, I. Horrocks, and D. Tsarkov. Automated Benchmarking of Description Logic Reasoners. In B. Parsia, U. Sattler, and D. Toman, editors, *Proceedings of the 2006 International Workshop on Description Logics (DL 2006)*, volume 189 of *CEUR Workshop Proceedings*, Windermere,UK, May 30–June 1 2006.

[GHVD03] B. N. Grosof, I. Horrocks, R. Volz, and S. Decker. Description Logic Programs: Combining Logic Programs with Description Logic. In *Proceedings of the 12th International World Wide Web Conference (WWW 2003)*, pages 48–57, Budapest, Hungary, May 20–24 2003. ACM Press.

[GQPH07] Y. Guo, A. Qasem, Z. Pan, and J. Heflin. A Requirements Driven Framework for Benchmarking Semantic Web Knowledge Base Systems. *IEEE Transactions on Knowledge and Data Engineering*, 19:297–309, 2007.

[Gru93] T. R. Gruber. A translation approach to portable ontology specifications. *Journal of Knowledge Acquisition*, 5(2):199–220, June 1993.

[GSW05] P. Groot, H. Stuckenschmidt, and H. Wache. Approximating Description Logic Classification for Semantic Web Reasoning. In A. Gómez-Pérez and J. Euzenat, editors, *Proceedings of the 2rd European Semantic Web Conference (ESWC'05)*, volume 3532 of *Lecture Notes in Computer Science*, pages 318–332, Crete, Greece, May 29 – June 1 2005. Springer.

[GTH06] T. Gardiner, D. Tsarkov, and I. Horrocks. Framework For an Automated Comparison of Description Logic Reasoners. In *Proceedings of the 5th International Semantic Web Conference (ISWC 2006)*, volume 4273 of *Lecture Notes in Computer Science*, pages 654–667. Springer, November 5–9 2006.

[Gtv04] P. Groot, A. ten Teije, and F. van Harmelen. Towards a Structured Analysis of Approximate Problem Solving: A Case Study in Classification. In D. Dubois, C. A. Welty, and M. Williams, editors, *Proceedings of the Ninth International Conference on the Principles of Knowledge Representation and Reasoning*, pages 399–406, Whistler, British Columbia, Canada, June 2–5 2004. AAAI Press.

[Gua97] N. Guarino. Semantic Matching: Formal Ontological Distinctions for Information Organization, Extraction, and Integration. In *SCIE '97: International Summer School on Information Extraction*, pages 139–170, London, UK, June 16–August 8 1997. Springer.

[Gua98] N. Guarino. Formal Ontologies and Information Systems, Preface. In *Proceedings of the 1st Conference on Formal Ontologies and Information Systems (FOIS'98)*, pages 3–15, Trento, Italy, June 6–8 1998. IOS Press.

[GZ96] Joshua Grass and Shlomo Zilberstein. Anytime algorithm development tools. *SIGART Bulletin Special Issue on Anytime Algorithms and Deliberation Scheduling*, 7(2):20–27, 1996.

[Hay04] P. Hayes. RDF Semantics. `http://www.w3.org/TR/rdf-mt/`, 10 February 2004.

[HBN07] M. Horridge, S. Bechhofer, and O. Noppens. Igniting the OWL 1.1 Touch Paper: The OWL API. In C. Golbreich, A. Kalyanpur, and B. Parsia, editors, *OWLED*, volume 258 of *CEUR Workshop Proceedings*, June 6–7 2007.

[HDG$^+$06] M. Horridge, N. Drummond, J. Goodwin, A. L. Rector, R. Stevens, and H. Wang. The Manchester OWL Syntax. In B. C. Grau, P. Hitzler, C. Shankey, and E. Wallace, editors, *Proceedings of the OWLED 06 Workshop on OWL: Experiences and Directions*, volume 216 of *CEUR Workshop Proceedings*, Athens, Georgia, USA, November 10-11 2006.

[Hef01] J. Heflin. *Towards the Semantic Web: Knowledge Representation in a Dynamic, Distributed Environment*. PhD thesis, University of Maryland, College Park, 2001.

[Hep08] M. Hepp. Ontologies: State of the Art, Business Potential, and Grand Challenges. In M. Hepp, P. De Leenheer, A. de Moor, and Y. Sure, editors, *Ontology Management*, volume 7 of *Semantic Web And Beyond Computing for Human Experience*, pages 3–22. Springer, 2008.

[HEPS03] M. Hori, J. Euzenat, and P. F. Patel-Schneider. OWL Web Ontology Language: XML Presentation Syntax, W3C Note. `http://www.w3.org/TR/owl-xmlsyntax/`, June 11 2003.

[HM01a] V. Haarslev and R. Möller. Description of the RACER System and its Applications. In C. A. Goble, D. L. McGuinness, R. Möller, and P. F. Patel-Schneider, editors, *Proceedings of the 2001 International Workshop on Description Logics (DL 2001)*, volume 49 of *CEUR Workshop Proceedings*, Stanford, CA, USA, August 1–3 2001.

[HM01b] V. Haarslev and R. Möller. RACER System Description. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proceedings of the 1st International Joint Conference on Automated Reasoning (IJCAR 2001)*, volume 2083 of *LNAI*, pages 701–706, Siena, Italy, June 18–23 2001. Springer.

[HMS$^+$04] V. Haarslev, R. Möller, R. Van Der Straeten, R. Van, Der Straeten, and M. Wessel. Extended Query Facilities for Racer and an Application to Software-Engineering Problems. In Volker Haarslev and Ralf Möller, editors, *Proceedings of the 2004 International Workshop on Description Logics (DL 2004)*, volume 104 of *CEUR Workshop Proceedings*, Whistler, British Columbia, Canada, June 6–8 2004.

[HMS05] U. Hustadt, B. Motik, and U. Sattler. Data Complexity of Reasoning in Very Expressive Description Logics. In L. P. Kaelbling and A. Saffiotti, editors, *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI 2005)*, pages 466–471, Edinburgh, UK, July 30–August 5 2005. Morgan Kaufmann Publishers.

[Hor05] I. Horrocks. Applications of Description Logics: State of the Art and Research Challenges. In F. Dau, M. Mugnier, and G. Stumme, editors, *Proceedings of the 13th International Conference on Conceptual Structures (ICCS'05)*, volume 3596 of *Lecture Notes in Artificial Intelligence*, pages 78–90. Springer, May 22–25 2005.

[HS01] I. Horrocks and U. Sattler. Ontology Reasoning in the $\mathcal{SHOQ}(\mathbf{D})$ Description Logic. In B. Nebel, editor, *Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI 2001)*, pages 199–204, Seattle, WA, USA, August 4–10 2001. Morgan Kaufmann Publishers.

[HV05] P. Hitzler and D. Vrandecic. Resolution-Based Approximate Reasoning for OWL DL. In Y. Gil, E. Motta, V. R. Benjamins, and M. A. Musen, editors, *Proceedings of the 4th International Semantic Web Conference*, volume 3729 of *Lecture Notes in Computer Science*, pages 383–397, Galway, Ireland, November 7–11 2005. Springer, Berlin.

[KC04] G. Klyne and J. J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. `http://www.w3.org/TR/rdf-concepts/`, 10 February 2004.

[KLW95] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of the ACM*, 42:741–843, July 1995.

[KOM05] A. Kiryakov, D. Ognyanov, and D. Manov. OWLIM – A Pragmatic Semantic Repository for OWL. In Y. Gil, editor, *Web Information Systems Engineering – WISE 2005 Workshops*, volume 3807 of *Lecture Notes in Computer Science*, pages 182–192, Cologne, Germany, September 2 2005. Springer.

[Kor01] F. Koriche. A Logic for Approximate First-Order Reasoning. In L. Fribourg, editor, *Computer Science Logic, 15th International Workshop, CSL Proceedings*, pages 262–276, Paris, France, September 10–13 2001. Springer.

[KS04] A. Kaya and K. Selzer. Design and Implementation of a Benchmark Testing Infrastructure for the DL System Racer. In S. Bechhofer, V. Haarslev, C. Lutz, and R. Möller, editors, *KI-04 Workshop on Applications of Description Logics (ADL-2004)*, pages 40–50, Ulm, Germany, September 24 2004. CEUR Workshop Proceedings.

[Kur05] D. Kuropka. Uselessness of simple co-occurrence measures for IF & IR – a linguistic point of view. In R. J. Brachman, editor, *Proceedings of the 8th International Conference on Business Information Systems*, pages 198–202, Poznan, Poland, April 20–22 2005.

[Lev84] H. J. Levesque. A logic of implicit and explicit belief. In R. J. Brachman, editor, *Proceedings of the Fourth National Conference on Artificial Intelligence (AAAI 1984)*, pages 198–202, Austin, Texas, USA, August 6–10 1984. AAAI Press.

[Llo87] J. W. Lloyd. *Foundations of logic programming, 2nd extended edition*. Springer, New York, 1987.

[LTHB04] L. Li, D. Turi, I. Horrocks, and S. K. Bechhofer. The Instance Store: DL reasoning with large numbers of individuals. In V. Haarslev and R. Möller, editors, *Proceedings of the 2004 Description Logic Workshop (DL 2004)*, volume 104, pages 31–40, Whistler, BC, Canada, June 6-8 2004. CEUR Workshop Proceedings.

[Mas98] F. Massacci. Anytime Approximate Modal Reasoning. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference*, pages 274–279, Madison, Wisconsin, USA, July 26–30 1998. AAAI Press / MIT Press.

[MBG$^+$02] C. Masolo, S. Borgo, A. Gangemi, N. Guarino, A. Oltramari, and L. Schneider. The WonderWeb Library of Foundational Ontologies: Preliminary Report. WonderWeb Deliverable D17, ISTC-CNR, Padova, Italy, August 2002.

[McB01] B. McBride. Jena: Implementing the RDF Model and Syntax Specification. In S. Decker, D. Fensel, A. Sheth, and S. Staab, editors, *Semantic Web Workshop*, volume 40 of *CEUR Workshop Proceedings*, Hongkong, China, May 1 2001.

[MFHS02] D. L. McGuinness, R. Fikes, J. Hendler, and L. A. Stein. DAML+OIL: An Ontology Language for the Semantic Web. *IEEE Intelligent Systems*, 17(5):72–80, 2002.

[MM04] F. Manola and E. Miller. RDF Primer. `http://www.w3.org/TR/rdf-primer/`, 10 February 2004.

[Mot06] B. Motik. *Reasoning in Description Logics using Resolution and Deductive Databases*. PhD thesis, Universität Karlsruhe, 2006.

[MPSG09] B. Motik, P. F. Patel-Schneider, and B. C. Grau, editors. *OWL 2 Web Ontology Language: Direct Semantics*. W3C Recommendation, 27 October 2009. Available at `http://www.w3.org/TR/owl2-direct-semantics/`.

[MS06] B. Motik and U. Sattler. A Comparison of Reasoning Techniques for Querying Large Description Logic ABoxes. In M. Hermann and A. Voronkov, editors, *Proceedings of the 13th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR 2006)*, pages 227–241, Phnom Penh, Cambodia, November 13–17 2006. Springer.

[MSS04] B. Motik, U. Sattler, and R. Studer. Query Answering for OWL-DL with Rules. In S. A. McIlraith, D. Plexousakis, and F. van Harmelen, editors, *Proceedings of the 3rd International Semantic Web Conference (ISWC 2004)*, volume 3298 of *LNCS*, pages 549–563, Hiroshima, Japan, November 7–11 2004. Springer.

[MYQ$^+$06] L. Ma, Y. Yang, Z. Qiu, G. T. Xie, Y. Pan, and S. Liu. Towards a Complete OWL Ontology Benchmark. In Y. Sure and J. Domingue, editors, *Proceedings of the 3rd European Semantic Web Conference (ESWC'06)*, volume 4011 of *Lecture Notes in Computer Science*, pages 125–139. Springer, June 11–14 2006.

[Nag07] G. Nagypal. *Possibly imperfect ontologies for effective information retrieval*. PhD thesis, Universität Karlsruhe, 2007.

[Neb90] B. Nebel. Terminological reasoning is inherently intractable. *Artificial Intelligence*, 43(2):235–249, 1990.

[NFF$^+$91] R. Neches, R. Fikes, T. Finin, T. Gruber, R. Patil, T. Senator, and W. R. Swartout. Enabling technology for knowledge sharing. *AI Magazine*, 12(3):36–56, 1991.

[O'K90] R. A. O'Keefe. *The craft of Prolog*. MIT Press, Cambridge, MA, USA, 1990.

[PSHH03] P. F. Patel-Schneider, P. Hayes, and I. Horrocks. OWL Web Ontology Language: Semantics and Abstract Syntax. http://www.w3.org/TR/owl-semantics/, February 2003.

[Rob65] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[RTH08] S. Rudolph, T. Tserendorj, and P. Hitzler. What Is Approximate Reasoning? In D. Calvanese and G. Lausen, editors, *Proceedings of the 2nd International Conference on Web Reasoning and Rule Systems (RR2008)*, volume 5341 of *Lecture Notes in Computer Science*, pages 150–164, Karlsruhe, Germany, October 31 – November 1 2008. Springer.

[Sal86] G. Salton. Another look at automatic text-retrieval systems. *Communications of the ACM*, 29(7):648–656, 1986.

[SBF98] R. Studer, R. Benjamins, and D. Fensel. Knowledge Engineering: Principles and Methods. *Journal of Data and Knowledge Engineering*, 25(1-2):161–197, 1998.

[SBK$^+$07] S. Schlobach, E. Blaauw, M. El Kebir, A. ten Teije, F. van Harmelen, S. Bortoli, M. C. Hobbelman, K. Millian, Y. Ren, S. Stam, P. Thomassen, R. C. van het Schip, and W. van Willigem. Anytime Classification by Ontology Approximation. In R. Piskac, F. van Harmelen, and N. Zhong, editors, *New Forms of Reasoning for the Semantic Web*, volume 291 of *CEUR Workshop Proceedings*, November 11 2007.

[SC95] M. Schaerf and M. Cadoli. Tractable Reasoning via Approximation. *Artificial Intelligence*, 74:249–310, 1995.

[SI94] C. Sakama and K. Inoue. An alternative approach to the semantics of disjunctive logic programs and deductive databases. *Journal of Automated Reasoning*, 13:145–172, 1994.

[SK91] B. Selman and H. A. Kautz. Knowledge Compilation using Horn Approximations. In *Proceedings 9th National Conference on Artificial Intelligence (AAAI-91)*, pages 904–909, Anaheim, California, July 14–19 1991. AAAI Press / MIT Press.

[SK96] B. Selman and H. Kautz. Knowledge Compilation and Theory Approximation. *Journal of the ACM*, 43:193–224, 1996.

[SPG$^+$07] E. Sirin, B. Parsia, B. Cuenca Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical OWL-DL reasoner. *Journal of Web Semantics*, 5(2):51–53, 2007.

[SS09] R. Studer and S. Staab. *Handbook on Ontologies – International Handbooks on Information Systems, 2nd extended edition*. Springer, 2009.

[SSS91] M. Schmidt-Schauß; and G. Smolka. Attributive Concept Descriptions with Complements. *Journal of Artificial Intelligence*, 48(1):1–26, 1991.

[Sv02] H. Stuckenschmidt and F. van Harmelen. Approximating Terminological Queries. In T. Andreasen, A. Motro, H. Christiansen, and H. Legind-Larsen, editors, *Proceedings of the 4th International Conference on Flexible Query Answering Systems (FQAS)'02)*, pages 329–343, Copenhagen, Denmark, October 27-29 2002. Springer.

[Tes01] S. Tessaris. *Questions and answers: reasoning and querying in Description Logic*. PhD thesis, University of Manchester, 2001.

[TGH08] T. Tserendorj, S. Grimm, and P. Hitzler. Approximate Instance Retrieval. Technical report, FZI Research Center for Information Technology, Karlsruhe, Germany, December 2008.

[TH06] D. Tsarkov and I. Horrocks. FaCT++ Description Logic Reasoner: System Description. In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR)*, volume 4130 of *LNAI*, pages 292–297, Berlin, August 16–20 2006. Springer.

[Tob00] S. Tobies. The complexity of reasoning with cardinality restrictions and nominals in expressive description logics. *Journal of Artificial Intelligence Research (JAIR)*, 12(1):199–217, 2000.

[Tob01] S. Tobies. *Complexity Results and Practical Algorithms for Logics in Knowledge Representation*. PhD thesis, RWTH Aachen, Germany, 2001.

[TRKH08] T. Tserendorj, S. Rudolph, M. Krötzsch, and P. Hitzler. Approximate OWL-Reasoning with Screech. In D. Calvanese and G. Lausen, editors, *Proceedings of the 2nd International Conference on Web Reasoning and Rule Systems (RR2008)*, volume 5341 of *Lecture Notes in Computer Science*, pages 165–180, Karlsruhe, Germany, October 31 – November 1 2008. Springer.

[Vol04] R. Volz. *Web Ontology Reasoning with Logic Databases*. PhD thesis, Institute AIFB, University of Karlsruhe, 2004.

[WGQH05] S. Wang, Y. Guo, A. Qasem, and J. Heflin. Rapid Benchmarking for Semantic Web Knowledge Base Systems. In Y. Gil, E. Motta, V. R. Benjamins, and M. A. Musen, editors, *The Semantic Web - ISWC 2005, 4th International Semantic Web Conference*, pages 758–772, Galway, Ireland, November 6–10 2005. Springer.

[WGS05] H. Wache, P. Groot, and H. Stuckenschmidt. Scalable Instance Retrieval for the Semantic Web by Approximation. In M. Dean, Y. Guo, W. Jun, R. Kaschek, S. Krishnaswamy, Z. Pan, and Q. Z. Sheng, editors, *WISE Workshops*, volume 3807 of *Lecture Notes in Computer Science*, pages 245–254, New York, NY, USA, November 20–22 2005. Springer.

[WLL$^+$07] T. Weithöner, T. Liebig, M. Luther, S. Böhm, F. von Henke, and O. Noppens. Real-world Reasoning with OWL. In E. Franconi, M. Kifer, and W. May, editors, *Proceedings of the European Semantic Web Conference, ESWC2007*, volume 4519 of *Lecture Notes in Computer Science*, Innsbruck, Austria, July 3–7 2007. Springer.

[WLLB06] T. Weithöner, T. Liebig, M. Luther, and S. Böhm. What's Wrong with OWL Benchmarks? In E. Franconi, M. Kifer, and W. May, editors, *In Proceedings of the Second International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2006)*, volume 4519 of *Lecture Notes in Computer Science*, pages 101–114, Athens, GA, USA, November 5–9 2006. Springer.

[Zil96a] S. Zilberstein. Resource-bounded reasoning in intelligent systems. *ACM Computing Surveys*, page 15, 1996.

[Zil96b] S. Zilberstein. Using anytime algorithms in Intelligent systems. *Artificial Intelligence*, fall:73–83, 1996.

[ZR96] S. Zilberstein and S. Russell. Optimal composition of real-time systems. *Artificial Intelligence*, 82(1-2):181–213, 1996.