From Ontology-enabled Services to Service-enabled Ontologies

Korotkiy, M.

2009

document version
Publisher's PDF, also known as Version of record

From Ontology-enabled Services to Service-enabled Ontologies

Making Ontologies Work in e-Science with Onto⇔SOA

Maksym Korotkiy

SIKS Dissertation Series No. 2009-14

The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

ISBN/EAN 978-90-5335-193-2

VRIJE UNIVERSITEIT

From Ontology-enabled Services to Service-enabled Ontologies

Making Ontologies Work in e-Science with Onto⇔SOA

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. L.M. Bouter,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op donderdag 18 juni 2009 om 10.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door
Maksym Korotkiy

geboren te Kirovsk, Oekraïne

promotor: prof.dr.ir. J.L. Top

To my family

Preface

To me the end of my PhD work signifies the end of a long journey through the vaults of knowledge at two universities. It began in 1996 in the Kharkov State Technical University of Radio Electronics (KhTURE) in Ukraine. I spent 9 semesters in the city of Kharkov studying Decision Support Systems. At that time I was very much fascinated by and overwhelmed with the vastness of Computer Science so I did not even think about doing any research in that area.

In 2000 I got an opportunity to take part in an exchange program between KhTURE and Vrije Universiteit Amsterdam (VU). In the end of 2000 I arrived to the Netherlands and enrolled into the master's program in AI. The AI study was fascinating but it was something else that discovered the joys of research for me. At VU I joint the group of Dieter Fensel as a part-time junior researcher. Being part of this dynamic, international, inspiring and very supportive team has helped me greatly to adjust to the new way of studying and working. Having spent two years as a junior researcher it became easier to see myself as a full-time PhD student which I became in the end of 2002 in a different group and with a new supervisor.

After graduating VU with a master's degree in AI I became a PhD student there under supervision of Jan Top. The subject of my research, the Semantic Web, was very much familiar to me from my previous experience but the work environment has changed significantly. It took me some time to fully realize what had changed and to adjust accordingly. At the end, doing PhD was not as joyful as I thought after working two years as a junior researcher. It turned out to be difficult and challenging work, on many occasions hard and sometimes even frustrating. But as we all know – the greater the challenge, the greater the reward.

To me the greatest reward is what I have learned and I believe I have learned a lot doing my PhD. I learned to do research independently, I realized that often your work is judged not only by using logic and objective criteria but also by probing how confident you are about

it. I learned to accept critique more readily, and try to get the most out of it to improve my research and papers. I learned to make an effort not to reject ideas too hastily, to take my time to try to find a context in which an idea would start making sense to me. Often it helped me to better understand the subject in question and why it was presented this way.

Through my PhD journey I have met quite a few people that have affected its direction in a significant way, helped me to shape my research and made it possible for me to learn. I am very much grateful to all of them and I hope that I also was (or will be) able to give something back in return. In particular I would like to thank Jan Top both for his significant contribution to my work and for his dedication to manage the unmanageable. I would like to thank all my colleagues from VU and the Agrotechnology & Food Sciences Group from the Wageningen University Research Center as they largely created the environment in which my research was developed. I would like to thank Dieter Fensel for showing that research can also be fun and visionary. I especially would like to thank my friends (notably Borys Omelayenko) and family (furthermost my wife Zhanna) for being there for me and supporting me all this time.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Knowledge Reusability

Although there is still an ongoing debate on what knowledge is (see [2, 3, 4] for some of proposed definitions), it is indisputable that it plays a prominent role in the advancement of humanity. The modern notions of *knowledge-based* society and *knowledge worker* [5] are confirmations to that. Knowledge and *intelligent behavior* closely relate to each other. This connection is well illustrated with the field of Artificial Intelligence within which many knowledge-related disciplines such as Knowledge Representation, Knowledge Management, Knowledge-based Systems have emerged over the years. Thus, we believe that knowledge possesses certain intrinsic qualities that enable such profound impact. In this section we will outline some of these qualities and how we can both benefit from them and further enhance them.

In this dissertation we consider *usability* and *reusability* to be among these enabling characteristics of knowledge. The reusability characteristic is most prominent in *declarative knowledge* that is typically expressed in descriptive sentences stating *what* exists in a domain of discourse [6]. Declarative knowledge is often opposed to *procedural knowledge* that instructs on *how* to perform a certain task in the most effective and efficient way. Procedural knowledge is considered to be less reusable than declarative knowledge due to its tight coupling to a single task whereas declarative knowledge can be exploited in various and unforeseen applications. However, we believe that for exactly the same reason, procedural knowledge is much easier to utilize in a given context, implying better usability.

The development of mankind has been accompanied with new and more advanced means for knowledge acquisition and expression. The appearance of spoken languages has

enabled human beings to externalize knowledge. By expressing and sharing it with others we achieve continuity of knowledge across generations. Hence, we can assume that one of the key qualities of knowledge is that it is *transferable* from one carrier to another. Knowledge transfer normally relies on *a language* (e.g., a spoken language) to express knowledge and *communication* means (e.g., act of speech) to deliver it to another party.

Written languages and writing systems such as book printing provided a more effective medium for knowledge communication. This has further improved precision of knowledge transfer and made accumulation of knowledge scalable beyond the capacity of a single individual. Ancient libraries already contained vast amounts of knowledge that no single person could ever accumulate and pass on relying on speech only. Written languages have further improved means to express, communicate and accumulate knowledge thus greatly contributing to its transferability making more knowledge available to a greater number of people.

The pioneer thinkers of Ancient Greece (Plato and Aristotle) have laid a basis for the notion of knowledge as we know it now (epistemology of Plato and numerous Dialogues, Aristotle's scientific method [7]). Their work has been further developed by Descartes (and others) into what has become the scientific method [8]. The scientific method has drastically changed the way knowledge is acquired and allowed to acquire *objective knowledge* – knowledge that is not specific to a certain person or situation. Being objective – independent from its source – knowledge acquires more potential destinations, becomes more transferable and further increases its outreach.

The scientific method has also greatly facilitated progressive construction of scientific theories that heavily relied on their predecessors. In just a few hundred years the critical amount of knowledge has arguably triggered (or at least significantly contributed to) the industrial revolution [9]. This gave start to a dramatic transformation of society making it ever since dependent on knowledge. The technological progress made more resources available to science. The combination of effective scientific method and ever increasing resources contributed to the explosive growth of information and knowledge.

At one point, creation, distribution and use of information have become dominant activities transforming the industrial society into an information/knowledge-based one. To support and manage this growth and large scale distribution information technologies empowered by Computer Science (Knowledge Management) have been employed. To keep up with the accelerating growth novel methods and solutions are required to effectively manage distributed information and knowledge.

The scientific method is primarily a tool for understanding the world around us, i.e. for acquiring knowledge. The advances in scientific method made this tool ever more efficient. However, it still remains a challenge to find out *how to represent acquired knowledge in such a way that it can be efficiently applied*. We believe that application of knowledge (extraction of utility from it) is the final phase of knowledge transfer. In this work we focus on the application of knowledge and will use usability, as a characteristic of knowledge that enables its effective and efficient application.

The trade-off between usability and reusability of knowledge seems inevitable. The well-established separation of knowledge into declarative (i.e. reusable) and procedural (i.e. usable) is an illustration of that fact. In the Knowledge Representation field declarative knowledge has received most attention so far, whereas traditional Software Engineering typically is centered on procedural knowledge. In our work, however, we bring procedural knowledge into the focus striving to bridge these two types of knowledge. More specifically, we seek for an approach that enables flexibility in shifting the balance between usability and reusability of knowledge representation.

Usability represents the pragmatic (utilitarian) perspective on knowledge. From this perspective, knowledge transfer is effective only when a receiving party can efficiently extract utility from the acquired knowledge. In this work we define a method that facilitates application of knowledge. The presented method expands upon two existing Knowledge and Software Engineering techniques: ontologies and services. We claim that the presented approach contributes to *efficiency* and *effectiveness* of both of its underlying techniques.

1.2 Ontologies

The increasing size of the World Wide Web is a prominent example of the rapid and explosive growth of information usage on a truly global scale. In just about a decade the Web has grown into a world wide, densely connected information network. Modern information technology is scalable enough to store and transfer large amounts of data. However, the capabilities of information analysis methods are insufficient to effectively manage and use all these resources.

The main reason for this is that software (agents, applications, components, etc) cannot cope with the high degree of variety and human-orientation of Web resources. To address these problems, Tim Berners-Lee has proposed the vision of the Semantic Web – an evolution of the Web where disparate and decentralized software agents can “understand” Web resources and facilitate their exploitation [10]. To enable this understanding the Semantic

Web relies on ontologies to facilitate effective knowledge transfer between creators of Web resources and software that exploits these resources.

Since we focus on the pragmatic side of knowledge, we assume that *effectiveness* and *efficiency* of ontologies are determined by the amount of utility (effectiveness) a software agent can extract from it and by how much effort (efficiency) is required for that. Therefore, we seek to improve these two qualities of ontologies by introducing elements of procedural knowledge into the predominantly declarative knowledge representation employed in the state of the art ontology languages.

Having been borrowed by AI/KR researchers from philosophy, ontologies were exposed to a wide range of research fields such as Intelligent Agents, Knowledge Management, Semantic Web. In philosophy Ontology is the study of things that exist. The first reference to the term ontology is considered to be in the works Ogdoas Scholastica (1606) by Jacob Lorhard (Lorhardus) and Lexicon philosophicum (1613) by Rudolph Gockel. However, the subject of Ontology – the study of what exists – was addressed already at the times of Aristotle. From the very beginning Ontology attempts to describe the world in terms of *categories* and *relations* between *entities*. The same concepts are still employed in the state of the art ontology languages.

In Artificial Intelligence one of the earliest and the most frequently cited definitions of an ontology is given by T.R. Gruber: *an ontology is an explicit specification of a conceptualization* [11]. Undeniably, it is hardly possible to define an ontology in one sentence, therefore numerous additions to that definition have followed. Already in 1995 clarification was required on what the term ontology means in the fields of Artificial Intelligence and Knowledge Representation [12]. Recently, because of the activities of the World Wide Web Concortium (W3C) (RDF/S [13, 14], the OWL family of ontology languages [15], the Semantic Web [16]) ontologies have attracted even more interest. Nevertheless, we believe that the definition by T.R. Gruber has largely defined the perception of ontologies in numerous research fields.

The dream of Artificial Intelligence – making computers understand – is yet to come true but ontologies are expected to play a key role in achieving it. Ontologies aim to achieve the effect of understanding in software agents through *ontological commitment* – a shared agreement on what a concept means. Ontologies in the Semantic Web are not directly concerned with what a concept actually means to humans. Instead, they aim at achieving a wide-enough agreement about its meaning between software agents. An ontology specifies this meaning by declaring *what* kind of relationships a concept maintains with other

concepts. Such declarative knowledge can then be employed to validate *how* a concept is interpreted by a software agent.

Domain knowledge described in ontologies can be employed to enrich information and data resources and make them more understandable to software agents. The ontology-based annotations of Web pages [17], Web services [18, 19] are typical examples of such an approach. Application of ontologies can potentially improve the accuracy and effectiveness of analysis of annotated resources, facilitating such tasks as information and document retrieval [20]. However, the declarative nature of the represented knowledge leaves a significant gap between *what* constitutes a domain and *how* that domain can be exploited, thus making ontologies less suitable for a direct utilization. We believe that by introducing procedural elements into ontologies we can transform them into knowledge sources suited for direct consumption by software agents.

To facilitate large-scale ontological commitment modern ontology languages employ formal declarative representation techniques such as Description logics [21]. These techniques allow to describe what a concept means thoroughly and unambiguously (with respect to the underlying formal foundation). The benefits of this approach are numerous and undeniable. However, so are the obstacles to overcome on the way to effectively and efficiently employ concepts defined in such a way:

- Formal methods rely on abstraction to reduce the number of represented aspects of a concept. The main consequence of this is that a formally represented concept may lack aspects crucial for unforeseen application scenarios and may clash with intuitive understanding of the meaning of the concept.

- Different formal methods often employ non-complementary (or even incompatible) modeling techniques making their integration very difficult and often even impossible.

- Complexity of formal methods requires highly trained personnel to create ontologies and then to apply them. Significant effort (investment) is required to design ontologies and even then it is not immediately clear how to apply them to a problem at hand (let alone to an unforeseen application).

- Declarative specification of *what* a concept means does not directly imply *how* we can extract utility from it.

If we refer back to Gruber's definition of ontologies we conclude that there is a limitation: this definition encapsulates the essence of an ontology in the notion of *conceptualization* without defining the latter. Moreover, the definition implies that the specification and conceptualization mechanisms are independent. However, since the capabilities of formal representation mechanisms (traditionally employed in ontology languages) are inherently limited, they dictate the conceptualization. Such a link between specification and conceptualization does not pose a problem for artificial and abstract domains (e.g., mathematics), however it poses significant difficulties if applied to the domains grounded in real-world (physics, biology, business domains, etc).

The prevalence of specification over conceptualization implies that knowledge to be captured in an ontology has to be conceptualized according to the formal mechanism underlying the ontology language. Such conceptualization of knowledge is likely to be significantly different from (misaligned to) a conceptualization employed by a domain expert. We believe that this negatively affects the usability of ontologies, preventing efficient extraction of utility relevant to the domain of the ontology.

Moreover, if we consider all stakeholders (domain experts, ontology engineers, software engineers) involved into the creation and exploitation of an ontology (in software) then the effect of misalignment is amplified significantly. This, we believe, prevents effective use of ontologies in software applications.

To summarize, ontologies aim to achieve shared understanding through a wide scale commitment. However, by choosing a formal and declarative approach they not only introduce a significant gap between *what* exists in a domain and *how* it can be exploited, but also lead to the domination of specification over conceptualization resulting in a misalignment between the captured meaning of a concept and its understanding by a domain expert. The gap and the misalignment decrease the usability of ontologies and hinder their acceptance. In our work we aim to mitigate these drawbacks and to improve the usability of ontologies by extending them with non-declarative and non-formal elements.

To improve the usability of ontologies and knowledge captured therein we propose to expand their underlying specification mechanism beyond the formal ones. This will provide more flexibility in choosing knowledge conceptualization, thus making it less dependent on specification. We expect the proposed extension to allow for better alignment between ontological conceptualization and an expert's perception of a domain as well as to provide a readily-available utility to software agents. We propose to use a *service* to carry ontological utility directly exploitable by software agents.

1.3 Services and Service-Oriented Architectures

Arguably, the success of the Web has been the single most important factor behind the rise of services. Companies have recognized the value of openness and accessibility of the Web and started offering business services accessible via the Web. The raise of Web shops, hotel and ticket reservation services gave birth to e-commerce and e-services.

World-wide penetration makes the Web very attractive not only for reaching potential customers but also as a global medium for business-to-business communication. In the same manner as the Web has enabled Web pages to refer to each other, it enables inter-software communication on a global scale. Each of the many approaches to distributed computing (distributed objects, client/server, remote procedure calls, services, etc) can benefit from such communication medium. Services and Service-Oriented Architectures (SOA) attract a significant amount of attention in industry and academia by promising to provide an effective way to reuse (transfer and apply) business functionality captured in software components. This will enable enterprises to timely respond to changes in the business environment by either adjusting existing or providing new services.

The exact meaning of what a service is differs across communities [22]. In this work we do not attempt to embrace all existing viewpoints on a service. Instead, we consider a service to be a software component and focus on its two arguably most distinguishing characteristics – business alignment and loose coupling:

- *Loose coupling* means that a consumer (e.g., a software component) can employ a service using only a small number of assumptions about its interface. No knowledge of internal implementation details is required.

- *Business alignment* determines the effectiveness of a service by ensuring that a service directly supports a business process. Business alignment also implies that a service encapsulates a coarsely-grained functionality visible at the business-level.

We consider a service as a means to efficiently deliver functionality readily exploitable by software agents. The delivered functionality, on the other hand, must be effective for the purpose at hand. Thus, we believe that ontologies and services can complement each other and have a potential to:

- improve effectiveness (i.e. business alignment) of services by employing proper domain conceptualization specified in ontologies;

- improve usability of ontologies by allowing them to exploit services as carriers of procedural domain knowledge directly available to software agents;
- improve overall knowledge transfer by using a combination of an ontology (containing declarative knowledge) and a service (containing procedural knowledge) to capture knowledge preserving intuitive domain conceptualization and making it readily-available for consumption by software agents.

We do not claim that this work contains a complete account of a new ontology- and service-based *knowledge representation* technique. However, our belief is that by applying the introduced framework to real life e-Science problems we provide a sufficient evidence of the potential of the proposed approach, and the combination of ontologies and services in general.

1.4 e-Science

Normally the term e-Science refers to computationally intensive and massively distributed numerical computing for scientific purposes [23]. However, recently the term has gained a broader meaning and refers to a scientific process facilitated by advanced information and knowledge processing means. One of the reasons for this expansion of the term's meaning is that science itself, the primary cause of knowledge explosion, started requiring more advanced techniques to manage vast amounts of accumulated knowledge. Over time the depth of scientific explorations has grown, implying an ever more narrow specialization of a growing number of (sub-)fields and disciplines. These fields departed from each other, increasing the effort required to reuse knowledge across fields. Presently, interdisciplinary fields (bioinformatics, biophysics, etc) experience these problems the most. To provide effective support to scientists in managing and applying volumes of existing knowledge novel approaches are required.

In this dissertation we apply techniques and ideas originated in the field of the Semantic Web (ontologies) and Software Engineering (services) to the field of e-Science. e-Science supplies us with practical problems that we employ as use cases that are addressed with the proposed framework.

Generally speaking, the scientific process can be seen to consist of the following activities:

- Define a research question.

- Gather and analyze relevant information and knowledge.
- Form a hypothesis explaining or answering the question.
- Plan and perform an experiment.
- Analyze the results (data) of the experiment to confirm or reject the hypothesis. If necessary refine and re-evaluate the hypothesis.
- Publish results (acquired knowledge).

The effectiveness and efficiency of these steps depends on the capability of scientists to discover, assess and apply various types of knowledge and information. For example, to gather and analyze knowledge (normally in the form of publications, reports, results of experiments, etc) a scientist has to be able to quickly locate it and assess its relevance to the question at hand.

For experiment execution and subsequent analysis of results a scientist has to be able to effectively *apply* knowledge sources discovered at the preceding steps. This indicates that we should assist the scientist not only in processing (discovering, estimating relevance, etc) static descriptions of knowledge but also in actual applications of existing knowledge.

Finally, when new knowledge is acquired it has to be expressed and published in a form that facilitates its future (re-)use (discovery, assessment and application).

Providing an automated support for these activities is a challenging task due to a great variety of forms scientific knowledge and information take, their distribution, interconnectedness, disparate origin and so forth. These characteristics make the challenge rather similar to the general task of improving accessibility of Web resources addressed in the Semantic Web. More specifically, software agents experience difficulties in coping with weak structure, ambiguity and conceptual incompatibility of resources.

There is also a number of features that distinguish the field of e-Science and its environment from the Web in general:

- e-Science is knowledge intensive: it is concerned with producing new knowledge by using available knowledge sources as much as possible. In particular a strong dependency on mathematics sets e-Science apart from other areas of application.
- e-Science is interdisciplinary: knowledge sources come from a variety of fields and many of them are based on rather unique and disjoint conceptual systems.

- e-Science is computation intensive: many e-Science fields require extensive simulation, computational processing of obtained data, etc. However, even when a knowledge resource is well specified, it is still a problem to apply it: the step from a description of knowledge to executable software is a non-trivial one.

- e-Science varies in scale: the scale of knowledge reuse and application differs significantly from a single scientist, to a laboratory, research institutions, research community, discipline, field, etc. Most of the day-to-day activities take place on a small to medium scale in labs and research teams.

In e-Science information and knowledge resources are expressed in a variety of forms. In our work we employ use cases that address the tabular representation of data (e.g., spreadsheets), executable models (expressed in the Matlab language), reports and publications. We believe that ontologies and services can provide the means to improve many e-Science activities. More specifically they have a potential to:

- improve accessibility of static knowledge resources to software agents, thus improving effectiveness (e.g., precision) of specific computer-facilitated tasks such as information retrieval;

- make knowledge resource (methods, algorithms, computational models) readily applicable;

- improve compatibility and integration of various knowledge sources both static and dynamic;

- facilitate development of software solutions supporting e-Science by enhancing the development process and by enabling a greater degree of reuse of existing knowledge components.

In this thesis we demonstrate that by applying the proposed ontology- and service-based framework we can indeed realize many of these benefits.

1.5 Research Questions

In our work practical application of knowledge plays a central role. Therefore our initial research question is:

1 How can typical e-Science tasks be facilitated with ontology-enabled software solutions? In this research question we specifically investigate how the following e-Science tasks (employed as use cases) can be facilitated with an ontology-enabled and service-oriented technique:

- **1.1 The document retrieval task** is about finding documents by matching a given query against a collection of documents (publications, experimental results, etc).
- **1.2 The unit conversion task** is about finding a conversion factor that can be used to convert one unit of measure into another (e.g., meter to inch).
- **1.3 The task of checking consistency** of models and datasets in terms of the units of measurement and dimensions used.
- **1.4 The integration of tabular data with an ontology-ready data-model such as RDF** is about making results of experiments represented in a tabular form (e.g., in spreadsheets) available to RDF-aware systems.

The initial research question outlines the application context of this work – e-Science. To answer this question we have to find a uniform ontology- and service-oriented framework that can be employed to provide solutions to the above-mentioned tasks. Thus, we arrive at the *central research question* of this work.

2 How can ontologies and services (Service-Oriented Architectures) be integrated into a framework facilitating application of knowledge in e-Science (and beyond it)? The answer to this question constitutes an abstract framework consisting of general, ontology language and technology independent, design principles and guidelines. In order to validate this abstract framework and to be able to actually employ it to the target use cases we have to answer two more specific sub-questions:

- **2.1 How can we integrate RDFS ontologies with REST services?** In this sub-question we apply the general design guidelines to the RDF and RDFS ontology languages and REST-like services. The result should provide us an operational framework applicable to the described e-Science tasks.
- **2.2 How can ontology-enabled services work together?** Composition of services enables effective construction of complex functionality from simpler artifacts and,

thus is an important component of a service-oriented approach. To answer this research question we will design and implement a composition framework for ontology-enabled services.

By employing tools such as ontologies and services to e-Science tasks we gain insights in what constitutes usability and how ontologies and services themselves benefit from the integration with each other. In addition to the main research questions we will address a number of more general issues that summarize our experience gained while pursuing the central research question. These general issues are:

3 How can we attach a service to an ontology and what does this imply for ontologies? To answer this question we investigate the effect of ontologies and services on each other's *usability*. We will also analyze the framework proposed to address the second question from the ontological perspective by re-interpreting it as a service-enabling mechanism for ontologies.

4 How do ontologies affect characteristics of software development such as software quality and development effort? In this question we analyze the overall potential of ontologies with respect to Software Engineering practice. The answer to this question will allow us to estimate the overall practical feasibility of ontology-enabled Software Engineering.

An important orthogonal requirement to the aforementioned research questions is that the proposed solutions should be easily deployable by both software and knowledge engineers. This requires us to maintain an acceptable level of complexity, reusing well-established design principles and combining existing methods and technologies in a novel but still natural and intuitively understandable way.

1.6 Research Method

In this work we employ the constructive research method [24]. We aim to provide feasible (implementable in a prototype) and specific solutions to the stated research questions. At the same time we strive to make the proposed approach general enough (or easily generalizable) and outline the boundaries of its applicability.

Whenever possible and practically feasible we build prototype implementations of the

proposed solutions. In some case this requires further specialization of the proposed solution. Hence, our answers to the research questions can be considered on three levels of generality:

- a general technology, and ontology language independent framework applicable as widely as possible (Ontology-enabled Services, Service-enabled Ontologies, Blackboard-style Collaboration of Ontology-Enabled services);
- operationalization of the proposed frameworks and their implementation in a corresponding prototype middleware or application program interface (API) (RDF/S-enabled REST Services, the FDR2 approach for linking relational and RDF models);
- a concrete application of an operationalized framework to the use cases that results in a proof-of-concept (Web) application.

The border between the first two types of abstractions is not defined strictly. We allow for a transition of assumptions and design choices between these two levels. Therefore, the combination of the abstract framework and its operational counterpart (e.g., the Ontology-enabled Services and RDF/S-enabled REST Services frameworks) should be regarded as a single approach even if they are described in several chapters.

The operational frameworks are employed in the target e-Science use cases. In all use cases, except one, the novelty is in the application of the proposed approach to design and implement the solution rather than in the details of the solution itself. In other words, we employ the use cases to illustrate the design choices guided by the proposed approach to arrive at an ontology-enabled service-oriented architecture. However, internally in the components composing this architecture we employ existing techniques for document retrieval, unit conversion and consistency checking. The exception to this is the use case **1.4** (the integration of relational and RDF models) where both the proposed solution and the approach to its implementation are novel.

In two chapters we use a more theoretic (speculative) approach. In Chapter 3 we reinterpret the proposed ontology-enabled service-oriented framework from the Ontology Engineering perspective. In Chapter 7 we perform a theoretical estimation of the impact of ontologies on Software Engineering practice in general. The argumentation employed in these chapters is derived from our experience in implementing ontology- and service-enabled solutions to the target use cases. We believe these chapters provide valuable insights into relationships between ontologies and services and, more generally, Ontology and Software Engineering practices.

1.7 Summary of Contributions

The main contributions of this dissertation are the following:

- Onto⇔SOA – a framework that combines ontologies and services to facilitate representation and application of knowledge.
  - Definition of a restricted document-oriented service model that emphasizes the *domain alignment* and *loose coupling* characteristics of Service-Oriented Architectures.
  - Ontology-enabled Services – an approach aimed at improving effectiveness of Service-Oriented Architectures by enhancing their *domain alignment* and *loose coupling* characteristics by means of ontologies.
  - A Blackboard-style Service Composition mechanism extends Ontology-enabled Services to provide an application-independent way to combine such services in a simple yet effective way.
  - Service-enabled Ontologies – a general mechanism that couples a service to an ontology to provide domain-specific inferences capabilities and capture *application semantics* of domain concepts.
- The *MoRe* framework – an operationalization of Onto⇔SOA that employs the RDF data-model, RDFS ontologies and REST-like services.
- A number of solutions for the e-Science domain have been designed and implemented in a novel ontology- and service-enabled way according to the Onto⇔SOA framework:
  - the Document Retrieval, Unit Conversion, Unit and Dimension Consistency Checking services and demo applications;
  - the FDR2 approach and corresponding services for connecting relational-like (tabular) data to the RDF-data model.
- An estimation of the effect of ontologies on the quality of web application and development effort has been given. This indicates when we can expect benefits from introducing ontologies into software engineering practice.

Chapter 2

Ontology-enabled Services and Service-Oriented Architectures

In this chapter we address the central research question of the dissertation: *"How can ontologies and services be integrated into a framework facilitating application of knowledge in e-Science?"*. To answer this question we introduce Onto⇔SOA – a framework that integrates ontologies into Service-Oriented Architectures. The framework is based upon a *restricted service model* defined as an architectural style that focuses on *domain alignment* and *loose coupling* of services. In Onto⇔SOA we propose to employ ontologies to further enhance these service characteristics. We propose to use an ontology as a *service schema* that defines a *document-oriented* service interface. Through this interface a direct exchange of ontology-based messages between a service and its consumer takes place. In this chapter we employ the Document Retrieval use case as a running example that demonstrates the main ideas behind Onto⇔SOA.

◇

Maksym Korotkiy and Jan L. Top: Onto⇔SOA: From Ontology-enabled SOA to Service-enabled Ontologies. In proceedings of International Conference on Internet and Web Application and Services (ICIW'06). Guadeloupe, 2006.

Maksym Korotkiy and Jan L. Top: Designing a Document Retrieval Service with Onto⇔SOA. In proceedings of the Semantic Web Enabled Software Engineering workshop at ISWC 2006. Athens, GA, U.S.A., 2006.

Maksym Korotkiy: Towards an Ontology-enabled Service Oriented Architecture. In proceedings of the PhD Symposium in International Conference on Service Oriented Computing (ICSOC'05). Amsterdam, the Netherlands, 2005.

◇

Nowadays Web Services and Service-Oriented Architectures (SOA) attract a significant amount of attention in industry and academia by promising to provide an effective and

efficient way to use functionality captured in software components. This will enable enterprises to timely respond to changes in the business environment by either adjusting existing or providing new services.

In this work we assume that effectiveness and efficiency of a service are largely determined by its *business alignment* and *loose coupling* characteristics. Hence, the main goals of this chapter are:

- to identify factors effecting the business alignment and loose coupling characteristics of a service;
- find a way to further enhance these characteristics by means of ontologies;
- provide guidelines on how to design an ontology-enabled service-oriented architecture that possesses the target characteristics.

Web Services [25] are the most widely established technology that can be used to implement SOA on the Web. From the perspective of our work, Web Services can be evaluated by how well they support the loose coupling and business alignment characteristics in SOA. Web Services supply means for *implementation*. However, they do not provide *design* guidelines on when (in what application scenarios) and how to apply Web Services to obtain a business aligned and loosely coupled architecture. Web Services support loose coupling on the implementation level by:

- facilitating service discovery (via service registries);
- reducing (the cost of) dependency on a concrete protocol employed in the messaging layer between a service and a consumer.

Web Services have proven to be a rather effective way to implement SOA. However, they do not provide sufficient support for business alignment. Web services can establish agreement on a service interface at the data level only, lacking the means to facilitate conceptual interoperability.

As in many other fields practitioners of Web Services formulate accumulated experience in the form of design guidelines or patterns [26, 27]. In Onto⇔SOA we include many of these guidelines into the framework itself. Therefore, one of the key differences between Web Services in general and the proposed approach is that we focus on *design* of services rather than on their *implementation*.

Semantic Web Services (SWS) take an approach in which, like in Onto⇔SOA, ontologies are introduced into services. SWS provides an ontology-enabled layer on top of Web Services to further improve their effectiveness through *automation*. OWL-S [28] and WSMO [29] are two of the most well-known SWS approaches. Both OWL-S and WSMO provide extensive ontology-based *description frameworks* for Web Services to enable automation of tasks such as discovery, invocation, choreography and orchestration. This automation will primarily reduce the effort required from a consumer to find or invoke a single service or to compose multiple services. These SWS approaches rely on extensive ontology-based meta-data available about a service, thus increasing the effort required from a service provider to roll out a service. The additional effort is likely to be off-set on the long term by simplifying composition of complex services from simpler ones.

The SWS goal of automation is very challenging due to its unrestricted scope (all varieties of services are targeted) and the finite capabilities of *formal* ontology languages employed in OWL-S and WSMO. Moreover, the SWS approaches simultaneously target a number of tasks each of which is complex in itself. All this significantly increases the complexity of the SWS approaches and requires a significant up-front effort investment by a service provider, thus hindering the overall effectiveness of SOA.

Our intuition is that the level of complexity observed in SWS approaches is not inherently required to improve the effectiveness of SOA by means of ontologies. We illustrate this with the Onto⇔SOA framework. To reduce complexity, we focus on the single task of *service invocation* and rely upon a number of assumptions about services. These assumptions shape the scope of the framework such that Onto⇔SOA and SWS become different, though overlapping approaches to improve the effectiveness of SOA (and Web Services). The main differences between SWS and Onto⇔SOA are:

- SWS rely on automation of tasks normally involving multiple services annotated according to an ontology-based description framework.

- Onto⇔SOA constrains properties of individual services and provides *design guidelines* on how to integrate an ontology into SOA.

Another key difference between SWS and Onto⇔SOA is that we do not employ ontology-based meta-data to describe a service. Instead, we propose to use ontology-based messages as a direct input to a service with a corresponding underlying ontology playing the role of a service schema. This differs from the more traditional data-oriented approach

in which a conceptual domain model is considered as an intermediate step in system design rather than a direct input.

Furthermore, we define the service model underlying Onto⇔SOA from the perspective of Software Architectures. This reduces the scope of possible interpretations of the notion of service, provides a basis for the analysis of the target SOA characteristics and supplies a framework for defining design guidelines.

To validate Onto⇔SOA we have employed it in a number of use cases from the e-Science domain. In this chapter we describe the Document Retrieval case where we design an ontology-enabled service-oriented solution to the problem of finding documents that match a given query. In e-Science this is a task faced by scientists almost daily.

To sum up, the main contributions of our work reported in this chapter are the following (in the order of appearance in the text):

- the definition of a restricted service model, presented as an architectural style, that focuses on the domain (business) alignment and loose coupling characteristics of SOA;
- the definition of Onto⇔SOA – an approach aimed at improving effectiveness of a service-oriented architecture by enhancing its domain alignment and loose coupling characteristics by means of ontologies;
- the application of an Onto⇔SOA solution to the Document Retrieval case.

This chapter is organized as follows. In Section 2.1 we introduce the Document Retrieval case that will be a running example throughout this chapter. Sections 2.2 and 2.3 introduce the Onto⇔SOA approach. After that, we use the Document Retrieval case to illustrate an Onto⇔SOA based solution in Section 2.4. In Section 2.5 we describe the differences between the Onto⇔SOA service model and the broader notion employed in Web Services. In the same section we highlight the differences between the proposed approach and Semantic Web Services. Finally, in Section 2.6 we discuss some issues raised by the proposed framework and conclude with Section 2.7.

2.1 Use Case: Document Retrieval

Effective document retrieval can significantly contribute to various stages of the scientific process. For example, to efficiently analyze existing publications and reports a scientist needs to be able to quickly retrieve relevant documents from available collections. We will

employ the task of retrieving documents as a case to demonstrate the main characteristics of the Onto⇔SOA approach and its underlying service model.

From the user[1] perspective the document retrieval task can be defined as follows. *Given a collection of documents and a text query, find all documents from that collection that match the given query*. We will refer to this definition as the Document Retrieval application (or business) domain – a conceptual description of the problem domain as perceived by the user. In this case, the application domain is described with concepts directly extracted from its definition: *document*, *document collection*, *query*, *retrieved documents* (see the left-hand part of Figure 2.1).

In this chapter we consider two alternative approaches to designing a software solution for the document retrieval task. We employ Lucene[2] – a well-known open source Java API for document retrieval – as a reference object-oriented (OO) design. Our experience indicates that it is common practice to design Web Services as thin, often automatically generated, wrappers around existing objects. Such an approach preserves most of the original OO design characteristics and differs from OO primarily in data serialization and communication layers. We, therefore, can employ the Lucene's OO design as an approximation to this Web Services practice and will contrast it to the Onto⇔SOA service model.

Lucene's approach to document retrieval consists of two main steps implemented by a number of objects, as depicted in the right-hand part of Figure 2.1:

1. *Document Indexing* is supported by the Parser, Analyzer and IndexWriter objects. Parser extracts the structure and the content from a document. Then, Analyzer applies Natural Language Processing techniques (stop-words filtering, stemming, etc) to the document's content. IndexWriter precomputes statistical information (term and document frequencies) and stores it in an *index* along with a term-document map. The main purpose of the index is to improve the performance of the document retrieval process by providing access to the precomputed statistics and documents containing a given term.

2. *Document Search* is supported by the IndexReader object that takes a text query, processes it with Analyzer and accesses the index to obtain the precomputed statistics and documents containing query terms. These statistics are used to compute the ranks of the retrieved documents reflecting how well each document matches the query.

[1] A user is one who is familiar with an application domain and who directly interacts with a service-oriented system by, for example, developing a client application or a service consumer component.

[2] http://lucene.apache.org/

In Lucene the described steps must be coordinated in several ways:

- *Document Indexing* must precede *Document Search*, otherwise the index may contain no data about the documents being queried;
- the same type of Analyzer must be employed in both steps, otherwise processed document terms may not match processed query terms;
- at each of the two steps, the index to be employed must be explicitly identified.

This coordination requires the user (a client object or a service consumer) to implement a complex protocol. According to this protocol the user must carry out steps in a correct order and all steps must be configured in the same way (Analyzer and index must be the same across steps). The user must be aware of these peculiarities of Lucene's approach in order to successfully employ the API.

Lucene's approach to document retrieval can be seen as a refinement of the initial more general definition of the Document Retrieval application domain. The main purpose of this refinement is to construct a rather fine-grained domain model that is expected to increase the probability of components reuse.

This reusability is obtained by exposing the components' *internal details*, introducing inter-component dependencies (e.g., Analyzer can be seen as an internal detail of IndexWriter) and complex interaction protocols (document retrieval is realized as a sequence of indexing and index search). Consequently, the user not only has to understand the application domain, but also to understand the finer-grained implementation model of the API. The user's overall conceptualization has to include the corresponding API concepts such as *analyzer* and *index*. However, strictly speaking, these concepts are irrelevant to the original definition of the Document Retrieval application domain.

In the rest of this chapter we introduce the Onto⇔SOA approach before applying it to the above Document Retrieval case. We begin by employing the Software Architectures perspective to introduce a service model underlying Onto⇔SOA (Section 2.2) and to analyze the loose coupling and domain alignment characteristics. After that, we introduce ontologies to that service model to enhance these characteristics.

2.2 The Underlying Service Model

Different communities discuss business and engineering merits of service orientation. Different viewpoints on and definitions of a "service" exist in these communities [22]. And

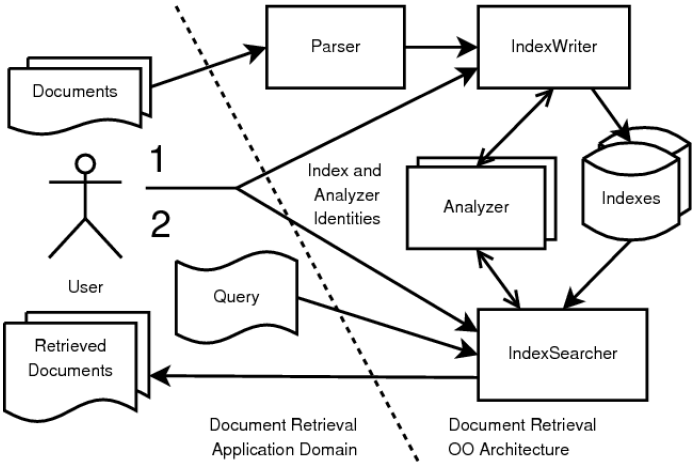

Figure 2.1: The application domain of the Document Retrieval case (left-hand part) and an object-oriented model of a Lucene-based solution to it (right-hand part).

even within the Software Engineering field alone there is no agreement on what "a service" is [30].

In Software Engineering there is a perception that any software component or application can be implemented as a service by making it accessible via the Web. This accessibility is most often achieved by introducing an interface description (e.g., a WSDL-based one), transport (e.g., HTTP) and messaging protocols (e.g., SOAP) into the essentially unmodified component. Such a view addresses only the implementation aspects of loose coupling. However, the *design* aspects of both business alignment and loose coupling are disregarded in this perspective. In this dissertation we focus on the *design* aspects and assume that a software component should be first *designed* as a service, and only then implemented as one.

We consider loose coupling and business alignment to be the key properties of an effective service. To account for this we use Software Architectures to introduce a service model as an *architectural style* [31] restricted to business-aligned and loosely-coupled services. Surprisingly, we have not come across any attempts to define SOA as an architectural style. We would like to emphasize that the interpretation and constraints we apply to services and SOA should be considered primarily within the context of this work (Onto⇔SOA) rather

than as an attempt to give a general definition of a service and SOA. We discuss the main differences between the introduced service model and the more traditional interpretations of services in Section 2.5.1.

By defining the Onto⇔SOA service model as an architectural style we pursue a number of goals:

- to analyze the relationships between the *business alignment* and *loose coupling* characteristics of a service and its internal properties to determine how we can further support these characteristics by means of ontologies;
- to limit the scope of SOA to only those services that possess the *business alignment* and *loose coupling* characteristics;
- to provide guidelines on how to design *business-aligned* and *loosely-coupled* services and architectures, and then to link these design recommendations to the implementation level constraints.

Software Architectures [32] is a field of Software Engineering that strives to provide guidelines for design and analysis of software systems that possess certain characteristics. In [31] an *architectural style* is defined as "*a coordinated set of constraints on architectural elements and relationships among those elements within any architecture that conforms to that style*". The same work defines a *software architecture* as "*an abstraction of run-time characteristics of a software system during some phase of its operation*". As an abstraction, an architecture provides a simplified view on a software system with only relevant characteristics highlighted. Since we can combine architectural characteristics in multiple ways, a software system can have many architectures (or rather architectural views). However, there is a limited number of characteristics (coupling, cohesion, distribution, etc) that are relevant in practice.

A *software architecture* provides the means to analyze the characteristics of a system but it does not instruct on how an architecture should be designed to possess those characteristics. An *architectural style* addresses this issue by imposing constraints on architectural elements, thus inducing desired characteristics on an architecture.

In Software Engineering general architectural elements are *processing components*, *connectors* and *data* [31]. Processing components can transform data elements. Connectors provide an abstract mechanism that mediates communication, coordination or cooperation between components [33]. From the processing component perspective, connectors transfer

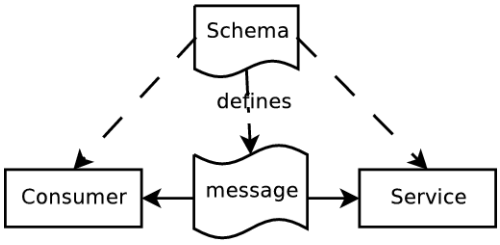

Figure 2.2: Elements of a service-oriented architecture.

data without modifying them. Nevertheless, internally a connector can contain a complex subsystem that subjects the data to a number of intermediate transformations.

Typical elements of a service-oriented architecture are shown in Figure 2.2:

- *Consumer* – is *a processing component* that initiates communication with a service, sends a message request to it and receives a response message from a service. We regard a consumer as a processing element because it is responsible for instantiation of messages.

- *Service* – is *a processing component* that accepts message requests from a consumer, processes them and responds with a message containing the result of processing.

- *Message* – is *a data element* that contains either a request from a consumer or a response from a service.

- *Schema* – is *a connector element* that facilitates communication between a service and its consumer by specifying agreements (syntactical, structural, etc) about the content of messages or provides meta-information about the service itself (e.g., preconditions, process model, etc).

We define the service model underlying Onto⇔SOA as an architectural style that constrains the above-mentioned service elements to induce the loose coupling and domain (or business) alignment characteristics. In the subsequent sections we analyze these characteristics to identify design decisions that allow to realize their benefits as fully as possible. We define these design guidelines by means of architectural constraints that in many instances have the form of (or are inspired by) fairly well-known design (architectural) patterns.

2.2.1 Loose Coupling

Generally speaking, *loose coupling* implies a weak dependency between system components. This characteristic is beneficial for systems subjected to frequent changes because components of a loosely coupled system can be modified independently from each other.

What constitutes loose coupling varies across different types of software systems. In Object-Oriented (OO) design loose coupling often refers to an interface-mediated relationship between objects [34]. An interface describes what types of messages an object can process. Internally objects may be implemented in different ways but as long as they use the same interface they can communicate with each other. In other words, in OO a design is loosely coupled if objects do not depend on the concrete implementations of each other but rather on abstract interfaces (Figure 2.3). If taken literally, this type of loose coupling can be rather easily implemented by merely introducing interfaces of these concrete implementations. For example, in the Java language this can be done by extracting signatures of (public) methods into interfaces. Though such an approach does provide some merits of loose coupling, we believe it does not realize its full potential.

We approach loose coupling on a more general level and use it to refer to reduction of dependencies between processing components. The interface-mediated relationship can then be seen as one of the means to improve loose coupling by reducing dependency on how functionality is actually implemented in a host object. This aspect of loose coupling is readily-achievable in SOA through service schemas. Therefore, in our analysis we have to consider additional ways to reduce dependencies between a service and its consumer.

In services and, more generally SOA, dependencies are either described in a schema (an interface of a service, its location, etc) or expressed in the form of *underlying assumptions* (schema language, service model, etc) that are agreed upon and left outside the schema. These underlying assumptions determine what is described in a schema and how it is expressed. Our aim is to improve loose coupling by minimizing both categories of dependencies.

In the extreme (and hypothetical) case, a service that does nothing (e.g., its schema specifies a service location only) and that can be invoked in the simplest possible way (e.g., by opening a socket connection and sending arbitrary data to it) will approach a perfect loose coupling with its consumers. Obviously, although such a service is very easy to use (understand and invoke) or replace, its very low (absent) utility does not make it attractive to consumers. Therefore, our task is to find a balance between how easy it is for consumers to understand and invoke a service, and the ability of a service to carry sufficient utility.

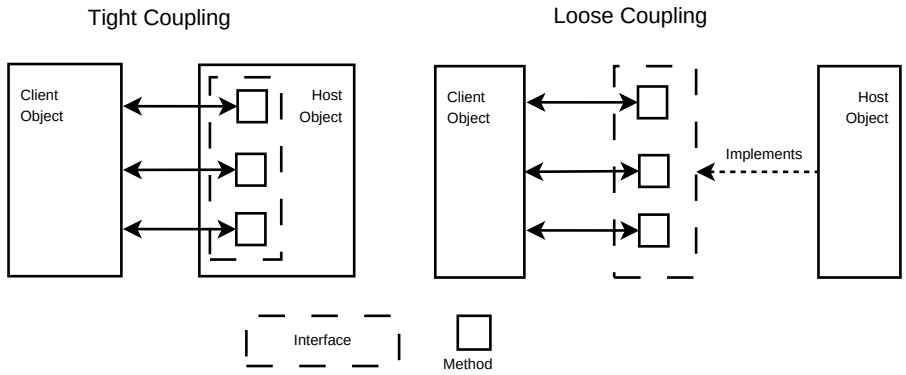

Figure 2.3: Loose coupling as an interface-mediated relationship between objects.

We analyze loose coupling by means of *artificial* and *real dependencies* [35] between the consumer and service components. *Real dependency* exists if a component requires another component's functionality. *Artificial dependency* exists if a component must use a particular API, protocol and so forth in order to employ that functionality. Both artificial and real dependencies always exist, therefore the design goal is to reduce them (or their costs) as much as the context permits.

In the Document Retrieval case there is a real dependency between the IndexWriter and Analyzer objects: IndexWriter requires Analyzer to perform text processing. Artificial dependency appear as follows:

1. In order to interact with Lucene's IndexWriter object a client object must use a Java-compatible invocation mechanism.

2. A consumer must coordinate service invocations as described in Section 2.1.

In the Document Retrieval case the first artificial dependency cannot be significantly reduced because an invocation mechanism is always required. The costs of the invocation mechanism, however, can be reduced. For example, Web Services aim at reducing the cost of dependency on an invocation mechanism by using SOAP as a platform-independent invocation mechanism. In our service model we abstract away operational details of the connector mechanism making it independent from a particular technology such as SOAP or REST.

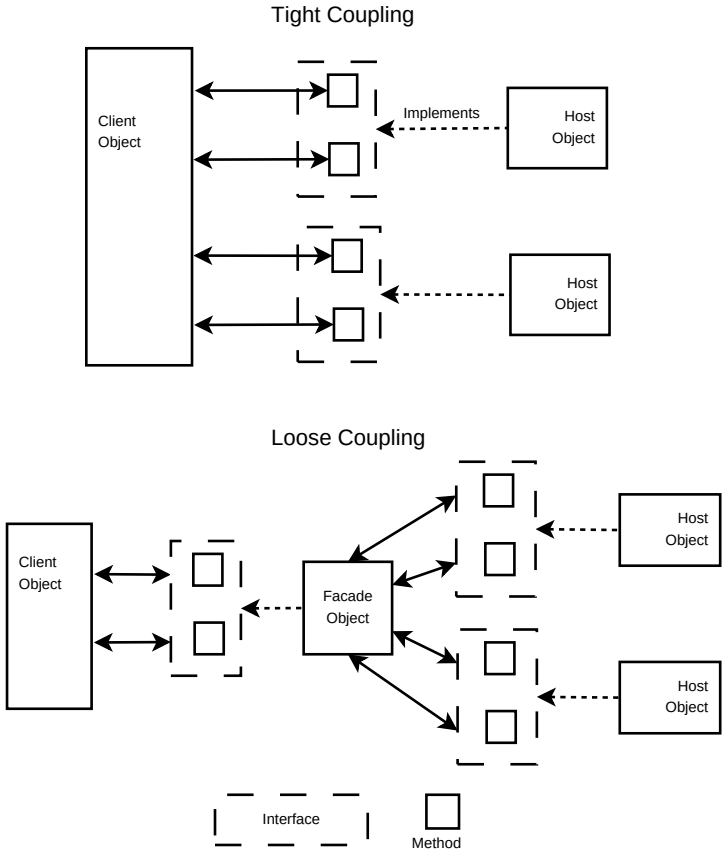

Figure 2.4: Loose coupling as reduction of a number of real (functional) dependencies achieved by introducing a Facade object that encapsulates multiple objects and exposes the composite functionality through a simpler, more coarsely-grained interface.

The second dependency can be reduced by simplifying the required coordination. Figure 2.4 depicts an approach to simplify an interaction protocol by introducing an intermediate Facade object [36] that encapsulates part of the original protocol hiding it from a consumer.

Inducing Loose Coupling in Onto⇔SOA

To reduce artificial dependency between a service and a consumer in the Onto⇔SOA service model we impose three constraints on the connector and data elements:

- **Connectors must be simple, generic and application independent**. This allows us to deploy the connector elements across different application domains and make them less affected by changes in the application domain.
- **Data elements must contain descriptive messages**. A descriptive message describes *what* problem is to be solved rather than *how* to solve it. Descriptive messages require fewer assumptions than prescriptive ones. A prescriptive message must specify an operation to apply and the required input data. Whereas a descriptive message will require only the input data.
- **The schema and message language(s) must be able to address a wide range of application domains**. Preferably the same (or compatible) languages should be used to express a schema and the messages. Such a language supplies interoperating components with a unified syntax and structure required to express the schema and messages. Since we keep the connectors simple and application independent, the messages should be able to accommodate all application-specific details and, therefore, the underlying language must be capable to address a wide range of application domains.

In *document-oriented* messaging[3] – a well-known message design pattern – a service and a consumer exchange messages, referred to as *documents*, that can be directly linked to concepts in an application domain (a purchase order or a document corpus, for example). This contrasts with the lower-level data-oriented messaging such as SOAP-RPC where communicated data structures often reflect the algorithms employed rather than domain concepts.

A document is a more coarsely grained entity than a SOAP-RPC request. The latter is often broken down into separate parts representing operation name, its parameters and result, etc. A document assumes which operation(s) should be performed on it, therefore the elements of a document are best described with a noun (e.g., purchaseOrder) rather than with a verb (e.g., submitPurchaseOrder) used as an operation name in SOAP-RCP.

[3]Document-oriented messaging should not be confused with the document-based encoding style supported by SOAP. For more info see: http://java.sun.com/developer/technicalArticles/xml/jaxrpcpatterns/

In the proposed service model we favor document-oriented messaging because it is *descriptive* and coarsely grained. This allows to establish a link between a document, contained terms and domain (business) concepts, thus facilitating understanding of a service by its users.

In SOA there is a unidirectional real dependency between a consumer and a service: a consumer depends on the functionality provided by the service, whereas a service is independent from its consumers. This kind of a relation is also present in Client-Server architectures. Decoupling between a consumer and a service is increased when we simplify their interaction protocol. *Session-stateless service* is a design pattern that allows to achieve that.

Under *session-stateless* interaction we understand a *non-conversational* interaction between a service and a consumer. One way interaction is an example of a stateless interaction. However, we believe that the utility of such type of interaction is not sufficient for many application domains. Therefore, we extend the interaction to a single request-response interaction.

Session-stateless interaction contributes to the scalability of a service, simplifies communication with a service, facilitates its monitoring and recovery after failures. It also ensures that a service does not rely on a client to perform a complex sequence of actions to exploit the service's functionality. This implies a simpler interaction protocol and reduces artificial dependency.

Loose Coupling in Web Services

In Web Services loose coupling is attributed to *service discovery* and *late binding* as means to reduce the costs of artificial dependency between a service and a consumer. Via service discovery a consumer learns about the existence of a service capable of providing the required functionality. The result of service discovery is an identity of a service and a corresponding WSDL [37] description. The WSDL description provides not only an abstract interface but also binds it to concrete transport and message layers (HTTP and SOAP, for example). We consider service discovery to be an operational detail that is outside the scope of the Onto⇔SOA service model. We assume that a consumer knows about a service (via its schema) but how the consumer has obtained this knowledge is outside the scope of the core Onto⇔SOA approach. We also consider the transport and message layers to be an operational part of the connector upon which we do not elaborate in this chapter (in Chapter 5 we will return to this subject). This illustrates the main difference in the interpretation of loose coupling in the proposed service model and in Web Services: in the case latter we

reduce the costs of artificial dependency at the operational level only, whereas the former case we decrease artificial and real dependencies at the architectural level.

Knowledge-Oriented View on Loose Coupling

Given the introduced constraints we can interpret the communication between a service and its consumers from the Knowledge Representation perspective. Since the connectors are generic, all *application-specific semantics* must be expressed in descriptive messages that communicate an initial description of a domain from a consumer to a service. These messages specify *what is known to a consumer about its application domain* but not *how a service must process the available facts*. The consumer is unaware about how exactly a service processes a document.

A service acts as *a knowledge source* [38] that makes the application domain expertise available to a service consumer. A consumer assumes that a service will apply this expertise *to infer domain facts from the facts supplied by a consumer*. Since a service contains domain expertise (i.e. procedural knowledge), the schema effectively specifies a vocabulary (i.e. an ontology) required to utilize it. This view on a schema in the Onto⇔SOA service model enables a natural transition from a schema to an ontology that becomes an integral component of a service.

For example, in the Document Retrieval case in order to act as a knowledge source, a service has to provide a schema that enables consumers to describe *what* is known about the state of the domain. The consumer can use descriptive statements to specify facts such as:

1. There exists a concrete corpus C.
2. This corpus contains documents D0,..,Dn.
3. There is also a text query consisting of terms t1 and t2.

The consumer communicates these facts to the document retrieval service and expects it to infer new facts from the provided description, i.e. a list of documents matching the given query. The consumer does not know *how exactly* the service arrives to new facts (which would be the case with the RPC-style communication through Lucene API).

The proposed service model unifies the intent of the communication (inference of new facts) between the service and the consumer. The target service domain is described in the service ontology containing all application-specific concepts. The declarative (descriptive) character of the service ontology reduces the coupling between domain conceptualization

and a concrete implementation of the service. At the same time, by unifying the intent and assuming the presence of a service we can facilitate extraction of utility from the declarative service ontology. We elaborate on this ontology-oriented perspective in Chapter 3.

2.2.2 Domain Alignment

In [39] business alignment is succinctly defined as the delivery of the required results. In SOA services are often characterized as *business aligned* entities. Thus, services are regarded as software components that are (highly) *effective* within a target business (or application) domain.

More specifically, we understand *business alignment* as the property of a service that characterizes its ability to support, facilitate or enable business processes or meet business requirements. We translate business alignment into the more general *domain alignment* characteristic, that we define as *the ability of a service to have a direct relationship (support, facilitate, enable, etc) with most (ideally all) entities (processes, requirements, etc) in a target domain*. Domain alignment is beneficial not only for the effectiveness but also for the usability of a service:

- The effectiveness (the delivery of the required results) is improved by ensuring that a service operates within the boundaries of the target application domain.
- The usability is enhanced by eliminating the gap between a service interface and domain concepts, thus making the interface more understandable to users.

The extent to which a service is domain aligned can be defined as the degree of overlap between the concepts in the application domain and in the service interface. We assume that the perfect domain alignment is achieved when a service directly affects all concepts in a target application domain (Figure 2.5). We will consider any deviation from the perfect domain alignment as a misalignment. Several types of misalignment are depicted in fragments A, B, C and D of Figure 2.5.

The misalignment in the fragment **A** results from an application domain having a broader scope than supported by the service. This misalignment can be dealt with by either expanding the scope of the service or decomposing the application domain. For example, if we extend the Document Retrieval application domain to include ranking of matching documents then a service that retrieves documents but does not provide the ranking is considered misaligned to the service domain. To resolve this we can extend the functionality of the service to rank retrieved documents as well.

Fragment **B** depicts an application domain covered by several services. We consider this case to be a misalignment because each of the services is not perfectly aligned. Moreover, the coordination of multiple services requires a conceptual support that is likely to fall outside the application domain scope. This type of misalignment can be resolved by either merging the services into one or by decomposing the application domain. Domain decomposition is preferred if the services have no real dependency between each other. To give an example for the latter case, we can consider a domain of "document parsing and term analysis" and two services responsible for parsing and analysis. Each of these services is misaligned with the target service domain because it does not affect all domain concepts. Since the two services are independent the best way to resolve such misalignment is to decompose the service domain into two domains: "document parsing" and "term analysis". Each of the two services then becomes perfectly aligned to the corresponding application domain.

The misalignments in fragments **A** and **B** can be explained with too fine *granularity* of the service. A finer grained object is often considered more suitable for reuse because it is expected to fit to a larger number of application scenarios than a coarsely-grained object. However, a fine-grained object has to be combined with other objects to provide sufficient utility. This increases the number of dependencies, complicates interaction protocols between objects and makes the overall design more difficult to understand, thus hindering reuse. The right balance between the granularity of a component and its reusability is difficult to arrive to. We propose to use *domain alignment* as an indicator of the right degree of granularity of a service.

Fragment **C** shows a misalignment caused by a service exposing concepts alien to the application domain. This is often the case when a generic, configurable service is being reused in a new task/domain. This misalignment can be resolved by either extending the scope of the application domain to include the alien concepts or by encapsulating such concepts within a Facade [36] service. Which way is preferred depends on whether the increased complexity of the application domain incurred by the inclusion of new concepts still contributes to the effectiveness of such a service.

The misalignment **D** occurs if a service (interface) and an application domain share no concepts, implying that the service has no direct relation to that domain. However, this service may have an indirect relation through another service directly related to the domain. The resolution strategy for misalignment **C** can be employed in this case as well. However, extension of the application domain will be less desirable because of a greater conceptual

distance between the application domain and the service. To illustrate this misalignment we can consider the indexing service in the Document Retrieval domain. The concept of *index* falls outside the scope of the target application domain. However, the indexing service can be required to provide a real dependency to the document retrieval service. The indexing service is misaligned to the Document Retrieval domain. If we include the concept of *index* into the Document Retrieval domain then the users are likely to be confused because they do not expect *index* to be relevant to the task of retrieving documents. We, therefore, believe that the better way to resolve such misalignment is to encapsulate the indexing component into the document retrieval service, completely hiding it from consumers of the document retrieval service.

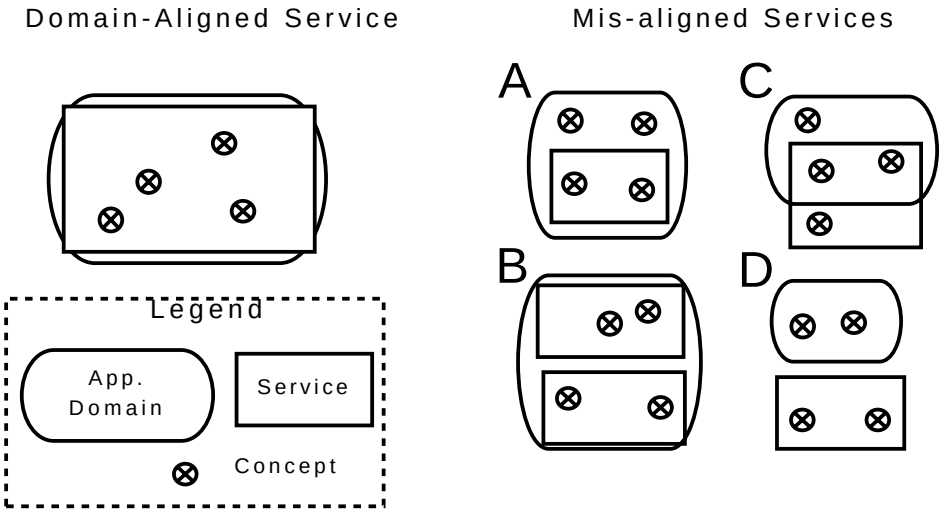

Figure 2.5: An illustration of the *perfect domain alignment* and four types of misalignment (A,B,C,D).

Figure 2.6 illustrates the alignment situation for the Document Retrieval domain and the interface domains of the Lucene objects required to implement the use case. We can see that the alignment situation is a composition of misalignments **C** and **D**. We can employ the Facade service to effectively resolve these types of misalignment by hiding the irrelevant concepts (index, analyzer, etc) and exposing domain-aligned concepts only. We will elaborate on this in Section 2.4.

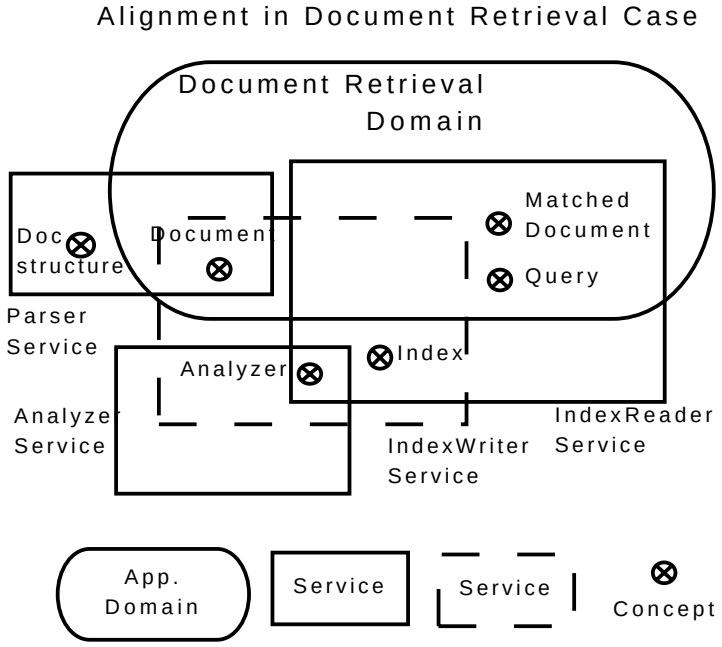

Figure 2.6: Domain alignment in the Document Retrieval case.

We believe there is a connection between domain alignment of a component (a service, an object, etc) and its granularity. A *coarsely-grained* component encapsulates complex functionality that is likely to have a direct connection to the application domain. From our definition of domain alignment it follows that a perfectly aligned service is the most coarsely-grained component in that application domain. Inversely, the more coarsely-grained a component is – the closer it is to perfect alignment to the target domain, and therefore, the better it is suited to become a service.

The connection between domain alignment and granularity of a processing component implies that it should be always possible to define exactly one *the most coarsely-grained processing component* for a target service domain. This allows us to formulate a constraint to induce the domain alignment characteristic in the Onto⇔SOA service model: **For a given domain, a number of Onto⇔SOA services should be reduced as much as possible, preferably to one**.

Omnipotence of a service is a direct consequence of the strict enforcement of this domain alignment characteristics. Omnipotence is ascribed to a self-contained service that

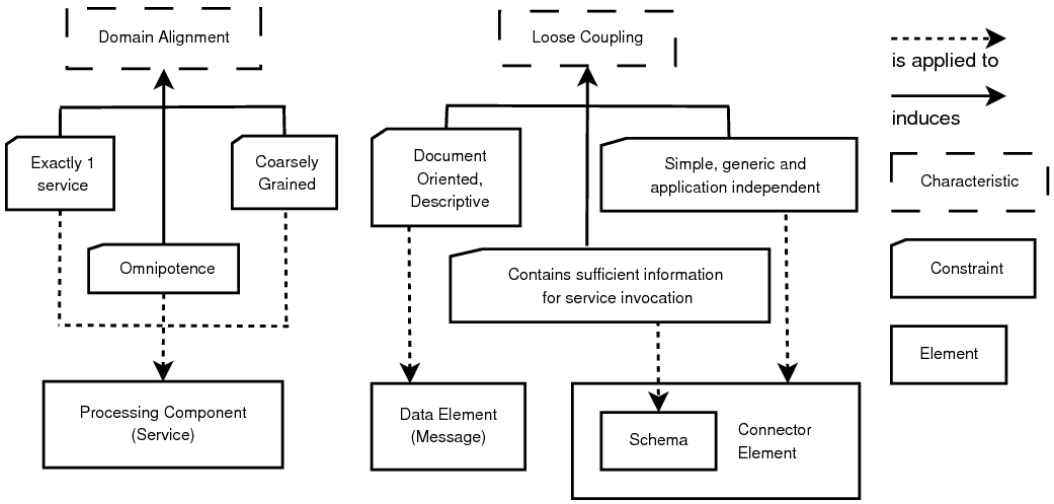

Figure 2.7: The proposed service model: constraints, architectural elements affected by them and induced characteristics.

requires no other services to provide its functionality. A service is omnipotent if it has no real dependency on other services. By favoring omnipotent services we reduce real dependencies between services, and thus contribute to loose coupling. Omnipotence does not restrict interaction between services that belong to different domains (or Onto⇔SOA service models). Nevertheless, such cross architectural interaction is outside the scope of the core Onto⇔SOA framework.

In the Document Retrieval case the IndexWriter and Analyzer components are well aligned to their actual application domains (document indexing and natural language processing). However, none of these components is directly related to the Document Retrieval domain, thus does not fit well to become a service in that domain. We arrive to the same conclusion by observing that none of the components is *omnipotent*, i.e. sufficient to solve the Document Retrieval case on its own.

Figure 2.7 summarizes the introduced constraints, architectural elements effected by them and the induced characteristics. We assume that an ontology is a domain-aligned entity (it fully specifies concepts that exist in a target application domain), and therefore has a potential to further support domain alignment of services. In the next section we describe how to realize that potential.

2.3 Onto⇔SOA: Ontology-enabled Services

The restricted service model described in the previous section provides a foundation for Onto⇔SOA. In this section we extend this service model with ontologies to define the core of the Onto⇔SOA framework. We assume that an ontology *is capable of supplying a domain-aligned conceptualization*, and thus has the potential to enhance the corresponding service characteristic. We elaborate on other distinguishing characteristics of ontologies in Chapter 3.1. To incorporate ontologies into the service model we introduce a direct exchange of ontology-based, document-oriented messages between a service and a consumer. We will also supply additional constraints on the connector and data elements (messages) of a service-oriented architecture.

In Onto⇔SOA we propose to employ ontological conceptualization in a service schema. To achieve this we require to design a service schema by using ontology engineering or other knowledge engineering methods, ensuring that the resulting conceptualization is domain aligned. While designing a schema existing concepts from external ontologies may be reused as well as new ones created.

Consequently, we require that an ontology language is used to express a service schema and communicated messages. In Onto⇔SOA we assume that both a service and its consumer are aware about the ontology underlying the messages. In this way we abstract away the syntactical and structural aspects of messages and focus on the conceptual aspects defined in an ontology. We will refer to a schema designed as an ontology and expressed using an ontology language as a *service ontology* – a specification of a conceptual document-oriented interface to a domain-aligned, loosely-coupled and omnipotent service.

Onto⇔SOA emphasizes the role of a service as a Knowledge source. This implies that:

- a consumer employs a service ontology to present facts that describe an incomplete domain situation (e.g., a document collection and a certain text query);
- this description of an initial domain situation is sent to a service that contains domain expertise in the form of procedural knowledge (e.g., a Lucene-based implementation of a document retrieval algorithm);
- the service applies its domain knowledge to extend the domain situation with new facts (e.g., a list of retrieved documents) and sends it back to the consumer.

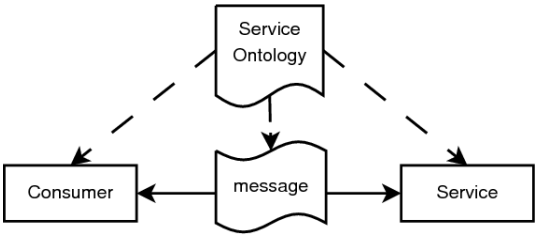

Figure 2.8: Onto⇔SOA elements.

2.3.1 Service Ontology

An ontology language provides a service ontology with a unified syntax and structure as well as *a minimal set of conceptual primitives*. The primitives must be simple, aligned to target application domain(s) and *readily understandable* to the domain users. We do not require the language to have a well-defined formal semantics because in Onto⇔SOA we ground the meaning of concepts in a service and the user's domain understanding.

When choosing an ontology language for an Onto⇔SOA service we seek to fulfill the constraints of the service model and to further enhance the domain alignment and loose coupling characteristics. This implies that an ontology language should:

- provide a simple yet flexible structure (i.e. data-model) that can be related to conceptual primitives widely accepted by experts in a range of target service domains. We believe that graph-based data models (e.g., directed labeled graphs – DLG) have proven to be generic and flexible enough in a wide range of domains (semantic networks, bond graphs [40], etc). The elements (nodes and arcs) of the DLG data model underlying the RDF/S languages are translated into primitives such as subject, predicate and object that are both expressive and understandable by experts in a variety of domains.

- not an introduce expensive artificial dependency. This implies that a language specification should be simple and publicly accessible allowing anyone to employ this language with as little overhead as possible. Furthermore, a language should not be tied to any specific task. The only task a language is required to support is expression of facts. Everything beside that we treat as an artificial dependency that should be

avoided in Onto⇔SOA service ontologies.

In a service ontology we can distinguish two types of concepts:

1. *conceptual primitives* that provide basic modeling building blocks not affecting service behavior. So far, in all use cases we have employed subject, predicate, object as conceptual primitives. It is equally possible to employ other primitives such as *Entity - Relation* as long as there is a sufficient shared understanding of their meaning among the target domain users. This shared understanding of conceptual primitives is established outside Onto⇔SOA, for example by means of standards, formal specifications or background domain knowledge.

2. *domain concepts* that affect service behavior. The semantics of these domain concepts is grounded in an Onto⇔SOA service. These concepts constitute an interface to a service, needed to describe domain facts. Unlike with conceptual primitives, Onto⇔SOA can facilitate shared understanding of domain concepts: in Chapter 3 we will transform Onto⇔SOA into a mechanism capable of grounding *application semantics* of domain concepts in a service.

A service ontology always contains domain concepts because they are required to interface with the domain knowledge contained in the service. At the same time, we believe that conceptual primitives are optional. Or, in other words, the role of conceptual primitives can be played by domain concepts. Conceptual primitives can be seen as an auxiliary means for modeling service domains that are neither simple enough to be specified using a few domain-specific terms (e.g., document, corpora, query, etc) nor developed enough to contain own primitives.

Onto⇔SOA is an abstract framework. In order to operationalize it we have filled in the missing details and devised *MoRe* [41] – an extension of Onto⇔SOA that employs the RDF/S languages [13, 14] to express a schema of a document-oriented HTTP-based service. We dedicate Chapter 5 to a detailed description of the *MoRe* framework.

2.3.2 Reusing Service Ontologies

In Onto⇔SOA our focus is on usability of a service. The main step in designing an Onto⇔SOA service is to identify and scope a domain of a service. At this step we can consider existing services and their relations to the target domain. If expertise contained in existing services can be employed in the target domain then there is an opportunity for

service reuse. Onto⇔SOA opens up the possibility to analyze the properties of such reuse from the *ontological* perspective.

Internally an Onto⇔SOA service can have an arbitrary architecture which can, in turn, contain other service(s). We consider this kind of composition to be an implementation detail that should not be exposed via the service ontology. Nevertheless, if an internal service architecture contains ontologies, we would like to investigate *what are the relationships between the service ontology and external ontologies (such as other service ontologies or application-independent domain ontologies)?*

In Onto⇔SOA we assume that there is a 1-to-1 relation between a service and its service ontology. This acknowledges the underlying assumption that *each service exists in a unique context that determines the meaning of the associated ontological concepts*. In other words, our default assumption is that, concepts in a service ontology are unique (exclusively belong to the corresponding service). If we intend to reuse a concept across application domains, we have to guarantee consistency of interpretation of this concept. This means that ontologies of services employed internally *by default are not exposed* through a service ontology of an enclosing service (even if it seems natural to expose some of the concepts).

For example, let us assume that the Document Retrieval service can internally reuse the Analyzer and Parser services. Such internal reuse is not visible (does not take place) from the Onto⇔SOA viewpoint. To determine the effect of this internal reuse we have to determine conditions under which the Document Retrieval service ontology will expose some of the concepts borrowed from the Analyzer and Parser ontologies. If we choose to design an exclusive ontology for the Document Retrieval service then no internal concepts are exposed, and therefore, no reuse takes place (from the Onto⇔SOA perspective).

We assume that concepts from internal service ontologies may be exposed in a service ontology if there is means to control the environment where these services (e.g., the Document Retrieval, Analyzer and Parser) are employed. We can distinguish at least three ways to achieve this control:

1. A domain ontology can employ formal semantics to establish consistent interpretation of concepts within a certain logical framework. Such an ontology can be shared between several Onto⇔SOA services, and in this way an indirect reuse of concepts from service ontologies will take place.

2. If a target application domain itself provides means to establish a consistent interpretation of concepts then domains that include such an application domain can rely on the provided consistency.

3. Some control is achieved by the fact that the services are designed given the same set of assumption. This happens if, for example, the services are designed by the same team with service reuse in mind.

Applying the first scenario to the Document Retrieval case we could rely on "Linguistic" ontology containing concepts such as *term* and *document*. Both Document Retrieval and Parser services could then commit to this ontology and reuse the concept of *document*.

To illustrate the second scenario, let us assume that the Document Retrieval domain belongs to a broader domain of Document Management. If we assume that the Document Management domain is a well-established one, then this domain should be capable of facilitating consistent interpretation of its concepts. This can be achieved through, for example, international or industry-wide standards. Therefore, we can rely on this common foundation to supply concepts unambiguously understood across sub-domains of Document Management.

In the third scenario there would be no shared formal ontology. Instead, we would introduce the *document* concept in the Parser service ontology keeping in mind its future reuse in other services we develop. The Document Retrieval service would be such a service. Since we control both services, we can directly reuse the *document* concept in the Document Retrieval service. In this example, no concept from the Analyzer domain is reused in the Document Retrieval service because concepts such as *term*, *stem*, *lemma* etc are not aligned with the Document Retrieval application domain.

2.4 Solution to the Use Case

In this section we describe the design of an Onto⇔SOA-based solution for the Document Retrieval case introduced in Section 2.1. The solution has been implemented using the *MoRe* framework described in detail in Chapter 5. In the following sections we elaborate on the design of the two major Onto⇔SOA artifacts: the service (Section 2.4.1) and the service ontology (Section 2.4.2).

2.4.1 Document Retrieval Service

As introduced in Section 2.1, the application domain of the Document Retrieval case consists of *a corpus* containing a number of *documents*. The user provides *a query* that is used to find a set of *matching documents*. To design a document retrieval service we first consider components of the Lucene API shown on Figure 2.1. By applying the Onto⇔SOA

constraints we determine that:

- None of the components is well aligned to the Document Retrieval domain. The IndexWriter and IndexSearcher objects require an *index* – a concept not present in the target application domain. Neither Parser nor Analyzer perform functions directly related to the Document Retrieval domain. We illustrated this mis-alignment in Figure 2.6.

- The components are of the same granularity and depend on other components. This does not allow distinguishing a single, omnipotent component that can provide the required service to a consumer: the consumer has to interact with at least two components (IndexWriter and IndexSearcher) to realize the document retrieval functionality.

- To find documents matching a given query the user has to follow a rather complex interaction protocol: the interaction with IndexWriter and IndexSearcher must be coordinated using consistent identifiers of the index and Analyzer. Such coordination is likely to require a stateful session to maintain these identifiers across component invocations that must also be properly ordered. According to Onto⇔SOA this represents an artificial dependency undermining loose coupling.

The mis-alignment between the Lucene components and the target Document Retrieval domain causes the service schema to expose concepts (an index, an analyzer, a parser, etc) alien to that domain. These concepts are forced onto a consumer. They unnecessarily complicate the document retrieval task, compromise loose coupling and domain alignment by exposing implementation details, and ultimately hinder usability of the service.

As stated before, we consider the index to be an implementation detail irrelevant to the functionality of the Document Retrieval service. The sole *non-functional* purpose of the index is to contain precomputed data to speed up the retrieval process. Even if the index contains data relevant to a consumer (e.g., term and document frequencies) then only the associated concepts should be exposed, but not the index itself. In such a case the service domain should be redefined accordingly to include the concepts of term and document frequencies. This would effectively result in an application domain distinct from the Document Retrieval domain as defined in this use case and, therefore in a service different from the Document Retrieval service.

To fulfill these constraints we can employ the Facade pattern [36] to design a service as a component with a domain-aligned and loosely-coupled interface. This interface confronts the user with the concepts from the Document Retrieval application domain only,

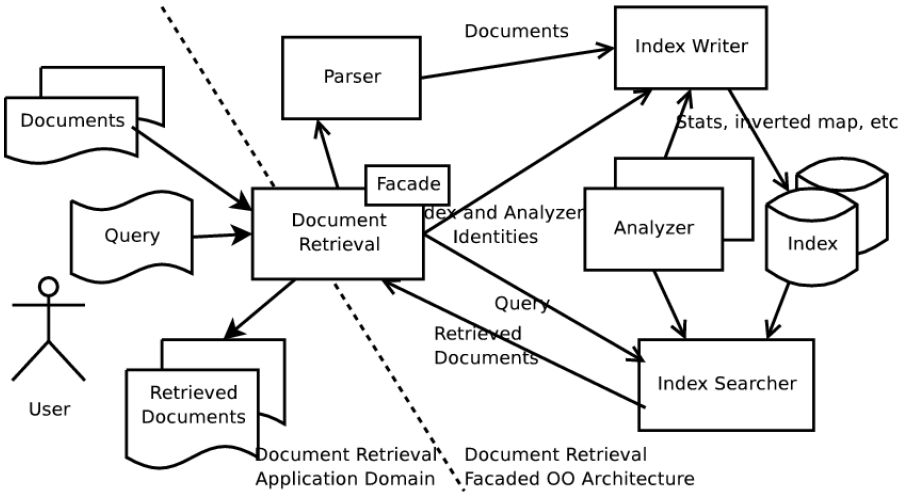

Figure 2.9: An Object-Oriented architecture with a Facade object for the Document Retrieval case.

while hiding the peculiarities of Lucene's approach to document retrieval (Figure 2.9). The Document Retrieval Facade component meets the Onto⇔SOA constraints:

1. It is well aligned to the target domain: its interface exposes only concepts that occur in the Document Retrieval domain.

2. It is the most coarsely-grained component because it provides the required functionality and encapsulates a number of finer-grained components (IndexWriter, Analyzer, etc). It is omnipotent: it depends on no other services to provide the required functionality.

3. It does not require a stateful session: neither the index nor the analyzer concepts are exposed, thus there is no need to maintain their identities across service invocations.

We have employed the *MoRe* framework to implement the Facade component as a domain-aligned, session-stateless, omnipotent and document-oriented service. The service accepts an RDF description of an initial situation in the Document Retrieval domain and returns another RDF document describing the situation extended with inferred facts. The terminology for both types of documents is defined in the Document Retrieval service ontology expressed in RDFS. We describe this service ontology in the next section.

Concept	Description	Provided by
DocumentRetrieval	the top-level container for the domain	consumer
– hasCorpus	points to a document collection	consumer
– hasQuery	points to a query to be matched against corpus documents	consumer
– hasRetrievedDocuments	points to a solution of a DR problem	service
Corpus	contains a collection of documents	consumer
– containsDocument	points to a document that belongs to this corpus	consumer
Query	represents a query	consumer
– hasQueryString	contains a literal value with a query	consumer
Document		consumer
– hasURL	contains an URL of a document	consumer
RetrievedDocuments	contains documents that match the given query	service
– hasRetrievedDocument	points to a retrieved document that belongs to a solution	service
RetrievedDocument	represents a matched document	service
– hasDocument	points to a document from the problem Corpus	service
– hasScore	contains match score for the retrieved document	service

Table 2.1: The Document Retrieval service ontology.

2.4.2 Document Retrieval Service Ontology

The Document Retrieval service ontology specifies a document-oriented interface to the Document Retrieval service. This service ontology provides conceptualization required to describe an instance of the Document Retrieval case. The conceptualization consists of a number of classes and properties summarized in Table 2.1.

The Document Retrieval service ontology is aligned to the Document Retrieval domain: all concepts can be readily found in the original domain definition. There is no conceptual gap between the consumer's view of the service domain (this was the basis for the definition of the Document Retrieval domain in the first place) and the interface to the service.

A consumer employs the Document Retrieval service ontology to describe an initial situation in the target domain: a corpus consisting of a collection of documents and a text query (the left-hand side of Figure 2.2). The service, in turn, will use this service ontology to express the facts inferred from that initial situation; i.e., a collection of ranked documents that match the query (the right-hand side Figure 2.2).

```
-DocumentRetrieval
 hasQuery aQuery
 hasCorpus aCorpus

-aQuery
 type Query
 hasQueryString ``dairy''

-aCorpus
 type Corpus
 containsDocument document1
 containsDocument document2
 containsDocument document3

-document1
 type Document
 hasURL ``.../index.htm''

-document2
 type Document
 hasURL ``.../OntoSOA.pdf''

-document3
 type Document
 hasURL ``.../Quality.pdf''
```

```
-DocumentRetrieval
 hasRetrievedDocuments retrievedDocs

-retrievedDocuments
 type RetrievedDocuments
 hasRetrievedDocument rDocument1
 hasRetrievedDocument rDocument2

-rDocument1
 type RetrievedDocument
 hasDocument document1
 hasScore 0.65255654

-rDocument2
 type RetrievedDocument
 hasDocument document2
 hasScore 0.9764538
```

Table 2.2: An example of the initial Document Retrieval domain description (the left-hand side) and the extension to it (the right-hand side) inferred by the Document Retrieval service.

2.5 Related Work

In Onto⇔SOA we employ a service model that is more restricted than the one that is commonly considered in the fields of Web Services and its semantic extension – Semantic Web Services. In the consequent subsections we outline the main differences between the two approaches and the most important implications of these differences.

2.5.1 Web Services

The term "Web Services" most often refers to WSDL/SOAP-based services, that can be seen as the most popular approach and the standard way to implement SOA on the Web. WSDL (Web Service Description Language) is an XML-based language and a corresponding description framework for Web Services [37]. WSDL is primarily used for service invocation. The SOAP [25] protocol provides for a standard to structure messages that can be carried over a variety of transport protocols, with HTTP being most frequently used.

WSDL/SOAP services are often implemented by wrapping existing software components with a SOAP-based messaging layer. This makes such components accessible via the Web to a wide range of consumers. Presently, development of Web Services is primarily concerned with how to *implement* a service by means of WSDL/SOAP standards, and is not concerned with how to *design* a service. What kind of software components suit best (or do not suit at all) to be transformed into a service is not restricted by the standard WSDL/SOAP approach.

We consider Web Services and the Onto⇔SOA service model to occupy two different levels of Service-Oriented Software Engineering. We consider Web Services to belong to the implementation level. Web Services are defined as a collection of specifications (and standards) that define a communication layer between services and their consumers. On the other hand we position the Onto⇔SOA service model at the design (architectural) level. It expresses a set of design constraints and guidelines that induce the targeted characteristics of domain alignment and loose coupling.

The two approaches interact at the border between the implementation and design levels. The implementation means defined in Web Services affect the way service-oriented software is designed. In our service model we explicitly limit the choice of implementation methods to those that are capable of meeting the proposed constraints. This interaction between Web Services and the Onto⇔SOA service model allows us to compare them.

In the Onto⇔SOA service model we do not assume that just any software component can be transformed into a service, regardless of its application (i.e., business) context. We require that to be considered as a service, a software component must be sufficiently well aligned with the functionality and concepts observed in the target domain. For example, in the Document Retrieval case such a component must communicate in terms of *documents* and *search query* only. IndexWriter cannot become a service associated with this interface. On the other hand, in the domain of indexes and documents IndexWriter could be the appropriate candidate for a service.

One of the most notable differences between Web Services and the Onto⇔SOA service model is in the favored messaging style. In Web Services, only the RPC (Remote Procedure Call) communication style was initially supported[4]. With SOAP 1.2 the RPC style has become optional and the *document-oriented* flavor was introduced. Nevertheless, SOAP-RPC approach still attracts most of the attention in the Web Services community.

From the Onto⇔SOA perspective, application of RPC introduces an artificial dependency between a consumer and a service hindering loose coupling between them. This dependency results from the requirements of the RPC interaction protocol a consumer must be aware of: the name of the operation, its input arguments and the external effects of invocation. In many application domains this artificial dependency can be reduced by employing the *document-oriented* communication style.

Furthermore, the RPC messages tend to be *prescriptive* rather than *descriptive*. With an RPC message a consumer commands a service *how* to solve a problem rather than describing *what* is to be solved. The prescriptive nature of RPC Web services often leads to stateful sessions as a way to coordinate multiple commands to a service. A stateful session leads to a complex interaction protocol that further strengthens the artificial dependency between a service and a consumer.

Web Services require a significant number of conceptual and architectural elements [42]. A considerable part of them is devoted to the RPC communication style that is to be avoided in Onto⇔SOA. Moreover, in many practical cases a *document-oriented* invocation that is performed via a well-established communication interface such as HTTP is enough to define an operational SOA. In Chapter 5 we will discuss in more detail the benefits of such *lighter* approach to services.

2.5.2 Semantic Web Services

The state of the art Semantic Web Services approaches (SWS) [43] such as OWL-S [28] and WSMO [29] employ ontologies to provide *formal* ontology-based descriptions of Web Services to automate discovery, invocation and composition of such services. Service models employed in SWS closely follow the Web Services model with a service consisting of a number of *operations* that have *inputs*, *outputs*, *preconditions*, *effects*, etc. SWS approaches propose formal service description frameworks (often expressed as ontologies) that can be combined with external ontologies to build a description of a concrete service. Such formal

[4] http://www.xml.com/pub/a/ws/2001/04/04/soap.html

description can then be employed to, for example, automatically discover services capable of providing a sought for functionality or even to automatically combine a number of services (or operations) to realize such functionality.

SWS employ ontologies to enrich *implementation* artifacts (e.g., a WSDL description) of Web Services with formal semantics. Consequently, ontologies employed in SWS focus on *implementation aspects* of a service. In contrast, in Onto⇔SOA we propose to employ ontologies to improve service characteristics through design. This makes the two approaches complementary rather than competitive. However, as we will demonstrate with the Blackboard-based composition mechanism (Chapter 4), explicit design decisions made according to the proposed guidelines can significantly simplify implementation issues of service composition.

From the Software Architecture perspective, SWS can be seen as a complex connector between a service and its consumer. Internally, the connector relies on ontologies to match consumer requests to available services, to ensure conceptual compatibility of messages, etc. However, the service still operates on data-level requests (SOAP-RPC in most cases) rather than conceptual, ontology-based content. Although the WSMO approach has the potential to use ontologies directly, in practice the SWS approaches rarely address a direct exchange of ontology-based messages.

As in traditional Web Services, in SWS there is a tendency (more visible in OWL-S than in WSMO) to disregard the characteristics of a service and, thus, to assume that any software component can be formally described to support the target tasks. This results in fairly extensive frameworks that require a large amount of meta-data to describe a service. Moreover, since the internal properties of a service are disregarded, it is difficult to provide guidelines on how to translate between the internal service properties and a corresponding SWS model, i.e. how to design a service or provide a meta-data description for an existing one.

The tasks targeted by SWS (discovery, composition, orchestration, etc) are of different natures and, therefore it is very challenging to cover all of them within a single formal approach (e.g., by a single ontology language with fixed formal semantics). For example, in order to support *automated service discovery* an ontology language capable of describing (and providing corresponding reasoning support) about hierarchically-organized domain concepts may be sufficient. On the other hand, automated service composition requires a language capable of describing workflows, pre- and post-conditions of operations, etc. By

targeting such distinct tasks within a single language the end result will be of high complexity and will introduce a significant entry barrier.

We argue that such high level of complexity is not inherently required to enable integration of ontologies and services. In Onto⇔SOA we focus on applying the ontology primarily for the invocation task. By means of the restricted service model we constrain the internal properties of a service. This simplifies the model of a service, reduces the amount of metadata required to describe it and provides guidelines on design of Onto⇔SOA services.

In the remainder of this section we employ OWL-S [44] to further illustrate the differences between SWS and Onto⇔SOA. OWL-S defines three models to describe the implementation of a Web service from different perspectives:

- *Process* models the internal details of a service in terms of input/output parameters, preconditions and effects. The purpose of *Process* is to assist in service composition.
- *Grounding* maps domain concepts to data-types (usually the XML ones) required to express requests to a service. The purpose of *Grounding* is to facilitate service invocation.
- *Profile* describes the functionality of a service in domain terms. Its main purpose is to assist service discovery.

In Onto⇔SOA we focus on service invocation. We hide internal non-domain-aligned details of a service from the consumer. Moreover, we focus on architectures consisting of an omnipotent service and we do not address inter-SOA interactions. All this eliminates the need for the *Process* model in Onto⇔SOA.

Onto⇔SOA services directly accept messages expressed in an ontology language. In other words, a consumer and a service interact via a conceptual domain description rather than lower level data structures. In Onto⇔SOA the data model of all messages is uniform (e.g., the RDF data model in *MoRe* described in Chapter 5) and all application-specific aspects are captured in domain-aligned concepts. This eliminates the need for *Grounding* in Onto⇔SOA.

There is a certain similarity in the purpose of OWL-S Profile and an Onto⇔SOA service ontology: they both can facilitate service discovery. Still, there is a significant difference in how this purpose is realized. *Profile* describes a service in terms of, usually externally-defined, domain ontologies. In Onto⇔SOA the default assumption is that the service ontology is dedicated to the service domain. The service ontology may be employed exclusively

with this service because we assume that the meaning of the ontological concepts (application semantics introduced in Chapter 3) is defined solely by the service. The service ontology may be directly related (e.g., by means of ontology mapping [45] or concept import) to concepts from an externally defined ontology. However, this should be done with care because of the possible differences between the pragmatic service-oriented application semantics in Onto⇔SOA and the formal semantics defined in an external ontology.

2.6 Discussion

In this section we elaborate on the relations between usability and reusability of Ontology-enabled Services and more traditional software components. Also we discuss in what type of application domains Onto⇔SOA can be most effective.

2.6.1 Balancing Reusability and Usability

In Onto⇔SOA we aim to improve domain alignment and loose coupling of services. These characteristics primarily represent the viewpoint of a service consumer. The proposed constraints and guidelines, however, are aimed at service designers. With these constraints we strive to reduce the impact of the implementation concerns on how efficiently and effectively a service can be utilized.

We believe that very often the implementation concerns over-emphasize engineering (non functional) properties of a service at the cost of effectiveness of actual utility and usability delivered to its consumers. The internal properties of software artifacts such as reusability (of code or components) are focused on by software engineers in an attempt to reduce the development effort. This however, comes at the cost of additional complexity and effort required to create reusable components and then to actually use them.

The increased internal complexity must be contained to not inhibit usability of an end product. In desktop software this is achieved with user-friendly human-computer interfaces designed to reduce effort required from users to learn and operate the product. By reusing what the users already know and expressing the interface in terms familiar to them we can make an application more easily (intuitively) understandable to the users. Although services are intended for programmatic consumption, with Onto⇔SOA we propose to employ ontologies to design user-oriented (i.e., domain aligned) service interfaces that consist of concepts from the domain of the service consumers. By doing so we re-emphasize usability and utility of a service and make it more attractive to its consumers.

Inevitably, there is a trade-off between usability (how easy it is to employ component's utility) and reusability (in how many distinct and unforeseen application domains a component can be applied). For example, application-specific components are the easiest ones to use. However, they are limited to a single application domain and, therefore are of limited reusability. On the other hand, highly reusable components (frameworks, libraries, etc) require special knowledge to apply them in different scenarios, and therefore have lower usability.

Another example of this trade-off is the balance between fine- and coarsely-grained objects. Fine-grained objects offer increased possibility for their reuse. However, when reused fine-grained components require extra effort to manage dependencies between them, to design complex interaction protocols and, ultimately, to understand the whole system.

Improved reusability by means of fine granularity is an example of what we can refer to as *software* reuse. In this type of reuse the main goal is to employ existing software element (function, class, component, etc) in as many places as possible to reduce development effort. In many cases usability of software components has a lower priority than their reuse and is readily sacrificed to increase the number of reuse opportunities. In Onto⇔SOA however, we make usability our main priority and we believe that reusability of a service follows from it. This demonstrates one of the crucial differences between traditional software components and Onto⇔SOA services.

Unlike a more traditional software component, an Onto⇔SOA service is not designed to be reused in as large number of software systems as possible. Instead, a service is designed to be usable in a target application (business) domain, thus effectively supporting it. Figure 2.10 illustrates a scenario in which software reuse can take place during the design of an Onto⇔SOA service. However, our main goal is not to maximize reuse but to build a service which is well aligned to the target application domain. Such a service will in turn provide domain-aligned, and thus efficiently utilizable, functionality to a business application (developers).

Reuse of an Onto⇔SOA service can take place across a probably much smaller number of application domains. This type of reuse is not so much about reducing the development effort but about discovering overlapping functionality in application domains and 'outsourcing' it to dedicated services. Figure 2.11 demonstrates a scenario in which two business processes include another sub-process. If we design an Onto⇔SOA service well aligned to that sub-process, then this service can be reused in business applications dedicated to the other two processes. Therefore, by facilitating domain-alignment of services Onto⇔SOA

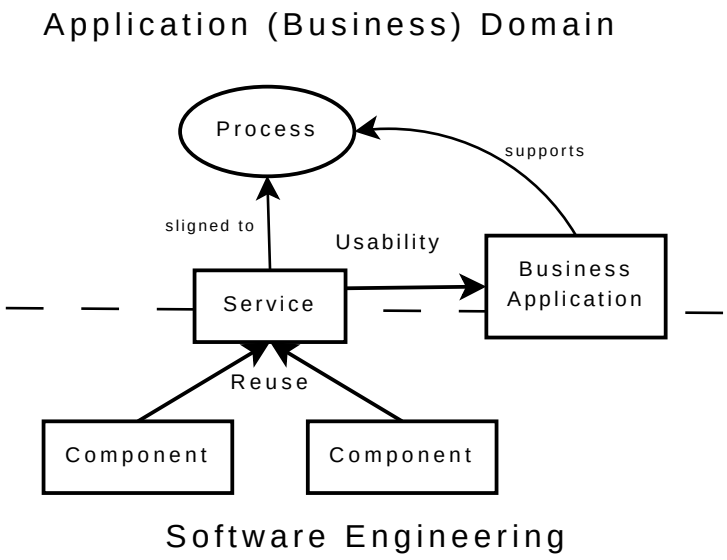

Figure 2.10: Usability in Onto⇔SOA services vs software reuse. Software reuse can take place during construction of an Onto⇔SOA service. However, the service itself is not designed to maximize reuse but rather to effectively support target domain (business) processes and to deliver its functionality in a usable way to business application (developers).

ensures that reuse that already takes place in an application domain can be exploited by business application developers.

2.6.2 Serviceable Domains

Due to our focus on the domain alignment characteristic, the notion of application domain becomes crucial to Onto⇔SOA. We require to scope a service domain thoroughly, otherwise neither a service nor its schema can be defined. We employ constraints to design a service such that it directly relates to the target application domain. With this we aim at enforcing that service provides the exact functionality required by the application domain and does not go beyond that.

The degree to which the proposed guidelines can be followed (and constraints met) in

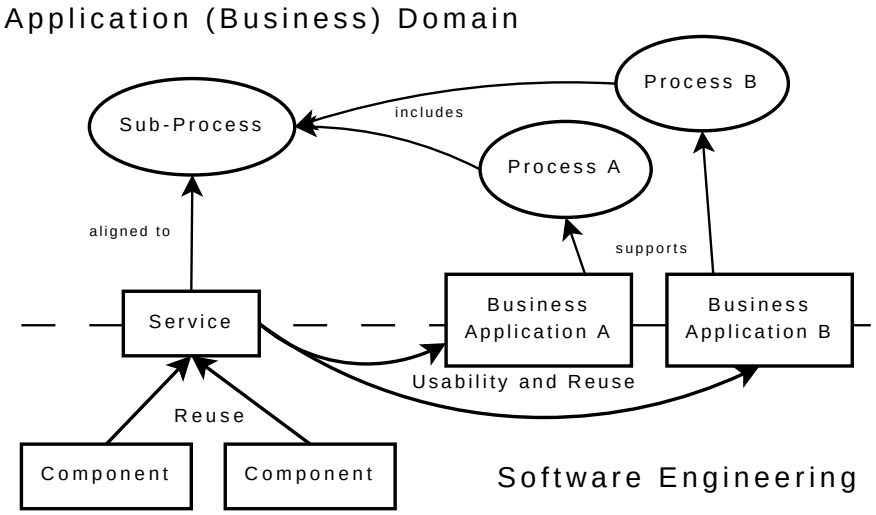

Figure 2.11: Reuse in Onto⇔SOA services. By focusing on domain alignment Onto⇔SOA becomes open to a 'natural' reuse of services. This type of reuse takes place when a target application domain already contains (or can be decomposed) into sub-domains (sub-processes) exploitable in multiple business applications.

a given application domain indicates how effectively that domain can be supported with an Onto⇔SOA service. In other words, we believe that these constraints can characterize an application domain or business process with respect to how well (if at all) domain alignment and loose coupling can be supported in that domain.

By restricting services to omnipotent and session stateless entities we aim to reduce real dependency, however by doing so we also limit the range of application domains where we can employ such services. Consequently, there are application domains to which Onto⇔SOA should not be applied.

If impossible to follow, the design guidelines can indicate directions in which the original application domain has to be modified (reshaped or decomposed) to facilitate design of a loosely-coupled and domain-aligned architecture. We believe, that domain decomposition guided by Onto⇔SOA constraints can provide a valuable aid in, for example, business process analysis allowing to distinguish sub-process most suitable for 'outsourcing' into services.

We propose to use the notion of *a serviceable domain* to refer to an application domain that can sufficiently well meet the Onto⇔SOA constraints. Thus, we define a *serviceable domain* as a *distinguishable*, *independent* and *established* application domain that consists of a *limited* set of entities required to support a *single task* (or a coherent collection of *dependent tasks*) exploitable through a *simple* protocol.

A service ontology is, therefore, an ontology of a serviceable domain, an Onto⇔SOA service is an encapsulation of expertise of a serviceable domain.

A serviceable domain:

- is established when most of its characteristics are well-understood and can be acquired directly from experts or indirectly from other sources;
- is distinguishable and independent when the scope of the domain can be defined precisely, and when the domain has as few as possible dependencies with other domains;
- can be described with a finite (limited) set of entities that will constitute the service interface;
- contains expertise that can be represented as a single/atomic task. This expertise can be exploited through a simple protocol, and has value of its own and is likely to be transferable/applicable to other application domains.

Onto⇔SOA should not be applied in a domain that neither has a dedicated utility nor a target user group. For example, a programming language API such as Lucene covers a collection of dependent tasks that are intended to be employed in a wide range of application scenarios, therefore, the application domain covered by it is not serviceable.

Another example, is a typical research project in its early stage at which there is no (yet) clear understanding of the target application scenario (utility to be delivered). In this case, the prototypes should be developed and experimented upon, after which the application scenario(s) can be defined outlining a serviceable domain.

Yet another example of domains that are not serviceable is so called application-independent domains. A traditional domain ontology is often intended to be used as widely as possible and, therefore has little utility of its own unlike a service ontology. A specific application context can supply utility to a domain ontology (or some parts of it).

2.7 Conclusions

In this chapter we have addressed the central question of this dissertation: *"How can ontologies and services be integrated into a framework facilitating application of knowledge?"*. As the answer to this question we have introduced Onto⇔SOA – a framework that integrates Service-Oriented Architectures and ontologies to emphasize usability of services.

The proposed framework is based on a restricted service model that constrains the internal properties of a service to induce the *domain alignment* and *loose coupling* characteristics. These constraints simplify the model of a service, reduce the amount of meta-data required to describe it, and allow to provide guidelines on design of Onto⇔SOA services.

Onto⇔SOA builds upon the proposed service model to address a direct exchange of ontology-based messages between a *document-oriented* service and its consumer. The framework employs an ontology as a *service schema* (referred to as a *service ontology*) that describes a *domain-aligned* interface to a service.

We have demonstrated the Onto⇔SOA-based design of services and corresponding *service ontologies* for the Document Retrieval use case and elaborated on the main differences between the proposed approach and Semantic Web Services.

Although we intentionally restrict Onto⇔SOA to the service invocation task only, we believe that the proposed constraints will also facilitate other service-related tasks such as composition, discovery, etc. In Chapter 4 we support this claim (and answer the corresponding research question) by demonstrating how Onto⇔SOA enables effective service composition using a Blackboard-based mechanism.

Chapter 3

Service-enabled Ontologies

The integration of services and ontologies can be viewed from two perspectives. In the previous chapter we provided the service-oriented perspective. In this chapter we introduce the ontology-oriented perspective that will be used to answer the research question: *"How can we attach a service to an ontology and what does this imply for ontologies?"*. In this perspective we shift the focus from services to ontologies and re-interpret Onto⇔SOA as a mechanism that allows to attach a service to an ontology making it *service enabled*. Such service-enabled ontologies are capable of capturing *procedural* (behavioral) domain aspects in a way that makes them readily-exploitable in software systems. We argue that the proposed service-enablement of ontologies not only offers greater flexibility for ontology engineering but also enables provision of readily-exploitable utility to ontology users.

◇

Maksym Korotkiy and Jan L. Top: Onto⇔SOA: From Ontology-enabled SOA to Service-enabled Ontologies. In proceedings of International Conference on Internet and Web Application and Services (ICIW'06). Guadeloupe, 2006.

Maksym Korotkiy and Jan L. Top: MoRe Semantic Web Applications. In proceedings of the End-User Aspects of the Semantic Web Workshop. European Semantic Web Conference. Crete, 2005.

◇

In Onto⇔SOA we integrate an ontology and a processing component (a service), bringing together both the conceptual domain aspects declared in the ontology and the application-specific procedural (behavioral) aspects encapsulated in the service. We refer to the relation between these two aspects as *the application semantics* that defines the meaning of ontological concepts in terms of the behavior of the corresponding services. By shifting the focus from a service to an ontology we can interpret Onto⇔SOA as a general mechanism

to attach a service to an ontology. We will use Service-enabled Ontologies to refer to this interpretation. Unlike *formal semantics*, application semantics of ontologies has not been directly addressed in ontology-related research.

We motivate the notions of the application semantics and service-enabled ontologies by pragmatic concerns of ontology engineering and application of ontologies in software. In Knowledge Representation there is a history of systems that supported so called *procedural attachments* (CLASSIC [46], KL-ONE [47], CLIPS [48]), enabling direct invocation of procedures from within a formal representation mechanism (e.g., production rules). We can explain the need for these attachments by pragmatism: complex real-life application scenarios are cumbersome (and often infeasible) to address with formal mechanisms only.

Such procedural attachments were regarded as a temporary workaround because they endangered properties (decidability, consistency, etc) of formal foundations. In contrast to this, in Service-enabled Ontologies we propose to employ services to explicitly address the application semantics of domain concepts. By integrating a service into an ontology we re-enable a hybrid *semi-formal or semi-declarative* approach to Ontology Engineering.

In Service-enabled Ontologies we regard a service as a container of domain-aligned functionality that brings the *behavioral* aspect, captured in a non-formal "procedural" way, to conceptual and declarative domain models. The main distinction of such approach from, for example, rule-based approaches, is that a service is not limited to a particular representation mechanism. We believe that the flexibility of Service-enabled Ontologies simplifies development and application of ontologies, thus improving their usability and facilitating their application.

Therefore, Onto⇔SOA can be seen as an approach that integrates ontologies and services unifying two perspectives:

- the Ontology-enabled Services perspective introduced in Chapter 2, and
- the Service-enabled Ontologies perspective introduced in this chapter.

Such integration of these two perspectives is enabled by our emphasize on

- domain alignment: both ontologies and services model an application domain at the level that is close to the user;
- utility of ontologies and services: we assume that they both have to be designed to provide their users with ready-to-use utility.

In Chapter 5 we introduce *MoRe* as an implementation of Ontology-enabled Services that integrates RDFS ontologies into REST-like services. However, we can also interpret *MoRe* from the viewpoint of Service-enabled Ontologies as a mechanism that allows connecting a REST-like service to an RDFS ontology. In this chapter we will describe how the ontology-oriented perspective is employed to the Document Retrieval scenario introduced in Chapter 2.

This chapter is organized as follows. Section 3.1 elaborates on our viewpoint on ontologies and introduces the notion of application semantics. In Section 3.2 we approach Onto⇔SOA as a mechanism that allows to attach a service to an ontology. We, then, discuss the potential Ontology Engineering benefits of this approach in Section 3.3. After that, we return to the Document Retrieval case to illustrate the ontology-oriented perspective (Section 3.4). Finally, we conclude with Section 3.5.

3.1 Ontologies and Application Semantics

Ontologies have been exposed to a wide range of communities. Already in 1995 clarification was required on what the term "ontology" means in different research fields [12]. Recently, because of the W3C activities (RDF/S [13, 14], Web ontology languages [15], the Semantic Web [16]) ontologies have attracted even more interest increasing the number of possible interpretations of the term. The main role of an ontology is to facilitate an agreement between parties (either human or artificial) on the intended meaning of domain concepts.

There are many ways to classify the variety of ontologies according to their degree of formality, generality [49], detail [50], etc. Normally, an ontology is considered as a domain model that captures application-independent semantics of concepts. Nevertheless, since we seek to employ an ontology in a certain application context, the applicability of an ontology in a particular scenario becomes an important factor.

One of the underlying assumptions behind our work is that *to be effective an ontology (conceptualization) should be created within a clearly defined application context*. The more precisely the context is defined, the fewer efforts is required to design an ontology and then extract utility from it. In ontology engineering methodologies [51, 52] the phase of determining the competence area (targeted application scenarios) of an ontology is widely acknowledged. However, we believe that in practice it does not receive due attention, resulting in ontologies that have no clearly defined application boundaries and therefore lack directly exploitable utility.

For example, in the document retrieval case introduced in Section 2.1 we deliberately limit the application domain to the task of retrieving of matching documents. If we do not restrict ourselves to the target domain, it would be tempting to include the notion of index into the ontology of that domain. However, since the definition of the document retrieval case does not contain the concept of index, neither may the ontology associated with the document retrieval service include it (even though the service uses this notion internally).

An ontology facilitates consistent interpretation of concepts (i.e., shared understanding) by un-related and de-centralized agents. The agent's interpretation of an ontology-based message must be validated by the original carrier of this domain knowledge – the domain expert. However, the effectiveness of such validation is limited by the availability, costs and quality of domain experts.

By constructing a *formal model* of the expert's knowledge and applying it to validate the interpretation we can mitigate these limiting factors. Presently, many attempts are made to establish shared understanding by defining formal knowledge representation standards such as RDF/S and OWL. However, there is always a tension between the expressiveness of these generic, application-independent languages and the ability to solve specific tasks in real-life applications. The effectiveness of the formal approach depends on a number of factors such as: expressiveness of the representation mechanism, the complexity of constructing and employing formal domain models, the range of provided reasoning services, adequacy of the formal approach to the application area, etc.

In the end, it is always the behavior of the (software) agent that displays the actual interpretation of ontological concepts. Here we propose to extend the consequences of this observation: *consistent interpretation of an ontology (and concepts described therein) can be achieved by providing widely accessible and directly exploitable software components that serve as the reference for the intended interpretation by displaying the required behavior*.

In Ontology-enabled Services the goal of an ontology is to transfer domain concepts into a service and to facilitate consistent interpretation of these concepts. An extensive formal foundation of the underlying ontology language can provide a valuable support for ontology construction and validation, however it is not the only means for expressing semantics. In a pragmatics-driven software design process the interpretation of domain concepts by software is predominately validated by domain experts.

We therefore can distinguish the following types of semantics of concepts contained in an ontology (Figure 3.1):

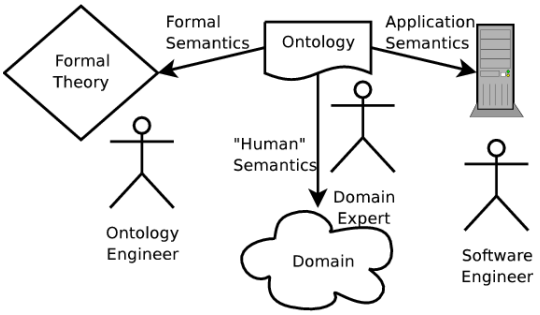

Figure 3.1: Kinds of semantics of concepts captured in an ontology.

- *domain semantics* represents the meaning of a concept as understood by a domain expert;
- *formal semantics* employed in ontology languages such as the OWL-family is based on Description Logics. The state of the art ontology engineering heavily relies on this approach to ontology construction;
- *application semantics* is determined by how a software agent interprets the concepts defined in an ontology.

So far, formal semantics of ontologies has attracted most of the research effort in Ontology Engineering. However, the complexity of the formal foundation of modern ontology languages hinders their usability and adoption by practitioners. The belief that advanced tool support will bring such ontology languages to a wide audience is yet to materialize. On the other hand, ontology-enabled applications are needed to implement the Semantic Web vision [53, 54]. In this chapter we focus on the more pragmatic aspect of applying ontologies: the application semantics. In Section 3.2 we propose to use services to ground the application semantics of concepts and relations captured in an ontology.

When an ontology is employed in a software system, the application semantics implemented by that system may overtake the original domain or formal semantics. F. P. Brooks in [55] described "specification by implementation" – a phenomenon where an implemented software system, rather than its (formal) specification, becomes the reference point for interpretation of concepts exported into other systems. By assuming *high availability and accessibility* of the reference system we can validate application semantics against it.

The application semantics of a concept (and an ontology in general) is grounded in a software component that implements the application logic. From the Knowledge Representation perspective, this can be interpreted as attachment of a domain- and task-specific inference component to an ontology. We propose to employ a service to represent such inference component. Such a service can be seen, for example, as a problem solver or a knowledge source [56] designed for the domain of an ontology.

From the application perspective, the ability of an ontology to capture such procedural (behavioral) domain knowledge is very attractive because it provides software agents with ready-to-use utility. Presently, standard RDFS reasoners provide such utility to some (rather limited) extent: in many practical applications computation of transitive closures for the `rdfs:subClassOf` relationship is the only directly exploitable utility. OWL reasoners are able to provide more inference services. Nevertheless, they are notoriously difficult to employ in software systems without a sufficient knowledge of the underlying formal foundations. Moreover, despite contrasting *application* and *formal* semantics we do not intend the former to replace or exclude the latter. As a matter of fact, OWL reasoners can be seen to provide the application semantics for OWL in the domain of logical reasoning.

There is an ongoing research on how to extend the capabilities of formal ontology languages by combining them with additional formal mechanisms more capable of expressing behavioral aspects. For example, a number of rule languages such as SWRL[1] and RuleML[2] have been proposed for this purpose. However, complexity and lack of flexibility of rule languages make it rather difficult to employ by practitioners. In many cases a non-formal approach is the only alternative viable in practice. For example, rule languages typically do not cover numerical reasoning, which is quite a limitation for applying them in e-Science, in particular. We believe, however, that rule languages themselves could be supported by services.

3.2 Attaching Services to Ontologies

In the previous section we have alluded to the difficulty in using formal methods to address the ever-changing requirements of real-world problems. This issue has been addressed in many Knowledge Representation systems (CLASSIC [46], KL-ONE [47], CLIPS [48]) by

[1] http://www.w3.org/Submission/SWRL/
[2] http://www.ruleml.org/

including a limited ability to interface with procedures implemented in traditional programming languages. We also consider procedural attachments to be an approach to capture application semantics of domain concepts. We believe that the ability to incorporate application semantics can contribute to the effectiveness of ontology languages in such a diverse, dynamic and pragmatic environment as the Web. Onto⇔SOA already contains components required to realize that ability.

To design an Onto⇔SOA architecture we have to create two main components: a service and a service ontology. Since in our approach a service ontology is the only specification of a service available to consumers, we can see the design of an ontology-enabled service-oriented architecture primarily as development of a service ontology[3].

By emphasizing the need for a service ontology we can re-interpret the Onto⇔SOA framework as a mechanism that: enables to attach a service to an ontology, thus defining application semantics of concepts captured in this ontology.

Since the Service-enabled Ontologies perspective covers the same components as the Ontology-enabled Services, it is subjected to the same constraints. The effect of the constraints, however, should be re-interpreted to reflect the change of focus from a service to an ontology. This re-interpretation will also explain the difference between a service ontology and a more traditional application-independent domain ontology.

A service ontology describes *an application domain* that can be seen as a combination of a certain domain (e.g., the document corpora domain) and a task applied to that domain (e.g., the retrieval task). On the other hand, a domain ontology is intended to cover an application-independent domain.

A concept such as "document" gains different facets of meaning when employed in different tasks. For example, for the retrieval task we assume that a document contains a natural language text that determines the relevance of a document to a given query. In other tasks, such as document archiving for example, the actual content may be irrelevant but other properties become important (creation and modification dates, size, etc).

There are always implicit assumptions that affect (a particular facet of) the meaning of a concept but that have to be left outside a formal model. These assumptions create a unique application context within which a domain concept is interpreted. An ontology-enabled service captures this context, thus potentially complementing a formal (declarative) representation of the concept.

[3]We also believe that the inverse should hold: development of an ontology that captures application semantics is largely equivalent to development of an Onto⇔SOA architecture.

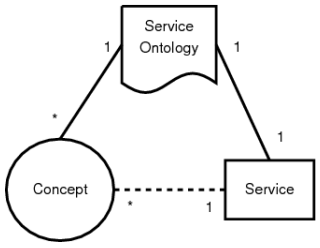

Figure 3.2: The relations between concepts, a service ontology and a corresponding service.

Our initial assumption is that each application context is unique, therefore *there is only one service attached to concepts from a given application domain (model)*. A service ontology contains a collection of related concepts, the application semantics of which is determined by the shared application context. These concepts cannot be employed outside that context because the implicit assumptions will be lost, thus potentially hindering consistency of interpretation of these concepts.

In Onto⇔SOA we have made it explicit that a concept can belong to one service ontology only (see Sections 2.2.2 and 2.3.2) that, in turn, can have only one service associated to it (Figure 3.2). This makes it trivial to use a concept within its application domain but explicitly requires additional measures (e.g., ontology mapping) to reuse this concept across different application domains. This property of service ontologies contrasts to *domain ontologies* that are expected to contain concepts ready to be reused across applications. Nevertheless, this contrast should not be perceived as a sign of a conflict but rather as an explicit acknowledgement of the problem of concept reuse across application domains.

Onto⇔SOA also suggests a solution to this problem of reuse: *by preserving a link between a concept, its service ontology and a corresponding service we are able to reuse that concept in different application domains as long as the corresponding service is used to interpret the concept*.

By integrating a service and an ontology, we combine the Software Engineering and Knowledge Representation perspectives on services. As we will demonstrate in Chapter 4, the latter allows to approach a service as a knowledge source. Knowledge Sources have been introduced as independent, self-contained computational components that carry domain knowledge. We find these characteristics of Knowledge Sources to match well to

those of Onto⇔SOA services.

We have already stated that the full formal semantics of advanced languages such as OWL may be too complex or restrictive for practical applications. Nevertheless, modern ontology languages such RDFS and OWL are de-facto standard means for specifying ontologies. Therefore, we will provide some guideline on how to service-enable these ontology languages and more generally languages with a strict formal semantics.

One of the approaches is to release part of the formal interpretation and allow for application semantics to overtake it. For example, the OWL restrictions can be interpreted (slightly) differently from their formal definition (see Section 5.4). The only requirement is that the software developer and the knowledge engineer ensure that the service displays behavior that complies with the expectations of the domain expert. In other words, we can allow for a short-cut between expert knowledge and service behavior using a light-weight interpretation of formal semantics of the ontology language used.

Another approach is to fully respect the formal semantics of an ontology language but select a language with less restrictive formal semantics. In this case the more extensive and restrictive it is – the more difficult its service-enablement can be due to a likely conflict between formal and application semantics. This implies, for example, that since RDFS has much less restrictive formal semantics than the languages from the OWL family, the former can be integrated with a service with less concerns about possible semantic conflicts.

3.3 Potential Benefits of Service-enabled Ontologies

In the rest of this section we outline the capabilities of service-enabled ontologies which we believe can make them more usable than strictly formal ontologies.

Refining Domain Models In Chapter 1 we have argued that although it is assumed that a specification mechanism underlying an ontology and domain conceptualization are independent, in practice it is not the case with ontology languages that have restricted formal semantics. In these languages the underlying formal foundation often forces a particular approach to domain conceptualization. This can lead to a mis-alignment between the meaning of a concept as understood by a domain expert and the formal meaning of the domain model.

Service-enabled ontologies deliver a facility to ground the meaning of a concept in a service. This provides ontology engineers an additional means to stay closer to the domain meaning of concepts. Such a hybrid approach makes the relation between a specification

mechanism and domain conceptualization more flexible allowing for a better alignment between an ontology and the expert's understanding of the domain.

One of the consequences of this increased flexibility is the ability of a service-enabled ontology to specify refined domain models. Often a traditional ontology language lacks the ability to describe a domain at the level of detail required in, for example, e-Science applications. For example, in the Unit Conversion case described in Chapter 5 we had to capture the mathematical aspects of the property that represents the conversion factor between two units of measure. None of the modern ontology languages is able to express this type of mathematical knowledge, let alone to provide the corresponding reasoning support. The only option the user has is to wait for an ontology language extension supporting mathematical expressions. On the other hand, in many domains including e-Science mathematical aspects play a very important role and have to be dealt with already now.

By enabling the mathematical aspect of the considered property to be captured in a dedicated service, we allow an ontology engineer to create a deep domain model with some aspects exposed formally and others captured in a service. We believe this significantly simplifies ontology development.

Efficient Inference Services In any practical situation the trade-off between generality and efficiency is inevitable. General-purpose reasoners and inference engines are always less efficient than reasoners tuned for a specific domain. Traditionally, reasoning support for an ontology language can be made more efficient by reducing the expressiveness of the language. Service-enabled Ontologies open up additional ways to improve computational efficiency by introducing domain-specific inference services. This can be achieved by choosing the most efficient domain-specific inference procedure and using it as a foundation for domain modeling. This not only provides optimal performance but also supports creating refined domain models. This inference mechanism can be either symbolic (e.g., logic programming or rule-based), computational (neural network, evolutionary) or an arbitrary software component.

Incremental Development Usually, it is much easier to initially capture knowledge in a procedural way because it does not restrict the user to a particular declarative representation. Later on, some parts of the procedural knowledge can be exposed in a declarative way through an ontology. In this way, Service-enabled Ontologies allow for an evolutionary transition from procedural to declarative knowledge representation by specifying *a declarative interface to procedural knowledge*. We believe that all this improves the utility of an

ontology, improves its usability and facilitates application of ontologies in software.

3.4 Ontology-oriented Perspective on Document Retrieval Case

In Chapter 2 we have described a solution to the Document Retrieval case from the perspective of Ontology-enabled Services. In this section we complement that description with the Service-enabled Ontologies viewpoint.

The formal semantics of the RDFS and OWL languages is supported by corresponding reasoning services. The formal semantics of these languages is usually hidden from the users (an ontology engineer, for example) by means of an advanced tool support. The reasoning services, however, are likely to be encountered by the users because they can facilitate ontology development (consistency checking service, etc). When we seek to apply an RDFS or OWL ontology the available reasoning services become less valuable because they offer little utility for the majority of application scenarios.

Computation of transitive closures in RDFS is an example of a reasoning service valuable in applications. The transitivity of the `rdfs:subClassOf` relation is described in the RDFS language specification [14] that defines the corresponding inference procedure by means of inference rules. These rules are implemented in middleware either as a dedicated inference service (inference models in Jena API [57], for example) or as part of an ontology query language (SPARQL [58]).

Unlike consistency checking services, computation of transitive closures is often applied not at the ontology design stage only but also at the deployment stage. Computation of transitive closures by a software component can be seen as an example of the application semantics of the `rdfs:subClassOf` relationship. The application domain in this case can be described as *RDFS reasoning*.

Let us for example compare reasoning services in the domain of the RDFS language and that in the application context of our Document Retrieval case. There are similarities between the application semantics of the `rdfs:subClassOf` relation and concepts from the Document Retrieval service ontology:

- in both cases we have an initial domain situation described by a number of facts that are then extended with new facts inferred by a reasoning service. In the `rdfs:subClassOf` case we can have a hierarchy described by two facts (A `rdfs:subClassOf` B; B `rdfs:subClassOf` C) and in the Document Retrieval case we have a description of a document corpora and a text query (the left-hand part of

Fugure 2.2). Having applied the transitive closure reasoning service we obtain a new fact (A `rdfs:subClassOf` C) and by applying the Document Retrieval service – facts describing a collection of matching documents (the right-hand side of Fugure 2.2);

- in both cases the reasoning service determines the application semantics of an ontological concept. In the `rdfs:subClassOf` case we compute the transitive closure and in the Document Retrieval case we parse the `documents`, analyze their content with respect to a given `query` to find `matching documents`;
- in both cases an (RDFS) ontology provides the terminology to describe initial domain situations and inferred facts.

A notable difference between the transitive closure and the Document Retrieval services is that the former is defined in a *formal* way and the latter is not. However, we believe that from the application perspective this is not very important as long as the corresponding inference services can be *implemented* in software.

The formal definition of transitive closures assures consistency between different implementations (e.g., the Jena middleware versus the RDQL/SPARQL query engines) of a corresponding reasoning service. Nevertheless, as we proposed in Section 3.1, a widely accessible and directly exploitable implementation of such a service can be employed to facilitate consistent interpretations. We do not argue to abandon formal semantics. We do argue, however, that ontological concepts not always have to be defined formally to be effective in practice. We believe, that an ontology can carry both formal and application semantics of domain concepts.

To summarize, the Document Retrieval service ontology captures a complex relationship between concepts supplied by a consumer and concepts inferred by a service. With Service-enabled Ontologies we define a mechanism that allows to attach a service to a Document Retrieval ontology preserving the application semantics and making it available to the ontology users.

3.5 Conclusions

We have introduced the ontology-oriented perspective on Onto⇔SOA. This perspective provides us with an answer to the research question: *"How can we attach a service to an ontology and what does this imply for ontologies?"*. By shifting the focus to a service

ontology we can re-interpret Onto⇔SOA as a mechanism that allows to attach an arbitrary service to an ontology, thus capturing *application semantics* of domain concepts.

We believe that the proposed integration of services into ontologies has a number of beneficial implications:

- decoupling of the inference service from the ontology language makes conceptualization less dependent on the representation capabilities of the ontology language;
- increased flexibility in specifying conceptualization reduces the gap between the meaning of concepts as defined by an ontology and expectations of domain experts;
- it provides the ability to provide efficient domain-specific reasoning services;
- the user is able to decide what type inference mechanism (symbolic, computational or procedural) suits best to represent domain knowledge;
- and, ultimately, this approach bridges Software and Ontology Engineering.

Overall, we believe that the ontology-oriented viewpoint on Onto⇔SOA demonstrates the potential of integrating procedural and declarative knowledge that allows us to benefit from usability of the former and reusability of the latter.

Chapter 4

Service Collaboration in Onto⇔SOA

In this chapter we investigate the research question: *"How can ontology-enabled services work together?"*. To answer this question we propose an approach to service composition based on the ideas from Blackboard Systems extensively investigated in AI in 1970-80s. We combine these ideas with Onto⇔SOA. The proposed Blackboard-style composition approach requires neither an extensive service model nor an explicit workflow specification and enables composite functionality to *emerge* by bringing a number of services together and making them interact via a shared repository. We illustrate that a Blackboard-style mechanism combined with a restricted service model is a feasible approach for non-trivial service composition scenarios. To demonstrate our approach in the e-Science domain we compose a number of services to check consistency of units of measurement in mathematical statements.

◇

Maksym Korotkiy and Jan L. Top: Blackboard-style Service Composition with Onto⇔SOA. In proceeding of the WWW/Internet 2007 conference. Vila Real, Portugal, 2007.

◇

Services are seen as software components that can be effectively combined to provide complex functionality to their consumers. A variety of SOA frameworks and approaches to service composition exists. The field of Semantic Web Services (SWS) is closest to the area of our research. OWL-S [28] and WSMO [29] are the two most well-known SWS approaches. Both of them rely on extensive ontology-based semantic service models to automate tasks such as discovery, invocation, choreography and orchestration of Web Services. The extensive formal frameworks defined by these approaches achieve, to a certain extent, the goal of automated service composition [59]. However, as these frameworks aim to cover the widest possible range of services, they tend to become highly complex hindering their overall acceptance.

In Chapter 2 we have proposed the Onto⇔SOA framework as a simple yet effective ontology-enabled approach to designing usable (and reusable) services. We aim to improve the usability of services by enforcing their *domain alignment* and *loose coupling* characteristics. To achieve that we have defined a restricted service model that exposes only a conceptual service interface captured in a schema referred to as a *service ontology* (Figure 2.8).

The service model in Onto⇔SOA requires a service to be *document oriented*, in contrast to communication in terms of remote procedure calls (RPC) commonly employed in (Semantic) Web Services. A service ontology, therefore, is a specification of a document-oriented service interface. It describes vocabulary employed in documents communicated to and from a service and exposes no other details about a service such as preconditions, effects and process model. In Onto⇔SOA a service is limited to a single operation: extending a request document with new facts.

Onto⇔SOA aims exclusively at the task of service invocation. Service discovery and composition are not targeted by the core framework. However, we believe that these tasks will be also facilitated by our approach. In this chapter we support this statement by further extending the framework with a composition mechanism based on Blackboard Systems [56].

The principle behind Blackboard Systems is best explained by drawing the analogy with a team of experts cooperatively solving a complex problem on a blackboard. The experts are independent, belong to different domains and do not directly interact with each other. Instead, they observe the current state of the problem solving process captured on the blackboard, and opportunistically contribute to it by applying their domain knowledge. Blackboard Systems exhibit the ability to supply a general and flexible composition mechanism capable of organizing multiple components and have been proven to work in a variety of application areas such as speech recognition, process contro, and case-based reasoning.

Our work is motivated not only by the need to address collaboration between services (i.e. the task of service composition). The service model defined in Onto⇔SOA bears many similarities with Knowledge Sources – one of the key components in Blackboard systems. Additionally, by introducing an ontology into a service we extend the latter in the direction of Bnowledge-based Systems, of which Blackboard Systems is a prominent representative.

In short, in this chapter we demonstrate that the Blackboard-based mechanism is a viable approach to service composition in Onto⇔SOA. The proposed approach is general and

application-independent. We also show that despite being intentionally restricted the service model in Onto⇔SOA is capable of supporting collaboration between services. More generally, we submit that in many non-trivial scenarios no extensive description of a service model or workflow is required to enable effective composition of services.

To support these claims we employ a use case from the e-Science domain. This use case addresses the problem of detecting inconsistent use of units of measurement and dimensions in mathematical statements. To solve this task we implement a number of Onto⇔SOA services and make them collaborate using a Blackboard-based approach.

We organize this chapter as follows. In Section 4.1 we describe the use case. After that, in Section 4.2 we describe the main components of Blackboard Systems as employed in AI. Next, in Section 4.3 we adjust the traditional Blackboard composition mechanism to Onto⇔SOA and apply it to the consistency checking use case in Section 4.4. We elaborate on the design of two services and describe a sample composition run in Section 4.4.3. Finally, we discuss some issues of the proposed Blackboard-style service composition in Section 4.5 and conclude with Section 4.6.

4.1 Use Case: Checking Consistency of Units of Measurement in Mathematical Statements

In many engineering and scientific applications consistent use of units of measurement and dimensions is an important quality assurance tool. There are numerous examples of severe losses resulted from inconsistent use of units of measurement. To name one, we can refer to a 125$ million Mars orbiter lost in 1999 because engineering teams used units from different measurement systems[1]. A loss like this could have been prevented if an automated unit consistency checking would have been implemented.

In order to determine consistency[2] of an expression (e.g., $F = m \times a$) we have to know what units are assigned to each of the variables (e.g., F – Newton, m – kilogram, a – meter per second squared). Given this information we can apply knowledge from the domain of units of measurement and determine consistency of that expression (e.g., Newton can be expressed as kilogram times meter per second squared, hence the assignment is consistent).

We will address this use case by designing a demo application that relies on a consistency checking service composed from a number of independent services. The workflow as

[1] http://www.cnn.com/TECH/space/9909/30/mars.metric.02/

[2] Here and further on we use consistency to refer to consistency of units of measurement and dimension.

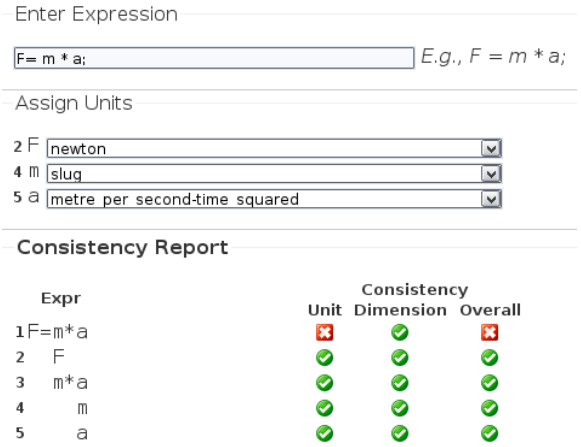

Figure 4.1: A screenshot of the Unit Consistency Checker demo application. In this example we can see that the assignment expression is inconsistent with respect to units of measure but consistent with respect to dimensions. The easiest way to fix this inconsistency is to replace the slug unit with kilogram.

implemented in the demo application consists of three steps which are clearly recognizable in the GUI (Figure 4.1). In the first step the user types in an expression. Next, the user assigns units of measurement to identifiers (variables) employed in the expression. Finally, the user activates the consistency checking procedure, analyses a consistency report, and if necessary reassigns units of measurement.

4.2 Blackboard Systems in AI

Blackboard Systems have been extensively researched in AI in 1970-80s. They have been applied in numerous application areas such as process control, planning and scheduling and speech recognition (see [60] for an extended introduction into the field). As we have already mentioned in the introduction the main idea behind Blackboard Systems can be illustrated by comparing it to a team of experts that cooperatively solve a problem via a blackboard. The experts are allowed to interact via the blackboard only and their access to the blackboard is managed by a dedicated, application-specific controller. Thus, we can

distinguish three main elements in Blackboard Systems: the Knowledge Sources (experts), the Blackboard and the Controller (Figure 4.2).

Knowledge Sources are mutually independent functional components capable of inspecting and modifying the Blackboard. In many Blackboard Systems a knowledge source consists of trigger and action procedures. The trigger procedure allows a knowledge source to determine if a blackboard contains facts sufficient to contribute to it. The purpose of a trigger is similar to the purpose of service preconditions employed in Semantic Web Services (SWS) approaches; a trigger can detect whether all required data are available for a component to start processing. However, unlike preconditions in SWS, triggers are normally neither specified declaratively, nor are they employed for automatic construction of workflows. Triggers enable the Controller to schedule Knowledge Sources to achieve the most efficient problem-solving process.

The Blackboard is a heterogeneous repository (fact storage) shared by all Knowledge Sources and the Controller. The Blackboard serves as a shared repository enabling cooperation among Knowledge Sources. It can also serve as a temporary buffer. The Blackboard can contain a symbolically represented and, often, hierarchically organized solution space. It also can store control data employed by the Controller. The structure of the Blackboard is usually application specific to achieve the most efficient communication among Knowledge Sources and the Controller. It is assumed that there is a certain syntactic and semantic compatibility between Knowledge Sources. This allows them to (at least partially) understand the content of the Blackboard and extend it with new facts, which in turn can be understood by other Knowledge Sources.

The Controller synchronizes and coordinates Knowledge Sources to establish an effective and efficient problem-solving process. The overall application-specific problem-solving strategy is normally embedded in the Controller. The strategy is flexible enough to enable arbitrary scheduling of Knowledge Sources that is decided upon by the Controller on the basis of trigger procedures.

In AI the following benefits of Blackboard Systems are emphasized most often [38, 56]:

- Blackboard Systems are arguably considered to be the most general and flexible architecture for building knowledge-based systems.

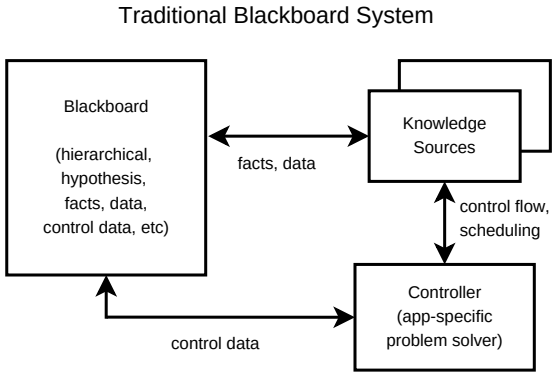

Figure 4.2: Major components of a Blackboard System.

- Blackboard Systems provide for an excellent integration framework for components (Knowledge Sources) that employ heterogeneous representations and expertise. This characteristic of Blackboard Systems is very attractive for Enterprise Application Integration for which SOA is also often employed.

- The Separation of concerns between the Controller and Knowledge Sources allows a Blackboard System to make dynamic control decisions: the Controller can steer the problem-solving process depending on its state and the information supplied by the triggers. This property is also very attractive for SOA deployed in the very dynamic environment of the Web.

- The inherent modularity of a Blackboard System and independence of Knowledge Sources provide for significant software engineering benefits: for example, each component can be implemented, adjusted and tested independently from others.

As summarized in [38] the main disadvantage of Blackboard Systems is that they do not scale down very well to simple problems. In addition, they are considered to be useful only during prototyping. For performance reasons, production systems are usually reimplemented with more conventional means providing better performance. Finally, the components of traditional Blackboard Systems appeared to be rarely reused: in most cases Blackboard Systems were designed from scratch. We will revisit these issues later in this chapter in the context of Onto⇔SOA.

4.3 Blackboard-style Service Collaboration in Onto⇔SOA

The generality and flexibility of Blackboard Systems make the underlying composition mechanism very attractive to Service-Oriented Architectures. Although Onto⇔SOA services share many properties with Knowledge Sources, there are also considerable differences between the organization of Blackboard Systems and the way we propose to design services in Onto⇔SOA:

- In Blackboard Systems Knowledge Sources are designed to work together on a specific, predefined task. On the other hand, Onto⇔SOA services are not aware of the complex application scenarios in which they participate.
- Blackboard Systems emphasize flexible and dynamic control that is to a significant extent achieved by employing triggers. Usability of Knowledge Sources is not a design goal as such. Contrary to this, in Onto⇔SOA usability of services is the most important concern. One of the main means in improving usability in Onto⇔SOA is by hiding implementation details of a service such as triggers.

Therefore, we will have to adapt the Blackboard-style composition mechanism to apply it in Onto⇔SOA. When doing so we will maintain the simplicity and usability of our framework. In the following subsections we describe the three main elements of Blackboard-style composition as applied in Onto⇔SOA: the Controller, services acting as Knowledge Sources and the Blackboard.

4.3.1 The Controller

We begin by defining the Controller because it requires considerable adjustments to meet the restrictions of the Onto⇔SOA service model. According to this model the Controller belongs to the connector mechanism because it is responsible for coordinating communication between a service and its consumers, or between services. Therefore, being part of the connector the Controller must be designed as a *simple* and *application-independent* component.

The Controller can be employed either as a stand-alone service or as a component included into a service that internally combines several services. In both cases, however, the Controller itself does not contain any application-specific logic that is to be contained either in one of the services or in the service consumer. Such an application-independent Controller allows Onto⇔SOA services to be easily combined and employed in different

application contexts. We assume that the Controller knows which services participate in collaboration but not what they do. At this point we do not elaborate on how the Controller discovers the services to be composed.

In Onto⇔SOA the Controller uses the following basic composition procedure:

1. The Controller sequentially invokes all services to be composed in a *non-predefined order*. Each service is invoked exactly once per cycle. During each invocation the Controller sends the (relevant part of the) content of the blackboard to one of the services as an input document. The service sends its output document back to the blackboard.

2. Another invocation cycle follows if the content of the blackboard has been modified after invoking all services. Otherwise, the process stops.

In the simplest scenario the Controller submits the complete content of the blackboard to a service. However, since it is likely that only part of the blackboard's content is relevant to a given service, the Controller can extract from the blackboard a fragment limited to concepts used in the corresponding service ontology. In this way the efficiency of the communication between the Controller and Knowledge Sources can be improved.

Presently, for simplicity reasons (in the spirit of Onto⇔SOA) and to avoid concurrency related problems, we assume that during composition at any given moment at most one service may access the blackboard.

Since the Controller contains no application-specific logic and the composed services expose their conceptual interfaces only, the Controller cannot predict whether a service can contribute to the blackboard at a given iteration. Hence, unproductive invocations – service invocations that do not add new facts to a blackboard – cause additional overhead. The Controller can reduce this overhead by inspecting the changes on the blackboard and adjusting the service invocation order for next iterations. We discuss this issue in some detail in Section 4.5. However, finding a generally applicable composition optimization mechanism is outside the scope of our work at the moment.

4.3.2 Onto⇔SOA Services as Knowledge Sources

Knowledge Sources are traditionally seen as domain-specific problem solvers. In the proposed Onto⇔SOA framework we treat services in a more general way as experts capable of applying their (procedural) knowledge to infer facts about the domain of discourse. A few additional assumptions and clarifications are required to enable the participation of

Onto⇔SOA services in Blackboard-style composition as described in the previous subsection.

First, the services must be compatible with each other on both the conceptual (semantic) level and data-model level. In *MoRe*, a specific implementation of Onto⇔SOA, we use the RDF data-model for expressing documents. To achieve semantic compatibility between the collaborating services we require that each service ontology overlaps with at least one other service ontology. This overlap enables a flow of facts through the services allowing them to benefit from each other's expertise.

Both semantic and data-model incompatibility can, in principle, be resolved via mediation mechanisms. However, we do not consider such mediation mechanisms to be an integral part of the composition framework. The main reason for not assigning any special role to them is that mediation services can be effective by participating in the blackboard-based composition in the same way as any other services.

Second, the introduced composition procedure terminates when none of the services modifies the blackboard anymore. We take measures to reduce chances that (conflicting) interaction between services prevents normal composition termination. For this we require that the services may only add new facts to the blackboard. They may neither remove nor modify existing facts. In addition, to prevent infinite expansion of the blackboard we require that a service does not modify a document generated by the same service: i.e. submitting an output document of a service to the same service will not modify the document.

However, there are still many scenarios in which composition does not terminate normally. In those cases termination could be achieved by enforcing a maximum number of service invocations. Such kind of abnormal termination is not preferred because it would require to expose internal details of a composition procedure hindering both domain alignment and loose coupling of such a service. The preferred approach to resolve such cases is to consult the application domain where service composition takes place. If this domain corresponds to a complex business process then this process should already contain a mechanism (business rules) that covers such abnormal termination of the process. By encapsulating this mechanism into the composed services we ensure that service composition always terminates in a way aligned to the underlying business process.

Finally, in Onto⇔SOA we favor session-stateless services. This implies that the collaborating services should not use the Blackboard as a buffer for *internal* intermediate results. Under intermediate results we understand facts that are of no utility to a consumer (for example, because they are superseded by other facts or reflect internal details of the reasoning

process) or other services being composed.

4.3.3 The Blackboard

The Blackboard contains information (a collection of facts) shared between services participating in composition. In Onto⇔SOA the Blackboard has a homogeneous structure, compatible with the data-model employed in the documents communicated to and from the collaborating services. For example, if we aim to compose *MoRe* services then the corresponding Blackboard must use the RDF data model. In Onto⇔SOA the Blackboard as such is conceptually neutral – it does not enforce any conceptual structure (e.g., a hierarchically-organized solution space as in traditional Blackboard Systems).

The Blackboard is used exclusively to enable interaction between services. As stated previously, we do not allow it to serve as a temporary buffer for intermediate results internal to the respective services. Neither may it contain control data (such as statistics of successful service invocations, for example).

In traditional Blackboard Systems the blackboard structure is optimized to achieve an efficient problem-solving process. Contrary to this, in Onto⇔SOA we emphasize usability of services achieved by unifying the structure of communicated documents and enforcing domain alignment of contained concepts.

4.4 Solution to the Use Case

We apply the proposed Blackboard-style service composition to the unit consistency checking case introduced in Section 4.1. We employ the UnitDim Ontology[3] as an explicit knowledge model of the domain of units of measurement.

In this use case we implement a demo application that uses the introduced application-independent Controller to compose five Onto⇔SOA services. The services do not depend on each other and are not specific to the task addressed in the use case, thus they can be reused in various application contexts.

A minimum amount of application-specific logic is contained in the demo application that acts as a consumer. This logic is available neither to the composed services nor to the Controller. The demo application supplies the facts describing the initial situations in terms of the mathematical statement and, optionally, units of measurement assigned to the identifiers used by it.

[3] http://www.atoapps.nl/foodinformatics/NewsItem.asp?NID=7

The Controller invokes (in a non-predefined order) the following services to work on the initial situations:

- The Parser service (see Section 4.4.1) transforms a mathematical statement[4] into a parse tree representing its underlying structure. For example, the statement

  $$F = m \times a;$$

  is decomposed into the assignment expression $F = m \times a$ that consists of variable F and the multiplication expression $m \times a$, which in turn includes two variables m and a.

- The Unit Assigner service recognizes variables by their name and automatically assigns units of measurement commonly used for those variables. For example, this service can automatically assign the unit of measurement Newton to variable F.

- The Unit Consistency Checker service (elaborated upon in Section 4.4.2) analyzes the structure of a mathematical expression with units of measurement assigned to its variables. The Unit Consistency Checker attaches one of the three possible values (*unit consistent*, *unit inconsistent*, *unit consistency unknown*) to the statement and to each of its sub-expressions.

- The Dimension Consistency Checker service determines dimensional consistency of the statement and its sub-expressions in a manner similar to unit consistency checking.

- The Overall Consistency Checker service combines the outcomes of the unit and dimension checks to determine the overall consistency of the statement and each of its sub-expressions.

In the coming subsections we elaborate on the design of the Parser and Unit Consistency Checker services.

4.4.1 Parser Service

In Onto⇔SOA a service is defined primarily via its service ontology. This ontology contains the vocabulary needed to express input and output documents sent to and created by

[4] We support a subset of the syntax of the Matlab language to express mathematical statements from science and engineering. This subset also occurs in many other programming languages.

the service. The sole task of the Parser service is to transform a given input statement into a tree of expressions. Hence, we have to define a service ontology capable of supporting this task.

The *statement* is the central concept in this service ontology. Each statement is linked to its *source* – a representation of the statement in a Matlab-like language, and to a collection of *expressions* constituting the statement. Each expression in turn also has a source and can be linked to other expressions, thus forming a hierarchical structure. An expression is an abstract type that does not directly appear in the communicated documents. Instead it serves as a super-type for more specific kinds of expressions such as *assignment*, *multiplication*, *identifier*, etc that do appear in those documents.

To illustrate the service let us consider an input document containing the fact:

```
statement
  has source "F = m × a;"
```

This document is extended by the Parser service into the following output document:

```
statement
  has source F = m × a;
  contains expressions
    assignment
      has source F = m × a
      contains expressions
        identifier
          has source F
        multiplication
          has source m × a
          contains expressions
            identifier
              has source m
            identifier
              has source a
```

If an input document does not contain the source of a statement, the Parser service cannot infer the corresponding hierarchical structure, thus returning an unmodified document. If an input document already contains the decomposition tree, then the Parser service returns it unmodified because it may not override existing facts.

4.4.2 Unit Consistency Checker Service

The service ontology of the Unit Consistency Checker service combines fragments from the service ontologies of the Parser and Unit Assigner services and adds the *unit consistency*

property to the concepts *statement* and *expression*.

During operation, the Unit Consistency Checker service takes an input document (like the one produced by the Parser service) with units of measurements assigned by the Unit Assigner service:

```
statement
  has source F = m × a;
  contains expressions
...
        identifier
          has source F
          has unit Newton
...
            identifier
              has source m
              has unit slug
...
            identifier
              has source a
              has unit metre per second squared
```

and determines consistency of each expression and of the overall statement resulting in the document shown on Figure 4.3 (please ignore the bold tags for the moment).

The Unit Consistency Checker service is able to infer unit consistency of a statement or an expression only if the conditions below are met by the input document:

- the statement has been decomposed into sub-expressions;
- units of measurement are assigned to identifiers;
- the input document does not yet contain the consistency of the statement or an expression: a service may not overwrite existing facts (even if they differ from facts inferred by the service).

By providing the service ontology it should be clear to the designers of the service consumer which input is required for the service to infer new facts.

4.4.3 Sample Run

To illustrate the composition process we describe a sample run in detail. In this, run we compose three services: the Unit Consistency Checker (UC), the Unit Assigner (UA) and

0 Matlab statement
0 has source $F = m \times a$;
3-UC unit consistency INCONSISTENT
1-MP contains expressions
1-MP assignment
1-MP has source $F = m \times a$
3-UC unit consistency INCONSISTENT
1-MP contains expressions
1-MP identifier
1-MP has source F
3-UC unit consistency CONSISTENT
2-AU has unit Newton
1-MP multiplication
1-MP has source $m \times a$
3-UC unit consistency CONSISTENT
1-MP contains expressions
1-MP identifier
1-MP has source m
3-UC unit consistency CONSISTENT
2-UA has unit kilogram
1-MP identifier
1-MP has source a
3-UC unit consistency CONSISTENT
2-UA has unit metre per second squared

Figure 4.3: A trace of the blackboard during a sample run. Tags on the left-hand side identify in which iteration and by which service (UC - Unit Consistency Checker, UA - Unit Assigner, MP - Parser) a certain fact has been introduced into the blackboard.

the Parser (MP). Figure 4.3 shows a trace of the blackboard during this run. Tags on the left-hand side identify in which iteration and by which service a certain fact has been introduced into the blackboard.

A description of the initial situation in the application domain (a mathematical statement) is supplied by the user. The demo application expresses it as a document and submits it to the blackboard. On the trace the lines marked with the **0** tag correspond to this state.

Iteration 1: The controller starts invoking services. Neither UC nor UA can contribute to the blackboard because it contains no expressions. MP contributes to the blackboard by

decomposing the statement into a tree of expressions.

The controller determines that the blackboard has been modified during the first iteration and proceeds with **iteration 2**. UC still cannot contribute to the blackboard because the expressions do not have units of measurement assigned. UA contributes to the blackboard by assigning units of measurement to the identifier expressions (variables). MP adds nothing to the blackboard because the statement has already been decomposed into expressions.

Again the controller determines that the blackboard has been modified and proceeds to **iteration 3**. UC contributes to the blackboard by inferring unit consistency of all expressions and the overall statement. Neither UA nor MP can contribute new facts to the blackboard.

The controller determines that the blackboard has been modified and proceeds to **iteration 4** during which neither of the services can contribute to the blackboard because it already contains all the facts derived by the services. After this iteration the blackboard is not modified, thus the controller stops iterating and sends the final content of the blackboard to the consumer (our demo application).

4.5 Discussion

In the described sample run 12 service invocations (4 iterations $\times$ 3 services) took place out of which only 3 were productive. This represents 300% *invocation overhead* and corresponds to the worst-case composition scenario. In this case there is only one productive invocation in each iteration. The final iteration has no productive invocations at all.

This overhead can be eliminated by enabling the Controller to remember the order of productive invocations and re-apply it in future compositions. When the Controller employs such a strategy and terminates composition after the final productive invocation our sample composition will take 3 service invocations only, thus completely eliminating the invocation overhead.

By enabling the Controller to dynamically adjust invocation order and to use the service ontology to communicate to a service only a relevant sub-set of the Blackboard content we can significantly reduce the composition overhead. This, we believe, demonstrates that even with the restricted service model that exposes only a document-oriented interface to a service, we can achieve effective and efficient service collaboration using the Blackboard-based mechanism.

The proposed collaboration approach, and service composition in general, should be

applied with extra care if invocation costs are significant. For example, data-intensive services that expect a large amount of data to be passed through their interfaces bear significant invocation costs, and thus are likely to be inefficient for composition. In Onto⇔SOA the preferred approach in such a case is to reduce the size of the communicated messages by re-modeling a data-centric service on a higher conceptual level, 'compressing' operations on raw data into semantically richer notions. For example, instead of *actually converting* arrays of numerical data measured in pounds into kilograms, a service can describe *how* a conversion between these two units of measurement can be performed: by multiplying values by 0.454. This design demonstrates a knowledge-oriented direction in service design promoted in Onto⇔SOA via ontology-based service interfaces, and further enforced by the Blackboard-based composition mechanism.

In Onto⇔SOA we encourage limiting an architecture to as few services as possible (ideally to one) for a given application domain. This can imply decomposition of the initial application domain into sub-domains (sub-tasks) that each can be effectively supported by a single service. If we assume that the proposed Blackboard-based collaboration is used to realize the initial complex task then the decomposition process can be directed by the constraints defined in Onto⇔SOA and in the proposed collaboration mechanism.

The Blackboard-based composition is also applicable to services that do not fit into the Onto⇔SOA service model, e.g. to Web Services in general. However, additional measures are required to resolve potential conflicts resulting from relaxed or missing Onto⇔SOA constraints (data-model, semantic, protocol and dependency related conflicts, concurrency issues, etc).

One of the distinguishing features of our approach is that, unlike in, e.g. Semantic Web Services, we do not aim to cover as many composition scenarios as possible requiring a complex composition framework. Instead, we limit ourselves to a unified, simple collaboration scenario, and identify requirements (provide design guidelines) which must be met for that scenario to work. Consequently, this limits the applicability of our framework primarily to the design of new services. Many existing services, especially RPC-based ones, are not designed to meet the Onto⇔SOA constraints (document-orientation and session statelessness, to name a few), thus cannot be readily composed with the proposed Blackboard-style mechanism.

Considerable effort is required to design a service according to the Onto⇔SOA requirements. However, the resulting services can be composed using the Blackboard-style with almost no additional effort. It is sufficient to bring the services together and make them

work on a shared repository. In the described use case, and we believe in many other cases, the application-specific workflow logic is minimal. A service consumer constructs an initial description of an application domain that is placed into the blackboard, starting up collaboration between services.

The Onto⇔SOA constraints are enabling factors behind this nearly effortless composition:

- a simple and application-independent connector (synchronous request-response via a stateless session) implies that a service can be placed into a workflow imposing very little restrictions on other participating components.
- service independence (omnipotence) guarantees that when placed into a workflow a service doest not require other services to be present there. This eliminates transitive dependencies and possibly implied conflicts.
- the unified interaction protocol between a service and a consumer simplifies composition of Onto⇔SOA services, because they all share the same intent (inference of new facts), and hence are compatible in that respect.
- a unified data-model is employed in messages allowing us to avoid data-level incompatibility. The semantic-level compatibility is explicitly addressed in Onto⇔SOA with service ontologies.

In the proposed composition scenario we assume that it is the responsibility of a consumer to determine what services are required to realize certain functionality. So far we have not directly addressed the task of service discovery. However, one of the design guidelines defined in Onto⇔SOA is to have the one-to-one relationship between a service and its service ontology (and corresponding concepts). If followed, this makes it trivial for a controller to find a service that corresponds to a concept on a blackboard. We will discuss the discovery problem later in this thesis.

4.6 Conclusions

We have proposed an approach to service collaboration inspired by the ideas from Blackboard Systems extensively studied in the AI community. It is believed that Blackboard Systems provide a very flexible and modular architecture for integration of independent coarsely-grained components. We have integrated Blackboard-style composition into

Onto⇔SOA thus answering the research question: *"How can ontology-enabled services work together?"*.

We have devised a Blackboard-style composition mechanism that uses an application-independent controller component and a homogeneously structured repository (i.e., a blackboard) through which services interact. The proposed approach requires neither an extensive service model nor an explicit workflow specification. It enables composite functionality to emerge by bringing a number of services together and making them interact via a shared data repository.

We have confirmed the feasibility of the proposed mechanism by having used it to compose five Onto⇔SOA services in the described units consistency checking use case. We have observed that the proposed Blackboard-style composition fits particularly well to the document-oriented and ontology-based service model. This implies that in many non-trivial application scenarios, such as the described use case, a service ontology (a description of a document-oriented service interface) is sufficient to enable service composition. We have also outlined possible solutions to a number of potential efficiency problems.

Chapter 5

MoRe: RDF/S-enabled REST Services

Onto⇔SOA is an abstract design framework that can be implemented in different ways. In this chapter we introduce *MoRe* – an implementation of Onto⇔SOA that employs RDFS and RDF as languages for describing service ontologies and messages communicated between REST-like services and their consumers. *MoRe* provides a middleware layer for implementing solutions to the targeted e-Science tasks allowing us to validate the ideas behind Onto⇔SOA. In this chapter we demonstrate *MoRe* using the unit conversion use case and illustrate how an OWL-ontology of units of measure can be integrated into an application by means of *MoRe* services. With *MoRe* we aim to answer the research question: *"How can we integrate RDFS ontologies with REST services?"*. With *MoRe* we also demonstrate how we can further bridge the gap between ontology and software developers.

◇

Maksym Korotkiy and Jan L. Top: MoRe Semantic Web Applications. In proceedings of the End-User Aspects of the Semantic Web Workshop. European Semantic Web Conference. Crete, 2005.

◇

Ultimately, with Semantic Web technologies we aim to significantly improve the experience of Web users. To achieve this we have to provide an environment that enables software developers to advance Web applications beyond state of the art. It is believed that the status of the current Semantic Web technologies does not yet allow to develop applications that realize the full potential of ontologies and the Web [53, 54]. A number of frameworks have been proposed to facilitate the design of Semantic Web applications. These frameworks

range from authoring [61], browsing and annotation frameworks [62, 63] to infrastructures for Semantic Web Services [64].

These approaches introduce ontologies into specific areas of Semantic Web enabled application development. In Onto⇔SOA, like in the Semantic Web, we rely on ontologies to be the keystone element. However, we take a different approach to facilitating software development: we propose to *integrate* ontologies and Service-Oriented Architectures. In Chapter 2 we have introduced Onto⇔SOA as an abstract design approach for enhancing domain alignment of services by means of ontologies. In this chapter we extend the framework into *MoRe*[1] that provides middleware directly applicable in software development. In *MoRe* we employ the RDF and RDFS languages to define service ontologies that describe document-oriented interfaces of REST-like [31] services[2].

Presently, application developers do not benefit much from the increasing availability of domain ontologies. In most cases these ontologies are designed in an application-*independent* way (i.e., without clearly defined application scenarios), and therefore cannot readily provide functional utility sought for by the developers. Moreover, the utility supplied by generic reasoners and query languages associated with ontology languages such as RDFS or OWL are often not efficient and transparent enough to be used in applications. Thus, we can also interpret *MoRe* as a readily exploitable framework that increases utility and usability of RDF/S ontologies by integrating them with REST-services. We believe that the availability of a readily accessible inference service will allow a software developer to faster evaluate an ontology and to incorporate it more rapidly into software. We have elaborated on this ontology-oriented perspective in Onto⇔SOA in Chapter 3.

In this chapter our objective is to answer the above-mentioned research question by operationalizing Onto⇔SOA and to demonstrate:

- how we can improve usability of REST-services by using RDFS ontologies to enhance their *domain alignment* and *loose coupling* characteristics;
- how we can improve usability of RDFS ontologies by increasing their functional utility by attaching inference services based on the REST framework.

We illustrate *MoRe* by applying it to a use case from e-Science, our field of application. In this chapter we use the unit conversion tasks to continue the theme of units of measurement started with the Unit Consistency use case in Chapter 4.

[1] *MoRe* used to be an abbreviation but its original interpretation is not relevant any more.
[2] REST Services is a "lightweight" approach to Web services. More details are in Section 5.2

Traditionally, a software developer would construct a specific algorithm for unit conversion, using an internal database of units and their values in terms of reference units. Such an ad hoc approach is error prone (validation of the conversion is not facilitated in any way) and the result is not very (re-)usable to third parties. With the availability of an ontology of units of measurement the software developer could employ the knowledge captured in this ontology either to validate the implemented conversion logic or even to directly employ the ontology through a corresponding reasoning engine. However, the reasoning capabilities of ontology languages such as RDFS and OWL do not seem suitable for the purpose of unit conversion. The software developer, therefore, is unable to *directly* benefit from ontologies and still has to employ conventional techniques to integrate the unit conversion knowledge into software.

With Service-enabled Ontologies we can extend the ontology of units and measures with a domain-specific inference service. To describe a document-oriented interface to this service we can employ concepts such as `ConversionExpression`, `sourceUnit`, `destinationUnit`. To exploit the unit conversion capability of this service a consumer (e.g., a software application) employs these concepts to express a request document with, for example, the following content:

```
ConversionExpression
    sourceUnit:      inch
    destinationUnit: yard
```

The unit conversion service applies the encapsulated procedural knowledge to this request document and sends the following document back to the application:

```
ConversionExpression
    sourceUnit:      inch
    destinationUnit: yard
    factor:          0.02777778
```

The resulting document contains a new fact (value of the `factor` property) allowing the application to convert inches to yards.

In the rest of this chapter we introduce the Unit Conversion case in Section 5.1 where we outline the main steps a software developer takes to employ an ontology in the application at hand. Then, in Section 5.2 we describe *MoRe* as a realization of Onto⇔SOA that combines RDFS ontologies and REST-like services. In Section 5.3 we elaborate on design and implementation steps for *MoRe* services and their consumers. After that, in Section 5.4 we apply *MoRe* to the Unit Conversion case. Finally, we conclude with Section 5.6.

5.1 Use Case: Conversion of Units of Measurement

To demonstrate the effect of *MoRe* on application and ontology developers we employ the Unit Conversion use case. In this scenario we consider the task of developing an ontology-based unit conversion application – Unit Converter.

To develop Unit Converter, an application developer begins by analyzing the application domain. The developer searches for existing ontologies covering the target domain. Let us assume that the developer discovers an ontology that describes the domain of units of measure. This ontology can be utilized to organize the unit space in a way familiar to the end-user, to select subsets of units that can be converted to each other and finally to determine a conversion expression between two given units of measure. In this chapter we employ the latter task as a use case.

Let us further assume that the ontology describes a number of units of measure (yard, inch, etc) and a conversion factor between each unit and a corresponding reference unit. For example, the ontology states that the yard has the `SI_unit_factor` property with value 0.9144 and for the inch this value is 0.0254. In this example, the conversion value refers to meter (meter is the standard SI-unit for length), implying that 1 yard = 0.9144 meter and 1 inch = 0.0254 meter. We can use these two values to compute a conversion factor between yard and inch: 1 yard = 0.9144 / 0.0254 inch.

At present we can distinguish two main approaches to employ ontologies in software applications. The first approach is to extract relevant information from the ontology into an application-specific form (most often a relational database or programming language code) and then employ traditional techniques to access the data and to apply application logic to them. The pseudo-code in Figure 5.1 demonstrates distinctive features of such an approach. It includes the use of a data query language and computation of the conversion factor in the application.

The main advantage of this approach is that as soon as relevant data is extracted from the ontology, a software developer is able to apply well-known techniques to access the

```
computeConversionFactor (srcUnit, dstUnit, factor)

  srcFactor = db.query(``
          SELECT SI_unit_factor
          FROM UnitsTable
          WHERE unit = $srcUnit''
      ).get(``SI_unit_factor'')

  dstFactor = db.query(``
          SELECT SI_unit_factor
          FROM UnitsTable
          WHERE unit = $dstUnit''
      ).get(...)

  factor = srcFactor / dstFactor
```

Figure 5.1: Pseudo-code of a traditional DB-based approach. Using general purpose ontology middleware leads to a similar solution.

data. The major drawback is that such an approach downgrades an ontology to the level of data, and therefore, does not fully employ domain knowledge captured in the ontology.

The second way to exploit an ontology in an application is to employ general purpose ontology middleware, such as Jena [57] or Sesame [65], that provides storage, inference and query facilities for ontologies. However, in our use case the inference capabilities of the supported ontology languages (RDFS and OWL) do not allow to determine the conversion factors directly (we elaborate on this in Section 5.4). Consequently, in our application scenario the middleware approach would be very similar to the first one, only the queries would be expressed in a different language (e.g., SPARQL [58] instead of SQL) and applied not to a database but to an ontology (repository).

In both cases, the application developer still has to incorporate an essential part of the domain knowledge (`factor = srcFactor / dstFactor`) into the application. Nevertheless, it is natural to expect that the way a conversion factor is computed is part of the units of measure domain. This part should be made available to users of the units ontology.

MoRe, as a particular realization of Service-enabled Ontologies, will allow an ontology

```
computeConversionFactor (srcUnit, dstUnit, factor)

  reqDoc=
      ``ConversionExpression
          sourceUnit $srcUnit
          destUnit   $dstUnit''

  resDoc = UnitsOfMeasureOntology.infer(reqDoc)

  factor = resDoc.getProperty(``factor'')
```

Figure 5.2: Pseu-docode of a *MoRe*-based approach. The user applies an inference service attached to the ontology of units of measure to infer the value of the `factor` property. This property contains a value of a conversion factor for the units of measure specified in the request document (`reqDoc`).

developer to incorporate such domain knowledge as a domain-specific inference procedure connected to concepts from the units of measure domain. If an application developer would have such a *MoRe*-ontology at his disposal, the pseudo-code depicted in Figure 5.2 could be used to utilize it.

The major difference with the previous cases (Figure 5.1) is that the application developer now employs domain knowledge via the inference service provided by the ontology of units of measure. Another difference is that the application developer does not have to use a complex query language to utilize the domain knowledge captured in the ontology. The developer only has to construct a request document using the concepts from the ontology and then to access the inferred value of the `factor` property. We will elaborate on this in Section 5.4. In the coming section we introduce the main ideas behind *MoRe*.

5.2 *MoRe* as RDF/S⇔REST

Onto⇔SOA is a technology and ontology language independent framework. In this section we extend it into *MoRe* – a framework that integrates RDFS ontologies [14] and REST services. With *MoRe* we aim at providing a simple, pragmatic yet efficient framework

for RDF/S-enabled development of REST services. *MoRe* can also be considered from the Service-enabled Ontologies perspective as a mechanism that allows to attach a REST service to an RDFS ontology. We will elaborate on this perspective in Section 3.2.

REST services are inspired by the REST (REpresentational State Transfer) architectural style [31]. They heavily rely on the HTTP protocol not only to provide the transport layer between a service and a consumer but also to supply services with the basic set of CRUD (create, read, update, delete) operations. These operations are mapped to the corresponding HTTP request methods: PUT, GET, UPDATE and DELETE. The services that implement all these operations are often referred to as REST-full Services. In *MoRe*, and Onto⇔SOA in general, we limit a service to the single operation of inference that will be implemented using the POST method. The main motivation behind using the POST method is implementation convenience. However, with this we also acknowledge that the inference operation cannot be directly mapped to any of the CRUD actions.

REST Services define neither a standard service description framework (e.g., WSDL) nor a standard message format (e.g., SOAP). REST services often employ XML to express messages and XML Schema as a schema definition language. Since there are no constraints on schema and message languages, it is straightforward to design a REST service that uses an ontology-based schema definition and messaging.

Another important feature of REST is that this framework has been designed to fulfill requirements of the Web [66]. REST services, therefore, gain many architectural properties of the Web that have already proven to be successful. Additionally, due to the strong connections to the Web, REST and RDF/S languages exhibit a significant degree of conceptual compatibility. More specifically, the notion of *a resource* plays the central role in the Web, the REST framework and the RDF/S languages.

REST services require little infrastructure in addition to what is already provided by the Web. Despite the fact that REST services are not the standard and less widely publicized (than for example the standard WSDL/SOAP Web Services) they are known in some cases to be preferred over Web Services. For example, Amazon provides interfaces for both WSDL/SOAP Web Services and REST services, and 85% of the usage is on the REST interface[3].

In *MoRe* we use an RDF/S document as an ontology-based message exchanged between a service and its consumer. All documents share the same structure (a collection of subject-property-object triples) and syntax (XML-RDF). Following the Onto⇔SOA guidelines, a

[3] http://www.oreillynet.com/pub/wlg/3005

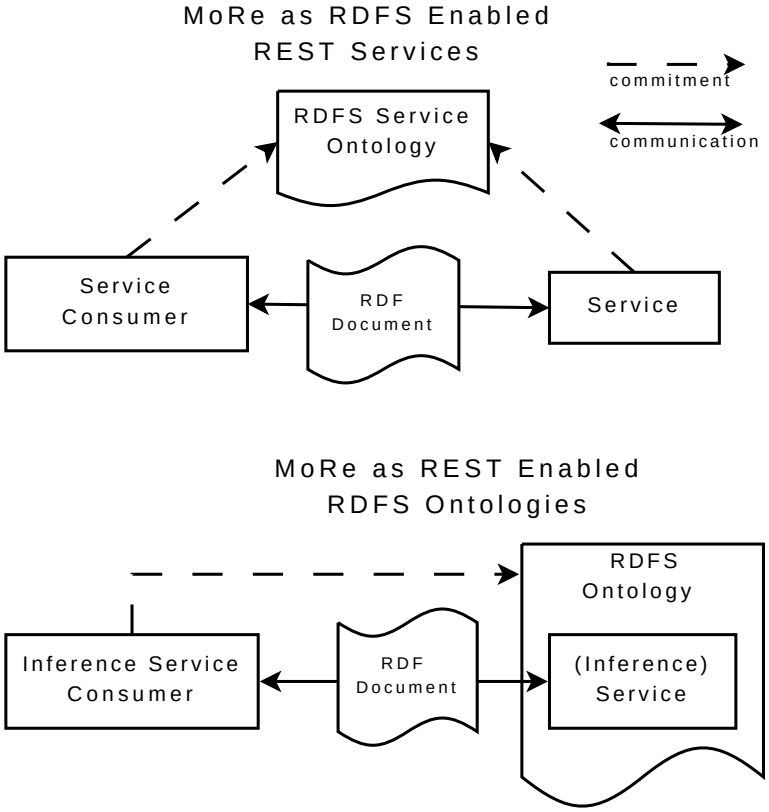

Figure 5.3: *MoRe* from the Ontology-enabled Services and Service-enabled Ontologies perspectives. The main difference between the two perspectives is whether the focus is on a service or on an ontology.

document describes a particular situation in a service domain and is created using the terminology specified by a service ontology. The service is invoked in a REST-manner via a HTTP POST request with one XML-RDF document attached.

In *MoRe* every service ontology (service schema) contains a service URL that can be used to invoke the service. A service ontology can be published on the Web and discovered along with the referenced service via conventional Web-search techniques.

In a typical interaction scenario, a consumer inspects the service ontology and composes an RDF document that contains triples describing a domain situation at hand. Next, this RDF document is communicated to the service referred to from the discovered ontology.

The service applies its domain expertise to the document inferring new facts. The result, an another RDF document, is delivered back to the consumer.

In *MoRe* we encourage creating an RDFS service ontology exclusive to a target application domain. Concepts from such a service ontology can be related to concepts from external ontologies (either application-independent or service ontologies) by means of ontology mapping methods. As we suggested in Chapter 3 this should be done with care because of possible conflicts resulting from interaction between the pragmatic service-oriented application semantics of service ontologies and formal semantics of an external ontology.

The formal semantics of a non-restrictive ontology language such as RDFS does not provide means to detect conflicts. This implies that in *MoRe* service ontologies we can safely import concepts defined in external RDFS ontologies. However, if we were to import concepts from an ontology with restrictive semantics (e.g., from an OWL ontology) then we would have to take extra care not to violate the formal semantics.

5.3 Designing *MoRe* Services and Consumers

Figure 5.4 outlines the main processing steps taking place inside a *MoRe* service and a consumer. Figure 5.5 illustrates the processing steps typical for a Web service implementation. This enables us to compare the two approaches.

A *MoRe*, or more generally Onto⇔SOA, service processes an incoming document using the following steps (we use the Unit Conversion case to illustrate the steps).

1. The service receives an RDF document containing a description of an initial situation in the Unit Conversion application domain. The document is expressed using terms from the Unit Conversion service ontology and serialized using the RDF-XML syntax. We refer to such RDF-XML serialization as an *external conceptual model* (of an application domain).

2. The received RDF-XML document must be parsed and converted by the service into an *internal conceptual model* to enable programmatic access to its content. In the Unit Conversion service we employ the in-memory RDF model provided by the Jena API [57]. A database-backed model can also be employed as long as the RDF API of choice supports it. The internal and external conceptual models are equivalent and, therefore, we can automatically transform them into each other by means of available RDF/S middleware such as Jena. In Onto⇔SOA we assume that both conceptual models are domain aligned.

3. The in-memory RDF model of an instance of the Unit Conversion case is then mapped to *an internal domain model* implemented in the service. The internal domain

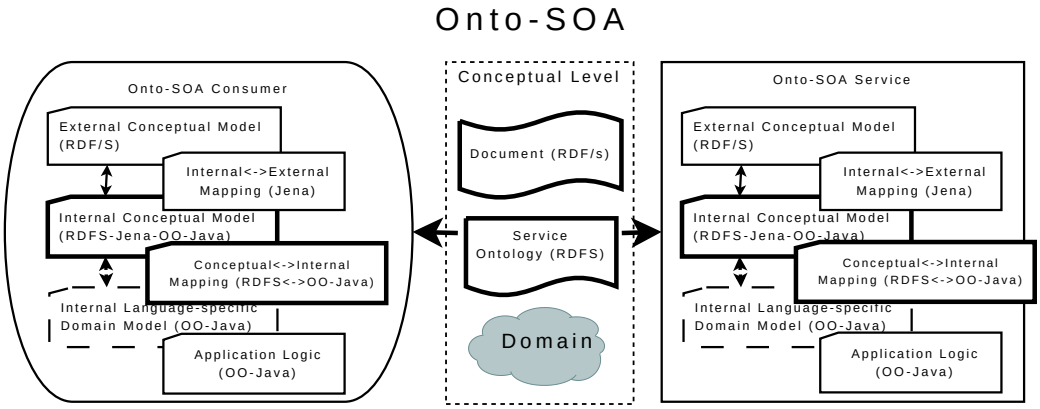

Figure 5.4: The *MoRe* process cycle. In the parentheses we refer to a specific technology employed in the Unit Conversion service and its consumer – the Unit Converter demo application.

model must be computationally feasible and, consequently, is effected by such *non-domain-aligned* factors as an implementation language (e.g., Java), available APIs, non-functional requirements (performance, security, scalability, etc) and the algorithms employed. The conceptual and internal domain models in most cases are not equivalent because they are affected by different (and often incompatible) factors. Mapping between the two models cannot be done automatically and requires implementation-specific means that will sacrifice domain-alignment of conceptual models for increased efficiency of the internal domain model.

4. The Unit Conversion service logic is then applied to the internal domain model of the domain. New facts inferred by the service (the value of the unit conversion coefficient) extend the internal domain model. The extended internal model is then transformed by ad hoc means into the internal conceptual model. Finally, the internal conceptual model is automatically transformed into the external conceptual model that communicates the newly inferred facts to a consumer. A consumer then follows steps 1-3 to interpret the response, and reverses the order of steps to invoke the service again, if necessary.

Generally speaking, the internal domain models of a consumer and a service will be different. In the Unit Converter demo application, that acts as a *MoRe* consumer, the internal domain model is relatively simple. The user's actions are translated almost directly into

the conceptual model, not incurring any additional overhead. On the other hand, the Unit Converter service cannot ignore the overhead of a conceptual model and requires an optimized internal domain model (the Document Retrieval service described in Chapter 2 is an even better example). A service is likely to require a model optimized for high throughput to perform well on requests from multiple consumers.

An internal domain model can be reduced by implementing application logic to operate as directly as possible over a conceptual model. Ontology tools and middleware (query languages, inference engines, rule languages, etc) simplify the use of conceptual models from within application logic. For example, by means of a query language (e.g., SPARQL [58]) RDF/S models can be processed in a rather efficient way directly from application logic. In the Unit Conversion case this would allow us to combine results of several RDF/S queries executed over the ontology of units of measure to infer conversion factors. However, an implementation-specific internal model still, potentially, offers better performance. Furthermore, the more specialized (and, therefore, more capable) ontology middleware gets, the more similar it becomes to implementation level tools such as (high level) programming languages. In such a case an internal domain model will not be reduced but rather transferred from a programming language to ontology middleware, an ontology query language for example.

To decrease the possibility of misalignment between conceptual and internal models a direct operation over a conceptual model should be preferred. If this is impossible because of performance considerations, then a direct operation can be approximated with intermediate coarse updates to the conceptual model. The discrete updates allow to re-align intermediate computational states to the conceptual model.

Now let us consider the processing steps taking place between a Web service and its consumer. In Web Services (Figure 5.5) the communication between a consumer and a service takes place at the data level. At the service side an external data model (XML data-type definitions) is automatically generated from an *internal model* by means of programming language-specific Web Service middleware (e.g., Apache Axis[4], Java EE Web Service tools[5]).

At the consumer side the XML data-type definitions contained in a WSDL description of a service are used to automatically generate an *internal stub model*. This model is either mapped to the consumer's internal model or is employed directly as is. Since conceptual

[4]http://ws.apache.org/axis/
[5]http://java.sun.com/webservices/

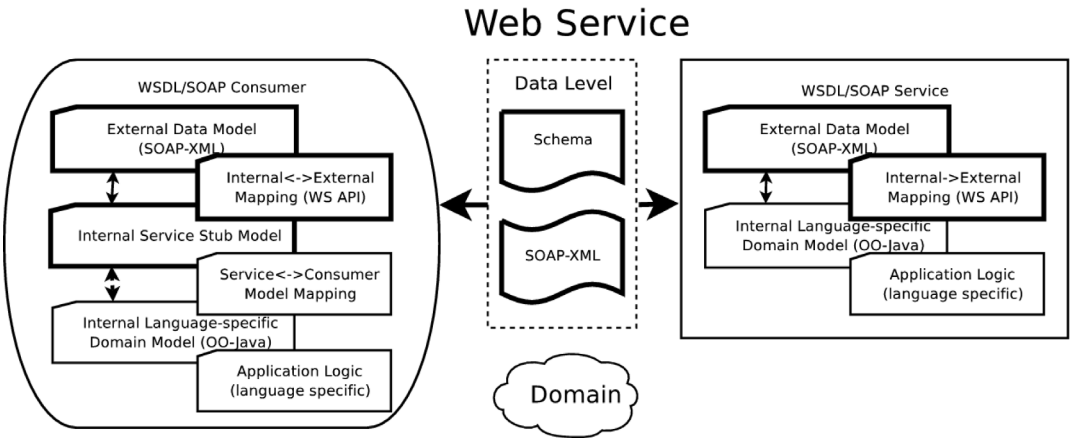

Figure 5.5: Outline of the processing cycle in Web Services.

models are not explicitly present in Web Services and domain alignment of a service is not enforced, a conceptual gap and misalignment can easily arise between the internal models of the service and the consumer.

In a service created for a simple application domain, sufficient domain alignment can be achieved without a conceptual model. In such a case a data model for that domain would play the role of a conceptual model. However, we believe that the more complex application domain gets – the greater becomes discrepancy between conceptual and data models because the latter is primarily concerned with computationally efficient algorithms employed rather than with domain concepts.

In Semantic Web Services (SWS) an ontology provides conceptualization for a model of a Web service. Ontologies in SWS can reduce the conceptual gap between a consumer and a service. However, since domain alignment is not enforced in Web Services, the ontology is likely to address implementation details of the service rather than the conceptual domain model. This reduces the effectiveness of ontologies with respect to aligning a service and its consumer. Onto⇔SOA addresses the potential conceptual misalignment between a service and its consumer by explicitly introducing conceptual models into their processing cycle and facilitating domain alignment of a service during its design.

5.4 Solution to the Use Case

In this section we elaborate on the unit conversion scenario presented earlier and describe how a *MoRe* service ontology can be designed for use in the corresponding ontology-enabled *MoRe* service. We will also demonstrate how an end-user application should be designed to exploit RDF/S-enabled REST services.

5.4.1 Unit Conversion Service Ontology

In the Unit Conversion service ontology we will reuse concepts defined in UnitDim[6] – an external OWL ontology of units of measurement. One of the usability problems we faced during the initial attempts to exploit UnitDim was the difficulty to access domain knowledge expressed in terms of OWL restrictions. Another reason to extend UnitDim with an application-specific inference service is the inability of general purpose OWL reasoners to infer conversion expressions between units.

We have addressed these two usability bottlenecks by creating a *MoRe*-based Unit Conversion service. An application-specific (service) ontology is less general and, therefore less reusable than a conventional application independent ontology such as UnitDim. However, by sacrificing some reusability we expect to gain additional utility and make the service ontology more usable. This demonstrates the general approach taken in Onto⇔SOA: rebalance reusability and usability towards the latter making services and ontologies more attractive to end-users. Additionally, by expanding an existing ontology with a service, we demonstrate how the knowledge captured in the ontology can be made more readily available to end-users (e.g., software developers) while preserving the generality of the original ontology.

Unit Conversion Service Ontology

In UnitDim the `SI_unit_factor` property describes a quantitative relation between every unit and its counterpart from the SI System of Units. Two such relations can be combined to determine a conversion factor between any two units. Unfortunately, there is no feasible way to achieve that using general OWL-reasoners, given a subset of N units, such that any two units can be converted to each other. In OWL we would have to use the complete enumeration of conversion factors. This would result in $N*(N-1)$ property values instead

[6] Rijgersberg, H., Top, J.: UnitDim: an ontology of physical units and quantities. http://www.atoapps.nl/foodinformatics. Sec. News (2004)

of the more feasible N `SI_unit_factor` values combined with a capability to infer the rest.

From the ontological point of view, the Unit Conversion service ontology can be seen to extend UnitDim to infer a conversion expression between two units. To achieve this we introduce the `ConversionExpression` concept to represent a ternary relation between two units and the corresponding conversion factor. The `ConversionExpression` class has three properties:

- `hasSource` – points to a source unit, the unit to which we apply the conversion factor;
- `hasDestination` – points to the destination unit to which the `hasSource`-unit is to be converted to;
- `hasFactor` – contains the numerical value of the conversion factor.

We design the Unit Conversion service in such a way that for every instance of `ConversionExpression` contained in an input document, the service determines `SI_unit_factors` for both the source and destination unit and combines them to compute the corresponding `hasFactor` value. More precisely:

$$Factor_{srcUnit,dstUnit} = factor_{SIUnit,srcUnit}/factor_{SIUnit,dstUnit}.$$

The computed factor allows us to use the following conversion expression in application software

$$srcUnit = Factor_{srcUnit,dstUnit} \cdot dstUnit.$$

Note that the Unit Conversion service ontology could have contained an OWL reasoner instead of the application-specific inference service if the OWL reasoner were able to provide the required functionality. In that case we could see the conversion service as a wrapper around UnitDim and the standard OWL reasoning mechanism.

A consumer of the Unit Conversion service will use the terminology defined in the service ontology to create a document describing an instance of the `ConversionExpression` class (Figure 5.8). This input document can then be communicated to the Unit Conversion service. The service infers the value of the `hasFactor` property and adds it as a new fact to the document, which is then communicated back to the application. The application updates its state according to the newly obtained facts.

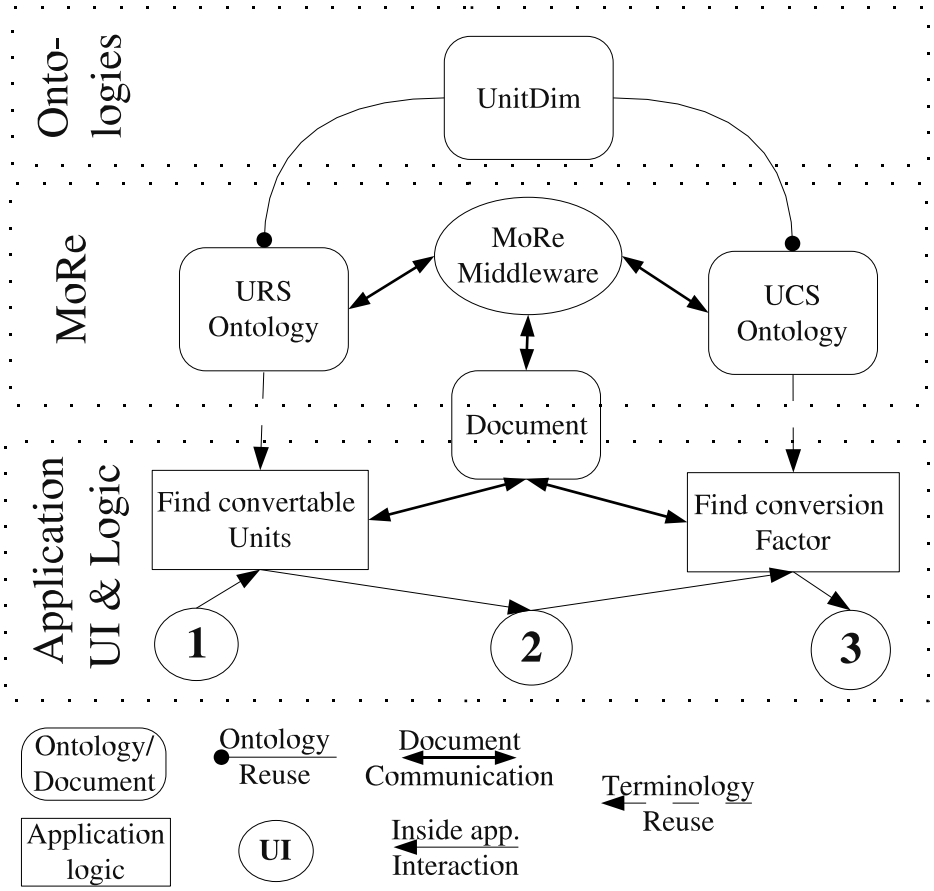

Figure 5.6: The architecture of the Unit Converter application consists of three layers: the layer of application-independent ontologies (UnitDim), the layer *MoRe*-based services (auxiliary Unit Retrieval and Unit Conversion services represented with their service ontologies: URS and UCS) and the application layer (see Figure 5.7).

The above scenario demonstrates how the *MoRe*-framework allows the application developer to abstract away remote procedure calls. The essential point is that since we maintain proper alignment between the service and the Unit Conversion service, the application developer can stay at the conceptual level when utilizing the service.

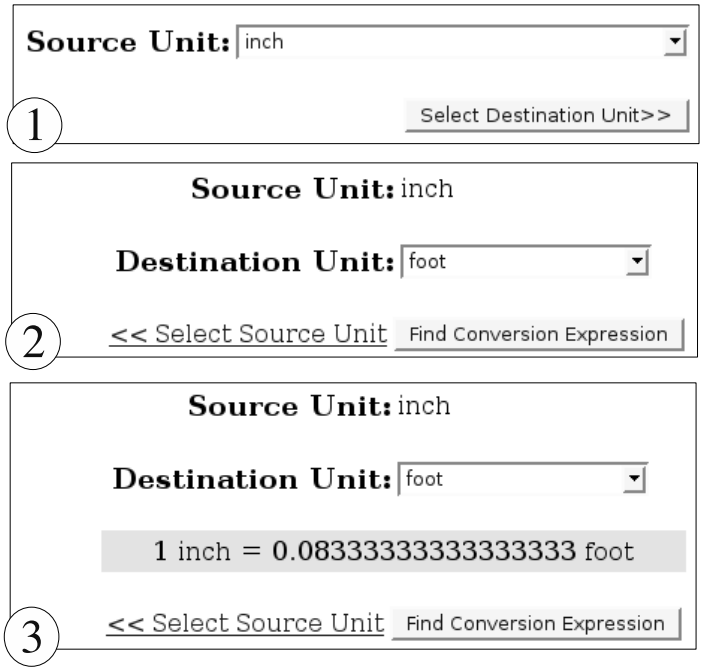

Figure 5.7: The user interface of the Unit Converter application. Numbers on the UI screenshots correspond to the UI components in the application layer on Figure 5.6.

5.4.2 Building the Unit Converter Application

We have employed the Unit Conversion service (together with an auxiliary Unit Retrieval service) to support the Unit Converter application. Figure 5.6 depicts the architecture of Unit Converter. In the figure we can see three distinct layers:

- The layer of traditional, application-independent ontologies is at the top. The Unit-Dim ontology is the only application-independent ontology employed in our application.

- The application layer is at the bottom. Inside we can distinguish two sub-layers that represent the application logic and the user interface (UI). The UI sub-layer accepts user's commands and displays the application state. In our case the user is able to perform three actions: select source and destination units (steps 1 and 2 in Figure 5.7),

```
  rdf:Description rdf:about = ``ceInst0''
*  hasFactor 0.0277776
   hasSource rdf:resource = ``inch''
   rdf:type rdf:resource = ``ConversionExpression''
   hasDestination rdf:resource = ``yard''
  rdf:Description
```

Figure 5.8: An example of a document communicated between a consumer (the application) and *MoRe* services. Initially the document does not contain a fact marked with "*" that is added by the inference service.

and ask for a conversion expression (step 3 on Figure 5.7). Two of these actions are connected to the application logic layer that contains two components responsible for determining 1) a set of convertible units and 2) a conversion expression between two units.

- The application and ontology layers are connected through the *MoRe*-layer. The Unit Retrieval service assists in accessing the domain knowledge captured in UnitDim. The Unit Conversion service extends UnitDim with new concepts (`Conversion-Expression`, `hasFactor`, etc) and provides an inference service capturing domain knowledge about these concepts.

  The application layer interacts with the *MoRe* layer in two ways. First, it employs ontological terminology defined in service ontologies for interfacing purposes (documents). Second, it communicates with *MoRe*-middleware. The middleware is responsible for delivering the input document to the corresponding ontology (its reasoning service) and communicating the output document back to the application layer.

Figure 5.8 contains fragments of input and output documents communicated between the "Find Conversion Factor" component (acting as a consumer) and the Unit Conversion service from the *MoRe* layer. The input document does not contain a fact marked with "*". The value of the `hasFactor` property is computed by the Unit Conversion service and added to the input document. The initial situation described in an input document reflects the state of the application after the user has selected the source and destination units. The

output document contains a new fact (`hasFactor` property value in our case) which is used to update the application state (step 3 in Figure 5.6).

5.5 Discussion

As we have mentioned in Section 5.5 XML Schema Datatypes (XSD) employed in WSDL descriptions allow for a relatively straightforward translation between a Web service interface and an interface expressed in programming languages such Java and C#. Indeed, many state of the art Web Services middleware and development tools such as JAX-WS[7] and XFire[8] provide extensive support for automatic generation of a WSDL schema from an (annotated) interface of a Web service implementation. These tools can also generate client-side proxies for a particular programming language from a WSDL schema (this process is also known as *WSDL import*). Such facilities are very much appreciated by software developers because they eliminate rather complex steps of creating a WSDL schema for a service as well as creating a client-side implementation of domain objects employed in a service schema.

Both WSDL generation and import reflect the Web Services view on a service as a software component invocable via the Web. Actually, the approach taken in WSDL can be seen as one of the numerous IDLs (interface definition languages) employed in remote procedure (or objects) invocation software. In Section 2.2 we have elaborated on the different between this view and the service model underlying Onto⇔SOA. According to this model a service is a software component that is designed with domain alignment and loose coupling in mind. We believe that with domain alignment we can significantly reduce complexity of a service interface, and therefore, reduce the need for advanced tool support for creating and importing a service schema in the following way.

Presently the middleware capabilities of *MoRe* cover the transport layer. *MoRe* facilitates the translation between an in memory RDF model and its external representation (i.e. the translation between the internal and external conceptual models) and communicating the external model via HTTP between a service and a consumer. Unlike the above-mentioned Web Services middleware, *MoRe* does not assist service and consumer developers in mapping the internal conceptual model to the implementation-specific domain model. Although we do not explicitly address the problem of translation between RDFS models and models

[7] https://jax-ws.dev.java.net
[8] http://xfire.codehaus.org/Web+Service+Design

expressed in a programming language, there exist a number of tools capable of providing this functionality at least partially.

For example, in order to generate object-oriented Java proxies for an RDF data model a framework such as RDFReactor[9] can be used. RDFReactor can generate proxy objects through which a software developer can interact with the domain model using familiar programming language expressions. For example, instead of explicitly modifying the underlying conceptual models with a statement like

```
addTriple(unitConversionExpressionURI, hasFactorPropertyURI, ``1.25'')
```

the developer can manipulate domain concepts in a more familiar way as if they were programming language objects

```
unitConversionExpression.setFactor(1.25).
```

At the moment of writing we are not aware about approaches capable of generating an RDF/S model given a set of, for example, Java classes. We believe that the ideas underlying the FDR2 approach described in Chapter 6 can be used to map a domain model expressed in, for example, Java classes to an RDFS ontology. We can map Java classes to RDFS classes and introduce RDF properties corresponding to either public fields or accessor methods (also known as setters and getters) of the Java class. The generated RDF/S classes and properties could then be employed directly as a service ontology or mapped (through a procedure described in FDR2) to an existing one.

Overall, we believe that the middleware capabilities of *MoRe* can be extended to facilitate import and generation of service ontologies. For this either approaches such as RDFReactor can be directly employed or an approach such as FDR2 can be used after some adaptation.

5.6 Conclusions

We have proposed *MoRe* – a framework for development of RDF/S-based REST-service as an answer to the research question: *"How can we integrate RDFS ontologies with REST*

[9] http://semanticweb.org/wiki/RDFReactor

services?". *MoRe* is an implementation of Onto⇔SOA that uses a restricted document-oriented REST-based service model and employs RDFS ontologies as conceptual service interfaces. *MoRe* aims to provide a simple and pragmatic foundation for development of ontology-based Web applications.

We believe that *MoRe* helps us to bridge the gap between ontological domain knowledge and software development in practice. On the one hand, it provides a pragmatic application-driven view on domain ontologies. On the other hand, it facilitates software development by integrating domain-specific inference services into software applications.

Chapter 6

FDR2: Linking Relational Data and RDF Models

A vast amount of information and data sources is stored in a tabular, relational-like form and is not directly accessible to ontology-enabled software. For example, in many research organizations results from experiments are stored in spreadsheets. Both managing and utilizing of such spreadsheets in ontology-enabled environments requires them to be represented in a proper form. In this chapter we propose an approach to integrate such tabular information sources with RDFS-aware systems. With this approach we aim to answer the research question: *"How can we integrate tabular data with an ontology-ready data-model such as RDF/S?"*. The proposed solution is purely RDF/S-based. We use RDF/S as a mechanism to specify and perform, by means of RDFS reasoning, linking of relational data to a predefined domain (or service) ontology. The approach does not require any additional run-time transformation components except an RDFS reasoner. The approach completely preserves the original structure of the relational data. This ensures complete and consistent RDF/S-enabled access to relational-like data.

◇

Maksym Korotkiy and Jan L. Top: From Relational Data to RDFS models. In proceedings of the International Conference on Web Engineering. Munich, 2004.

◇

The RDF and RDFS languages have been proposed as relatively simple yet flexible languages for expressing ontologies. These languages provide a number of widely supported syntaxes, a unified data model and enable separation of data (RDF) from meta-data (RDFS). RDF/S have formed the foundation for the family of Web ontology languages aimed to improve sharing and reuse of knowledge and information sources.

The formal acceptance of RDF/S by W3C [67] stimulates their utilization in many areas and by many organizations. However, in spite of the increasing acceptance of RDF/S, this is

still a new technology. Most information resources are not (yet) available in the RDF/S format. The relational data model, on the other hand, is well established and widely accepted. It has been successfully employed for decades and is currently supported by thousands of applications ranging from simple spreadsheets to complex relational databases.

In particular in many research organizations a considerable amount of data (and knowledge) is contained in experimental results that are in many cases expressed in a relation-like form in, for example, databases or spreadsheets. Information resources expressed within the relational model have neither a unified syntax nor a standard way to attach meta-data. These are crucial bottlenecks for (re-)usability of such resources. One of the tasks of e-Science is to facilitate management, sharing and utilization of these sources of knowledge at each step of the scientific process.

In this chapter we use the term *relational data model* to refer to data organized as a collection of records (tuples or rows) that is normally represented as a table. We would like to note that unlike in the *relational database model* we require a collection of records neither to adhere to the Entity-Relationship (ER) model nor to be normalized. More specifically, we do not assume that a record consists of *attributes* and represents an *entity*. In our approach we consider a record to be an instance of a complex relation defined over a set of *concepts*.

We employ such a broad view on tabular data because our experience indicates that in many cases users neither follow the ER model nor normalize their data. Scientists, for example, tend to organize data in an "intuitive" tabular way supported by spreadsheet applications.

The problem of mapping relational database models to other models is not new. Much work has been done in the database community to reverse engineer relational databases to extract Entity-Relationship [68], Extended Entity-Relationship [69] and Object-Oriented models. In our work we target the problem of integrating the more general (less constrained) relational data model and the RDFS model. Also, a substantial amount of research has been done to convert RDF/S data and models into relational [70] or Object-Oriented models. In our work we focus on the reverse case.

This chapter is organized as follows. Section 6.1 provides more details about the linking problem and its context. The proposed approach is outlined in Section 6.2 and then elaborated upon using a detailed example in Section 6.3. Section 6.4 demonstrates how the proposed technique can be extended to handle database-specific features such as cross-table references via primary and foreign keys. In Section 6.5 we discuss some features and limitations of the presented method, followed by an overview of possible extensions

and directions for future research. Related research is discussed in Section 6.6 and, finally, Section 6.7 summarizes this chapter.

6.1 Problem Context

We aim to solve the problem of allowing ontology-enabled systems to access results from scientific experiments captured in a tabular-like form (e.g., in spreadsheets). An ontology-enabled system, for example an Onto⇔SOA-based one, can then be employed to improve reusability and manageability of these scientific data. Such a system can assist in making the transition from the traditional experimental science environment to e-Science, facilitating collaboration between scientists and automated reasoning. More specifically, with the FDR2 approach proposed in this chapter we link a tabular data representation to the RDF data-model and RDFS ontologies.

The main objective of linking the relational and RDF/S models is to allow RDFS-based querying of the relational data. The original relational data must be made available to an RDFS reasoner and become query-able with a vocabulary specified by a domain ontology. This can be employed, for example, in Onto⇔SOA to facilitate the generation of ontology-based messages from existing relational data. More specifically, in a *MoRe*-based architecture the users can employ the FDR2 framework to create RDF messages using existing tabular data.

Additionally, an approach to linking the relational and RDF/S models should fulfill the following requirements:

- *reversibility*: it should be possible to recreate the underlying data from its RDF/S representation. This is required to completely preserve information and to enable the user to reach back to the relational data model to modify or reuse the data in its original form. For example, researchers prefer to use a spreadsheet to analyse the data, whereas the data can be stored in an RDF/S repository;

- to be practically feasible, a solution for the linking problem should be easily applicable within an RDF/S-based system without significant design or development efforts;

- to be generic and easily extensible to support additional, for example, *database-specific* features such as foreign keys (see Section 6.4).

The next two subsections contain a detailed description of the approach. The main ideas

behind the proposed linking technique are set out in Section 6.2, and a detailed example-based explanation is given in Section 6.3.

6.2 FDR2 Approach

Let us assume that we have a set of data, expressed in a relational way – a collection of records (e.g., a spreadsheet table), and a target ontology (a domain or service one) expressed in RDFS. We also assume that this collection of records contains a header record (a table header) that provides labels for the ordered elements of a record.

We break down the integration of the relational and RDF/S model into three steps:

1. At the schema level we explicitly define the underlying relation represented within a record. We refer to this definition of the relation as the *relational schema* and use RDFS to express it. The relational schema is required to express and preserve *structural (relational) semantics* of the original data. The relational schema makes it possible to link the relational data to the RDFS ontology.

2. At the data level we use RDF to express the actual content of the records according to the relational schema created in the previous step.

3. Link the relational schema to the target RDFS ontology. Since the linking cannot be done automatically due to the undefined meaning of the relational data, the user has to define the relationships between the relational schema and the ontology. Preservation of the original data structure makes it possible to track changes done on the ontological level back to the original relational level.

Below we elaborate upon these three steps and describe additional actions required to enable run-time interoperability between a relational schema and an RDFS ontology. For readability reasons we have left out syntactical details of the constructed RDF/S documents.

Step 1: Build an RDFS representation of the relational schema. As we have mentioned in the introduction, we do not assume that the elements of a record represent attributes and the record itself – an entity. We build a relational schema upon the notion of RDFS Class.

At this stage we analyze only the header record. We assume that every header element determines the name of a class that is defined as the collection of elements that have the same position as the corresponding header element. The members of the set of all classes C define the relationship $R = C_0, C_1, ..., C_n$ underlying the collection of records.

All other possible relationships between the members of C are made explicit by means of what we will refer to as *virtual relations*. The RDF data-model explicitly supports binary relations only. Since any binary relation defined over set A is a subset of $A \times A$, where $\times$ denotes the Cartesian product, we can use $C \times C$ as the set of all binary virtual relations (*virtual properties*) defined over the classes presented in the relational schema. We believe that in most cases it is sufficient to explicate only virtual properties (as it will be shown in the example). Construction of more complex (e.g., ternary) relationships and their representation with RDF/S is also possible but would require a mapping mechanism that cannot be implemented with standard RDFS reasoning.

After this step we have obtained a relational schema that is an RDFS representation of *(i)* a definition of all classes involved, *(ii)* a definition of the relationship underlying the collection of records, and *(iii)* a definition of the class properties. The resulting relational schema serves as a compact representation of the data and will be directly connected to the target ontology in step 3.

Step 2: Construct an RDF representation of the relational data. At this step we are dealing with the actual data – record elements (cell values). We consider every record to be an instance of R. Every record element is in turn represented as an instance of a class that corresponds to the element's position in R (i.e. in a row). In addition, we instantiate all *virtual relations* defined in the previous step.

This step provides us with instances that represent *(i)* record elements and *(ii)* once hidden but now explicit relationships between elements of a record. At this point we are already able to use general-purpose RDF/S repositories and querying engines to access relational data. However, we still cannot employ the vocabulary defined in the ontology to do that.

Step 3: The user links the relational schema with the domain ontology. The user links RDFS concepts from the relational schema (classes and virtual properties) to concepts in the ontology by means of the `rdfs:subClassOf` and `rdfs:subPropertyOf` properties. A set of all such links from ontological definitions to the relational schema constitutes what we will refer to as *Relational-RDFS Map* (RDMap).

From the RDF/S serializations of the relational data and the RDMap, an RDFS reasoner can deduct entailments that will link the relational schema to concepts defined in the ontology. We will illustrate this with an example in the coming section.

The complete workflow of linking relational data and ontology, and then querying it is

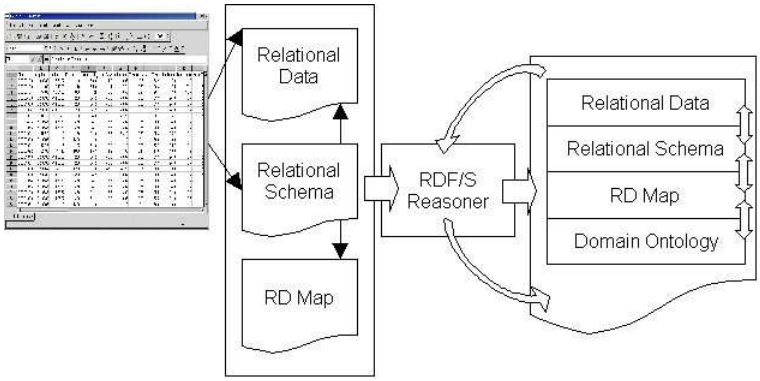

Figure 6.1: An example of a FDR2 workflow. The original spreadsheet is represented as a relational schema in RDFS and a collection of RDF data. Then, these RDF/S artifacts are transformed by an RDFS reasoner into a single model that can be queried with ontological terminology.

shown on Figure 6.1. First, the original spreadsheet is represented as a relational schema expressed in RDFS and a collection of data expressed in RDF. Then, these RDF/S artifacts are transformed by an RDFS reasoner into a single model that, finally, can be queried with terminology defined in an ontology.

6.3 Applying FDR2

In the previous subsection we have introduced the main ideas behind our approach. The present subsection provides more details illustrated with an example.

Let us assume that we have a simple table with data from some experiment on dairy products:

Product	**Judge**	**Supervisor**
milk	Pete	Heinz
cream	Charles	Pete

and a domain ontology expressed in RDFS. This domain ontology defines classes `DO#Product`, `DO#Judge`, `DO#Supervisor` and `DO#Person` with `DO#Judge` and `DO#Supervisor` being subclasses of it. Also there are two properties `DO#isJudgedBy` and

DO#isSupervisedBy. Both properties have the same domain – DO#Product. Their ranges are respectively DO#Judge and DO#Supervisor.

The goal is to enable data querying from the given table employing concepts defined in the domain ontology. For example, if we ask for all instances of DO#Person and want to see Pete, Charles and Heinz being returned. In order to achieve that we follow the steps described in the previous subsection.

For clarity reasons we prefix concept names with one of the labels: fdr2, DO, rdf and rdfs. The fdr2 prefix identifies resources automatically created in the relational schema and relational data representations. The DO prefix indicates that a concept belongs to a domain ontology[1].

Step 1 creates a relational schema and expresses it in RDFS as shown in Figure 6.2. The schema consists of three major parts:

- **Classes definition** – three classes are declared: fdr2#Product, fdr2#Judge and
  fdr2#Supervisor. These classes are extensionally defined with corresponding record elements. For example, class fdr2#Product has an extension that consists of two instances fdr2#milk and fdr2#cream.

- **Definition of *virtual properties*** contains a declaration of six *virtual relations*:

  - fdr2#vpPRODUCT-SUPERVISOR,
  - fdr2#vpPRODUCT-JUDGE,
  - fdr2#vpSUPERVISOR-JUDGE,
  - fdr2#vpSUPERVISOR-PRODUCT,
  - fdr2#vpJUDGE-SUPERVISOR,
  - fdr2#vpJUDGE-PRODUCT.

  These *virtual properties* are a result of a Cartesian product of the fdr2#Product, fdr2#Judge, fdr2#Supervisor set on itself. For pragmatic reasons, we ignore reflexive *virtual properties* such as fdr2#vpPRODUCT-PRODUCT.

[1]To construct a valid URI from a prefixed identifier one should replace a prefix with a (base) URI. For example, a URI for fdr2#RelationDefinition is constructed by replacing the fdr2 prefix with http://www.cs.vu.nl/~maksym/fdr2. Prefixes rdf and rdfs should be replaced with http://www.w3.org/1999/02/22-rdf-syntax-ns and http://www.w3.org/2000/01/rdf-schema respectively.

Class definitions

```
fdr2#Judge
    type:      Class

fdr2#product
    type:      Class
```

Virtual property definitions

```
fdr2#vpJUDGE-PRODUCT
    type:      rdf#Property
    domain:    fdr2#Judge
    range:     fdr2#Product

fdr2#vpJUDGE-SUPERVISOR
    type:      rdf#Property
    domain:    fdr2#Judge
    range:     fdr2#Supervisor

fdr2#vpPRODUCT-JUDGE
    type:      rdf#Property
    domain:    fdr2#Product
    range:     fdr2#JUDGE

fdr2#vpPRODUCT-SUPERVISOR
    type:      rdf#Property
    domain:    fdr2#Product
    range:     fdr2#Supervisor
```

Figure 6.2: An example of a relational schema automatically generated by FDR2.

- **Relation definition** declares the relation represented by the original tabular data. `fdr2#RelationDefinition` employs `fdr2:isDefinedOver` property to refer to sequences of classes over which the relation is defined.

Step 2 expresses the actual data (values of record elements) in RDF according to the schema defined in the previous step. Figure 6.5 shows an RDF source that consists of two major sections:

Class links

```
fdr2#Judge
    subClassOf:      DO#Judge

fdr2#Supervisor
    subClassOf:      DO#Supervisor

fdr2#Product
    subClassOf:      DO#Product
```

Property links

```
fdr2#vpPRODUCT-SUPERVISOR
    subPropertyOf:      DO#has_supervisor

fdr2#vpPRODUCT-JUDGE
    subPropertyOf:      DO#has_judge
```

Figure 6.3: An example of a manually created RDMap.

- `fdr2#RelationInstantiation` is an ordered collection of all records. The `fdr2#definedWith` property points to the corresponding relational schema and `fdr2:definedOver` refers to the actual sequence of records.
- The sequence of records consists of `fdr2#RowSeq0` and `fdr2#RowSeq1` which themselves are sequences of record elements (cells).
- A record element is represented as an instance of one of the classes `fdr2#Product`, `fdr2#Judge` and `fdr2#Supervisor`. These instances have their *virtual properties* filled in with values from the corresponding record elements.

Step 3 The user defines an RDMap – an RDF schema that links the concepts from the relation schema to the concepts in the DO. As shown on Figure 6.3, the linking is done by means of `rdfs:subClassOf` and `rdfs:subPropertyOf relationships`.

Having obtained three models (the relational schema, the relational data and the RDMap), we can run queries using a RDFS query engine capable of RDFS inference. Figure 6.4

Properties generated by FDR2

```
fdr2#pete
    type:                fdr2#Judge
    type:                fdr2#Supervisor
    fdr2#vpJUDGE-PRODUCT:       fdr2#milk
    fdr2#vpJUDGE-SUPERVISOR:    fdr2#heinz
    fdr2#vpSUPERVISOR-PRODUCT: fdr2#cream
    fdr2#vpSUPERVISOR-JUDGE:    fdr2#charles
```

Inferred by RDFS reasoner

```
    type:            DO#Judge
    type:            DO#Supervisor
    type:            DO#Serson
```

Properties generated by FDR2

```
fdr2#milk
    fdr2#vpPRODUCT-JUDGE:       fdr2#pete
    fdr2#vpPRODUCT-SUPERVISOR: fdr2#heinz
```

Inferred by RDFS reasoner

```
    type:                DO#Product
    DO#has_judge:        DO#pete
    DO#has_supervisor:   DO#heinz
```

Figure 6.4: An example of a run-time RDF/S model with entailments automatically inferred by an RDFS reasoner.

shows a fragment of a run-time model that contains entailments deducted from the above-mentioned models by the general RDFS reasoner. This run-time model can now be queried with terms defined in the ontology to access the content of the original relational data.

FDR2#Kit To test the proposed technique and to provide a basic tool support to the user we have developed ***FDR2#Kit*** - a Web-based toolkit consisting of three utilities:

- ***FDR2#Generator*** takes a tab-delimited text file with tabular data as its input and automatically generates RDF/S documents for the relational schema and relational

data.

- ***FDR2#Mapper*** assists the user in linking the relational schema to the DO.
- ***FDR2#Tester*** allows to run simple queries over the resulting combination of schema, data, RDMap and the DO.

In the ***FDR2#Mapper*** tool we have implemented a number of heuristics to automatically propose RDMap links between classes and properties from a domain ontology and relational schema. These heuristics propose the user to link classes or properties:

1. when parts of concept names that follow a namespace (after the '#' character in our examples) are lexically equivalent (e.g., `fdr2#Supervisor` and `DO#Judge`);
2. if the RDMap already contains linked properties, then we propose to link classes from domains and ranges of these properties. For example, if the RDMap links properties `fdr2#vpPRODUCT-SUPERVISOR` and `DO#supervisedBy`, then we suggest the user to link classes in domains (e.g., `fdr2#Product` and `DO#Product`) and ranges of these properties.
3. if the RDMap contains linked classes and these classes appear in domains and ranges of two properties, then we propose to link these properties. For example, if classes `fdr2#Product` and `DO#Product` are linked, and `fdr2#Supervisor` and `DO#Supervisor`, then we propose to link the properties `fdr2#vpPRODUCT--SUPERVISOR` and `DO#supervisedBy` because their domains and ranges are already linked.

These three heuristics are rather simple. Nevertheless, for the example employed in this section these heuristics are sufficient to fully automatically create an appropriate RDMap.

6.4 Cross-table References

Primary and foreign keys are essential parts of relational databases. They uniquely identify entities and enable referencing across tables. Although our core approach does not assign any special meaning to record elements containing keys, in this subsection we will show how the approach can be aware of foreign and primary keys.

Let us assume that there are two tables one of which refers to the other as depicted below:

Products

Product	Judge ID (foreign key)	Supervisor ID (foreign key)
milk	person1	person2

Persons

Person ID (primary key)	Name	Address
person1	Heinz	xyz1
person2	Pete	xyz2

The proposed approach does not support direct links between relational schemas but indirect links can be created in a number of ways. One way is to guarantee that the same identifier is created for lexically equal values. Having executed steps 1 and 2 of the FDR2 approach for the two tables shown above, the following classes and instances will be created (in addition to ones already described):

- table "Products": Classes `fdr2#JudgeID`, `fdr2#SupervisorID` with corresponding instances `fdr2#person1` and `fdr2#person2`;
- table "Persons": Class `fdr2#PersonID` with instances `fdr2#person1` and `fdr2#person2`.

We can see that although every table is processed independently, the way we created resources' URIs ensures that lexically equal record elements will always refer to the same resource.

Alternatively, we can link different tables via a domain ontology by assigning the same parent `DO#Person` to `fdr2#JudgeID`, `fdr2#SupervisorID` and `fdr2#PersonID` and preserving the original lexical value of their instances in a separate property `fdr2#cellValue`. This makes it possible to determine that some instances of `DO#Person` has equal `fdr2#cellValues` properties and therefore represent the same primary key. However, such an approach would require inference capabilities not available in standard RDFS reasoners.

This demonstrates that the proposed FDR2-approach supports referencing across tables if we can assume that lexically equal record elements represent the same resources. If such an assumption does not hold then indirect referencing can be implemented by introducing the `fdr2:cellValue` property and linking classes representing primary and foreign keys to the same parent class defined in the DO. However, the latter approach will require an ad hoc reasoning support.

6.5 Discussion

FDR2 relies solely on the RDFS semantics of the `rdfs:subClassOf` and `rdfs:subPropertyOf` relationships. We do not introduce any additional relationships that require special handling. This enables us to easily combine FDR2 with existing RDFS inference engines. FDR2 completely preserves the structure of the relational data. We only change representation of the data and add means to facilitate linking between a relational schema and an ontology.

The equivalence relationship in FDR2 In FDR2, we employ the `rdfs:sub[Class|Property]Of` properties to simulate the equivalence relationship not supported by RDFS. The W3C specification [14] defines the `rdfs:sub[Class|Property]Of` properties as transitive. Keeping in mind that the equivalence relationship is transitive, symmetric and reflexive we can see that `rdfs:sub[Class|Property]Of` lacks symmetry and reflexivity to act as the equivalence relationship. Symmetry can be achieved by stating for every A `rdfs:sub[Class|Property]Of` B that B `rdfs:sub[Class|Property]Of` A.

However, in practice a link between a domain concept and an element from a relation schema does not have to be symmetric. The main reason for this is that the FDR2 relational schema and the ontology play two distinct roles. The ontology defines a widely accepted view to which an ontology-enabled application commits. Whereas, the relational schema is only a medium that makes relational data available to the ontology-enabled framework (or software) that is aware of neither how the relational data is represented originally nor how it is made accessible to the framework.

This division of roles means that the ontology-enabled application is not supposed to directly exploit classes and properties defined in the relational schema. This implies that for our task transitivity of `rdfs:sub[Class|Property]Of` is sufficient. Therefore, we do not ensure symmetry by stating B `rdfs:sub[Class|Property]Of` A for every A `rdfs:sub[Class|Property]Of` B. This reduces the amount of reasoning needed to link the relational schema and ontology. Even more, if RDFS would provide a way to assign aliases (multiple identifiers) to the same resource then the same effect could be achieved with no reasoning at all.

Over-generation of virtual properties By means of virtual properties the relational schema explicitly defines sub-relationships between elements of a complex relation expressed in a

record. Generally speaking, it is possible to explicate not only binary sub-relationships but also more complicated ones. This could, however, introduce redundant relationships unlikely to be linked to domain properties.

Over-generated virtual properties may pose a performance problem. For example, a 10-element record will result in a relational schema with 90 virtual properties and a large number of them may be redundant. Such a relational schema itself does not require significant computational and space resource. However, the corresponding RDF-serialization of the actual data will be polluted with irrelevant data.

This problem can be addressed by introducing a separate step between automatic generation of the relational schema and RDF data serialization. At this stage we can remove redundant virtual relations from the relational schema. The user, for example, can interfere between steps 1 and 2 to remove irrelevant virtual relations. Another possible solution is to swap steps 2 and 3 and to exploit the created RDMap to (semi)-automatically remove virtual relations not linked to an ontology. The modified relational schema will determine the final structure of the RDF data serialization preventing from polluting it with irrelevant data.

Potential applications for the FDR2 relational schema The relational schema can facilitate analysis of the relational data on an abstract, intensional level. A possible practical application of this is that an RDFS-based information system can keep track of known relational schemas and corresponding RDMaps. This enables automation of handling complex input data. Since the relational schema is constructed automatically, once created the RDMap can be reused by many users who even do not know anything about the details of the linking procedure and they are still able to take advantage of RDFS inference.

Improvements and future work If a substantial amount of information about some domain is captured in relational-like data sources (e.g., spreadsheets), then a domain ontology can be constructed (or extended) incrementally by linking automatically generated relational schemata to the domain ontology. For example, starting with the ontology with only initial root concept(s) we begin every iteration by creating an RDMap and merging this map with the ontology. Such an approach can be generalized beyond relational data sources if there is a way to explicate relevant concepts (classes and properties) from an information resource.

The proposed approach is fully reversible: it allows to completely recreate relational representation of the original data. However, changes made within the RDF/S model cannot

be traced back to the relational representation. For example, if new instances of `DO#product` are created by the system then they will be lost when we try to revert to the original relational data model. At present it is not clear if full reversibility is important in supporting the relational data model but if we consider relational databases then it becomes obvious that full reversibility is a highly desirable feature necessary to provide a seamless link between relational databases and RDF/S-based systems.

6.6 Related Work

As we have already mentioned, a substantial amount of research effort has already been dedicated to the problem of mapping the relational data model to other conceptual models such as Object-Oriented, Entity-Relationship, etc. In the field of relational databases [71] and [68] provide overviews of theoretical approaches and practical systems (ARCUS [72], Penguine [73]) that assist reverse engineering of existing databases and enable interoperability with other models. Most of the existing solutions have a number of common features that we will discuss now.

Due to the limited expressiveness of the relational model (and any model in general), loss of semantics is inevitable during modeling of a target domain. Therefore, user intervention is required to establish proper semantic relationships between an existing relational model and other modeling frameworks. In FDR2 we require the user to do the actual linking of the relational schema to an ontology. Nevertheless, the relational schema itself is constructed automatically, and thus can be used independently for such tasks as schema discovery or retrieval.

In general, the above-mentioned existing techniques involve two main steps:

1. off-line user-guided schema mapping (often assisted with heuristics-based assistant-tools);

2. run-time data mapping performed by a dedicated software component.

The need for the run-time data-mapping component can be explained by the transformational nature of the approaches. The original structure of the relational schema and data has to be changed into the target representation. The mapping itself is defined as a set of rules for such a transformation. The transformation can cause arbitrary changes to the original schema/data, therefore the result has to be validated to ensure its completeness and consistency with the original data.

FDR2 is transformation-free in the sense that it preserves the original structure of the relational data. We do change its representation by encoding it in RDF, but we do not alter the structure itself. This, we believe, is a significant benefit of the FDR2 approach because it eliminates the need for a run-time transformation component and, more importantly, it automatically ensures that all relational data is accessible (completeness) and can be used in a way consistent with the original data model.

The mapping of a relational and a target schema is based on two major components: the definition of mapping rules and a run-time data-mapping component. In many cases, the mapping rules are stored in a relational database in an ad hoc format, thus they are difficult to share between users or software applications.

The run-time mapping component is required to perform a dynamic transformation of data according to the specified mapping rules. We can see that the specification of the mapping and the transformation engine are not independent from each other and, therefore they form quite a specific and inflexible solution. In our approach we rely solely on the well-defined semantics of RDFS. This allows us to produce a number of declarative definitions usable within any RDFS-enabled system.

The problem of mapping different vocabularies, schemas, ontologies have also received attention in the Semantic Web community. In that case, mapping has to deal with models that are semantically richer than the relational one. The proposed techniques also rely on the user to specify the actual mappings and provide the user with a minimal assistance.

In [74] the authors describe a naive approach for mapping RDBMS schema onto RDF (although we would rather call it RDBMS *data* mapping onto RDF). FDR2 takes it to the next level where it links the relational data and RDFS ontologies. An RDF serialization of the actual data is rather straightforward to realize and can be done in different ways according to application-specific restrictions.

FDR2 and existing approaches also differ in how they define mapping rules and how they perform the actual mapping at run-time. In [75] the author proposes a transformation-based mapping technique as an extension to the RDFS language. The authors of the D2R MAP XML-based language apply a similar technique to describe mappings between relational database schemata and OWL ontologies [70]. In practice, it means that the user is required to use a dedicated software component (e.g., a transformation engine or D2R processor) that supports the mapping languages introduced. The maturity of such software solutions is often unsatisfactory. In our approach we rely solely on the well-defined and

widely accepted semantics of RDFS languages. There exist a growing number of RDFS-enabled middleware components (Sesame [65], RDFSuite [76]) and APIs (Jena [57]) that can be directly utilized within the proposed framework.

We have to note that in the present work we do not address many issues relevant to mapping complex relational databases to RDFS domain ontologies. Our approach is designed to link tabular data (typically tables holding scientific observations) to RDF/S enabled software systems or frameworks such as Onto⇔SOA.

6.7 Conclusions

In this chapter we have described the FDR2 approach for linking relational and RDF data models. With this technique we answer the research question: *"How can we integrate tabular data with an ontology-ready data-model such as RDF/S?"*. FDR2 relies on three RDF/S components: an automatically generated RDFS schema of tabular data, an RDF serialization of the data itself, and a manually created map.

A relational schema is created automatically to explicate the structure and internal relationships (*virtual properties*) between elements of a relational collection of data. *Virtual properties* and generated RDFS classes allow the user to identify the `rdfs:subClassOf` `rdfs:subPropertyOf` relationships between concepts from the relational schema and a domain ontology. The actual relational data are automatically expressed in RDF according to the generated relational schema. Run-time integration is achieved by applying an RDFS reasoner to merge the above-mentioned components into a single RDFS model and to deduct necessary entailments. A resulting run-time model allows to access relational data with queries expressed according to the domain ontology.

The proposed approach is purely RDF/S-based and does not require any additional software components except an RDFS reasoner. FDR2 can be extended to fit particular needs as it was demonstrated by describing how primary keys can be exploited to support references across tables and how potentially redundant *virtual relations* can be eliminated. Moreover, the described technique is general enough to be applicable to data sources different from relational ones. By explicating the schema of the original data, serializing the data according to that schema and linking the schema to a target ontology we can semantically enrich the data and improve its accessibility to ontology-enabled software.

Instances corresponding to cell values

```
fdr2#cream
    type:      fdr2#Product
    fdr2#vpPRODUCT-JUDGE:      fdr2#charles
    fdr2#vpPRODUCT-SUPERVISOR: fdr2#pete

fdr2#pete
    type:      fdr2#Judge, Supervisor
    fdr2#vpJUDGE-PRODUCT:       fdr2#milk
    fdr2#vpJUDGE-SUPERVISOR:    fdr2#heinz
    fdr2#vpSUPERVISOR-PRODUCT:  fdr2#cream
    fdr2#vpSUPERVISOR-JUDGE:    fdr2#charles

fdr2#heinz
    type:      fdr2#Supervisor
    fdr2#vpSUPERVISOR-PRODUCT:  fdr2#milk
    fdr2#vpSUPERVISOR-JUDGE:    fdr2#pete
```

Instantiation of the relation

```
fdr2#RelationInstantiation
    definedOver:      fdr2#RelationInstantiationSeq

fdr2#RelationInstantiationSeq
    type:      rdf#Seq
    _1,2:      fdr2#RowSeq1, RowSeq2

fdr2#RowSeq1
    type:      rdf#Seq
    type:      fdr2#Row
    _1,2,3:    fdr2#milk, pete, heinz

fdr2#RowSeq2
    type:      rdf#Seq
    type:      fdr2#Row
    _1,2,3:    fdr2#cream, charles, pete
```

Figure 6.5: An example of an RDF serialization of relational data (automatically generated by FDR2 according to the schema on Figure 6.2).

Chapter 7

Effect of Ontologies on Software Quality and Development Effort

The Semantic Web is envisioned to significantly improve Web applications. Ontologies play a central role in realizing this vision and, therefore, are expected to have a profound effect on the *quality* of Web applications. The potential of ontologies, however, is not limited to the Semantic Web. In this chapter we estimate the effect of ontologies on a number of quality characteristics of software. Since, the expected qualitative gains can be attributed to increased development effort rather than to ontologies, we also estimate the effect of ontologies on software development *effort*. In this chapter we aim to answer the research question: *"How do ontologies affect characteristics of software applications such as software quality and development effort?"*. More specifically, we employ the Quint2 and WEBMO models to estimate how ontologies affect software quality and the corresponding development effort. The analysis and results reported in this chapter are largely derived from our experience in developing ontology-enabled software. We believe that this chapter provides an indication of the overall positive effect of ontologies on Software Engineering practice.

◇

Maksym Korotkiy: On the Effect of Ontologies on Web Application Development Effort. In proceedings of the Knowledge Engineering and Software Engineering workshop. Germany, 2005.
Maksym Korotkiy: On the Effect of Ontologies on Quality of Web Applications. In proceedings of the Workshop on Building and Applying Ontologies for the Semantic Web. Portugal, 2005.

◇

The vision of the Semantic Web [16] is to significantly improve the experience of users of Web applications. To achieve this, the Semantic Web (SW) has to provide an environment that will advance the *qualitative* characteristics of Web applications beyond the state of the art. Although ontologies play a key role in realizing the SW vision, their potential by no

means is limited to that area only. We have already demonstrated that with Onto⇔SOA by employing ontologies to improve usability of services. Usability is only one of the many quality characteristics of software and in this chapter we will investigate how exactly ontologies can affect a number of other quality characteristics of Web applications.

Estimating qualitative changes alone is not sufficient to judge the overall effect of ontologies on Software Engineering practice. The expected qualitative gains can be attributed to an increased development effort rather than to ontologies themselves. In that case it will remain uncertain whether ontologies contribute to more efficient transformation of effort into quality. Therefore, we also have to investigate how ontologies can affect Web application development effort. By estimating these two effects of ontologies we aim to obtain an indication of the overall benefits of applying ontologies in Software Engineering.

Ontologies are yet to be widely adopted by software developers. This significantly complicates empirical validation of the analysis performed in this chapter. Nevertheless, we believe that this analysis gives valuable insights into practical implications of using ontologies in software development. Our methodology consists of analyzing the effect of ontologies using well-known models for estimating quality (Quint2) and development effort (WEBMO) of Web applications.

Quint2 [77] defines a model that covers the user and developer perspectives on the quality of a Web application. The user perspective addresses external qualities of a software product. The developer perspective deals with internal qualities of the product during its development and maintenance. We employ Quint2 to perform a structured analysis of the impact of ontologies on the *functionality*, *maintainability* and *usability* quality dimensions.

WEBMO [1] is a method for estimating Web development costs. In our analysis we will consider an average sized (as defined in [1]) Web application (an information portal) to be developed by an average team of developers. We use a conventional software development process and the resulting product as a reference case and compare it against an ontology-aware counterpart. The ontology-aware case will be examined from two perspectives:

1. *the transition phase* that reflects short-term effects and the present state of the ontology engineering field;

2. *the maturity phase* that represents long-term effects and an optimistic outlook at the development of ontology engineering.

This chapter is organized as follows. Section 7.1 describes a variety of forms ontologies take, roles they play and expected general benefits. Section 7.2 explains the Quint2 quality

model and how we employ it to analyze the ontology impact on three quality dimensions: functionality, maintainability and usability. After that we look into development effort by describing the WEBMO estimation model in Section 7.3 and applying it to predict the effect of ontologies on the size of software and on development effort. In Section 7.4 we summarize our findings and reflect on the performed analysis.

7.1 Ontologies

We have already introduced the notion of ontologies in previous chapters. In this section we highlight a few general characteristics of ontologies that we believe are most relevant for the analysis of their effects on Software Engineering practice.

To be effective in practice an ontology must be expressed in a language that provides a unified representation (syntax, structure and semantics) mechanism. In this chapter we assume that an ontology is expressed in an ontology language that enables large-scale sharing and (re-)use of ontologies by both humans and machines.

Ontologies can possess a variety of forms starting from a simple list of terms (a controlled vocabulary), to a structured representation of domain concepts and relations between them or even to a knowledge-rich axiomatized representation of complex domains (see [12] for a detailed overview of different interpretations of the term "ontology"). In our analysis we employ the term "ontology" to refer to all these forms and will distinguish between them when appropriate.

One of the key roles an ontology can play is to be a source of commitment [11] – an ontology provides a unified conceptual basis that helps parties to unambiguously understand the communicated content (information or knowledge). There are, therefore, a number of practical benefits ontologies can offer to Software Engineering:

- Ontologies facilitate communication between agents of a different nature, their interoperability and reusability of ontology-enabled resources [78]. The structured and unified nature of an ontology language assists in reuse of the ontology itself. An ontology supplies concepts that can be used to describe or annotate design artifacts (Web pages, technical documentation, source code, etc) making these artifacts more accessible (reusable, interoperable).

- Ontologies are able to provide suitable abstractions (well defined problem-solving methods and domain theories) for software developers [79].

- As demonstrated with Onto⇔SOA, a domain conceptualization captured in ontologies contributes to the alignment between a domain and artifacts (e.g., services) designed for that domain.

In the rest of this chapter we will analyze how these general benefits of ontologies can affect software quality and development effort – the key elements of software engineering practice.

7.2 Effect on Quality of Web Applications

Quality of software products has been (and still is) studied extensively. The ISO 9126 standard provides a set of characteristics that describe various quality dimensions of a software product and of software development processes. The standard seeks to cover an exhaustive set of quality characteristics applicable to the whole range of software products.

The original ISO standard has been extended into the Quint2 model [77] that provides additional quality characteristics and, more importantly for our task, associates computable indicators with every sub-characteristic (Figure 7.1). These indicators intend to estimate the quality of software products with a certain accuracy and degree of confidence.

The indicators are attached to quality sub-characteristics. Quint2 does not specify how the indicators should be combined to obtain an integrated estimate of an overall quality characteristic. Integration of different indicators may be problematic or even unreasonable due to the variety of their scales and the different nature of the measured attributes. Moreover, the importance of a quality aspect[1] varies not only across application domains but also across different stages of the application life-cycle.

In our analysis we do not focus on quantitative effects of ontologies on Web applications. Instead, we investigate the sensitivity of a quality dimension to an ontology. Under the sensitivity we understand *a likelihood of a significant contribution of an ontology to a quality aspect*. This enables us to determine which dimensions are the best candidates to be improved by ontologies.

We derive the sensitivity of a quality characteristic from the sensitivity of its indicators or sub-characteristics. The sensitivity of an indicator is binary: it is either *likely* or *unlikely* to be significantly improved by ontologies. To determine the sensitivity of on an indicator we analyze its definition to find out if an ontology can contribute to underlying components.

[1] We employ "quality aspect" as a collective term for quality indicators and (sub)characteristics

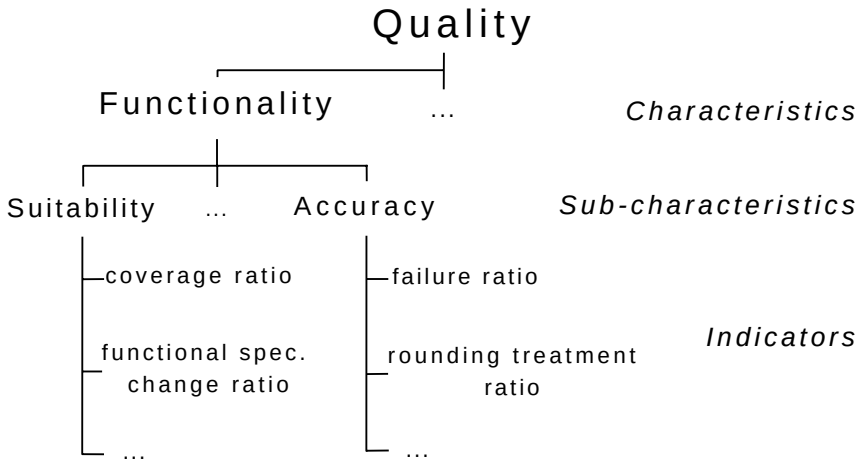

Figure 7.1: A fragment of the Quint2 quality model. The notion of quality is decomposed into a number of coarsely-grained characteristics that, in turn, consist of quality sub-characteristics. To each of these sub-characteristics Quint2 attaches computable indicators to provide a quantitative estimation of a sub-characteristic.

An integrated sensitivity of a number of given quality aspects is scored on an ordinal scale from 0 (worst) to 10 (best) according to percentage of positively effected aspects among the given ones. The numerical values of integrated sensitivity have *no absolute value*, however, we can employ them for ranking purpose.

To explore the benefits associated with application of ontologies, we do not consider them as a replacement for conventional software development methods but rather as a complementary technique. In our analysis we assume that an ontology can be employed as a container of application domain knowledge.

In this study we also assume that there already exist proper ontologies and associated methodologies allowing to apply these ontologies to a problem at hand. We assume that all required ontologies are available, of proper quality and ready to be applied.

In the coming subsections we will analyze the effect of ontologies on the *functionality*, *usability* and *maintainability* quality dimensions.

7.2.1 Functionality

The *functionality* quality characteristic addresses the actual presence of the desired functionality in the software product, the accuracy of the implementation, interoperability with specified systems, etc. In other words, this quality characteristic reflects the completeness and correctness of the implementation of the predefined functional requirements. Table 7.1 shows most of the functionality sub-characteristics such as *suitability*, *accuracy* and *interoperability*.

The *suitability* sub-characteristic considers the desired functionality actually present in a software product. We believe that this quality aspect is determined to a significant degree by the effectiveness of the requirements engineering phase. In this phase, a software engineer elicits the target functionality from domain users and experts. Ontologies are able to significantly improve the precision of the communication between these two parties and to facilitate unambiguous understanding of the application domain and problems to be solved. This will have a positive effect on the `coverage ratio`[2] indicator by reducing the number of missed or misunderstood requirements.

Definition	*Indicator*	*Sens.*
Suitability		
The percentage of desired functionality that is actually present in the software product.	coverage ratio	+
The ratio of functions that has been changed (change includes addition, modification, and deletion).	functional specification change ratio	+
The number of improvement requests for software functions from users per month after delivery.	improvement request ratio	+
Accuracy		
The ratio of incorrect processed transactions to the total of presented transactions.	failure ratio	+
The ratio of functions with the required rounding treatment to the total number of implemented functions.	rounding treatment ratio	-
Interoperability		

[2]Here and further on the actual definition of a quality indicator can be found in a table containing the corresponding quality sub-characteristic.

The effort needed to realize interoperability per unity of size of interoperability.	effort per interaction	+
The ratio of data formats matched to those of the other system in the interoperation.	matched data format ratio	+
Compliance		
The ratio of standardised data formats to the data formats to be standardised.	standardised data format ratio	+
The ratio of standardised interfaces to the interfaces to be standardised.	standardised interface ratio	+
Traceability		
The amount of time that is lost while processing due to operation control activities, manually or automatically.	operation control effort	+
The amount of effort needed to perform the operation control.	ease of operation control	+

Table 7.1: A fragment of the *functionality* dimension of the Quint2 quality model. Here and thereafter all definitions are given according to the Quint2 quality model.

Ontologies facilitate communication between all parties involved in requirements acquisition contributing to better understanding of the problems to be solved, and more efficient specification the desired functionality. Requirements engineering benefits from knowledge acquisition techniques [79, 6] that can be combined with ontologies to further improve their efficiency. All this prevents frequent modifications of requirements improving the `function specification change ratio` and `improvement request ratio` indicators.

The *accuracy* sub-characteristic estimates the correctness of the functionality, documentation, user manuals, etc provided with a software product. We can apply ontologies to structure functional requirements and facilitate acceptance tests. For example, we may employ ontologies to set restrictions on input/output data, and thus, assist in verification of the implemented functionality. The effectiveness of such assistance significantly depends on the ability of an ontology to capture the relevant relationships between domain concepts

(Onto⇔SOA enhances this ability by enabling domain-specific reasoning services). Hence, we expect such use of ontologies to improve the `failure ratio` indicator.

The `manual conformance ratio` indicator can be improved by increasing the number of functions described in a user manual. An ontology-assisted specification of requirements can improve the precision and understandability of the description of implemented functions. This simplifies the construction of a manual, makes it more efficient, thus allowing to cover a greater number of functions.

`Rounding treatment ratio` is not affected by ontologies. The indicator is rather influenced by technical aspects (target platform, performance and precision requirements, etc).

The *interoperability* sub-characteristic covers the ability of a software product to interact with specified systems. `Matched data format` and `interface ratios` are prime candidates to be improved by providing ontology-based descriptions of data formats and interfaces of systems in interoperation. Such descriptions can be shared and unambiguously interpreted with the help of inference services connected to an ontology (for example, as we propose in Onto⇔SOA). Ontologies help to promote standards by providing accessible reference specifications. All this can significantly improve the `effort per interaction` indicator implying that all indicators of the interoperability sub-characteristic are likely to be improved by ontologies.

The *compliance* sub-characteristic indicates how well a software product adheres to application-specific standards and regulations. If these regulations are accompanied with ontological specifications then the required compliance can be assisted using readily available inference capabilities provided by the ontology. We expect all compliance-related indicators to benefit from the availability of ontologies. However, this assumes that ontology engineers have at their disposal means to create ontologies for a wide range of application domains. We believe that a hybrid approach to ontology engineering such as Onto⇔SOA will be essential to achieve the expected impact.

Traceability indicates the amount of effort needed to verify the correctness of data processing. A readily available and structured ontological conceptualization of an application domain can facilitate the users in accurately expressing functional requirements in a way that allows their automatic verification. Moreover, an application domain ontology assists in the interpretation of the results of an operation, enabling domain-aligned and knowledge-driven verification of operations. We expect this to improve the overall efficiency of operation control process and, therefore to improve both traceability indicators.

7.2.2 Usability

Quint2 defines *usability* (Table 7.2) through a set of sub-characteristics that describe the effort needed to use a (Web) application as well as the associated user experience. Usability plays a central role in Onto⇔SOA. One of the main assumptions underlying our work is that an ontology contains concepts familiar to the domain users (experts, software developers, etc). In Onto⇔SOA we rely on ontologies to improve the usability of services. However, we believe that ontologies can be as well applied to other software engineering artifacts such as GUI, frameworks, components, etc. Ontological concepts can improve communication between the user and application developers, as well as between the user and a Web application. Quint2 decomposes usability into sub-characteristics such as understandability and learnability (Table 7.2).

Understandability concerns with recognisability of concepts and their intended applicability. `Concept` and `usage clearness` significantly benefit from the availability of the domain conceptualization captured in an ontology. The ontology can transfer these concepts through all stages of software development. This ensures that terminology elicited from the users during requirements engineering is preserved through all design and implementation steps and is used to interface with the users.

Learnability reflects the users' effort for learning a software product. The improved *understandability* and *clearness* of a software product facilitate users in learning how to operate the system and what kind of input and output it produces or requires. This improves the `effort required to learn one operation` indicator.

On the other hand, we estimate that the `availability of manual` indicator is not directly affected by an ontology but rather by other factors such as available human resources, complexity of a software product, etc. An ontology may affect those factors (for example, by increasing overall efficiency of development process and freeing up human resource), thus indirectly affecting the indicator. However, we do not consider such indirect influence significant enough.

Operability characterizes the users' effort for operation and operation control. An application domain theory expressed as an ontology facilitates understanding problems to be solved. The way problem solving methods are applied to the target domain go beyond what can be expressed in either a domain theory or an ontology of the problem solving method.

Neither we estimate that it is feasible to expect that domain ontologies can improve effectiveness of human-computer interaction (HCI) patterns. The main reason for this is that these patterns depend on many external (to the application domain) factors such as

types of users, applied problem-solving method, hardware and software limitations, etc. Since a domain ontology is unlikely to considerably affect these factors, we do not expect that an ontology can have a significant impact on the way interaction is organized between the user and software system. The multifaceted nature of HCI is also reflected in the fact that many of the operability indicators can be determined only experimentally via a subjective judgment by the user or an expert.

Definition	*Indicator*	*Sens.*
Understandability		
The proportion of functions that can be explained by using clear, familiar models to illustrate concepts.	concept clearness	+
The ratio of functions explained or by using clear models or presented to the user through demonstration software or anyway described.	usage clearness	+
The proportion of functions presented to the users through demonstration software.	availability of demonstration software	-
Learnability		
The ratio of time required to learn one operation for a specific task and operation time.	effort required to learn one operation	+
The degree of availability of reference manuals, on-line user's manuals and self-tuition documents such as operation manuals, grammar reference materials, installation manual, etc., for software functions.	availability of manual	-
Operability		
The proportion of system message terms that are standardised.	consistency of terms in messages	+
The proportion of system messages from software or system in which causes and corresponding action are clearly identified by the user who received those messages.	message clearness	+
The ratio of operating commands having default values to the total number of operating commands.	default value availability ratio	+
The proportion of functions for which operating methods can be selected to correspond to the user's level of skill.	skill level adaptability	+

Helpfulness		
The ratio of the amount of expounding text (including error messages) available in the software product to the total amount of text which can be presented on screen.	ratio of expounding text	-
The ratio of the amount of expounding text (including error messages) available in the software product to the size of the software product.	normalised ratio of expounding text	-
User-friendliness		
User-friendliness, as judged by a team of experts on topics as: screen composition, vocabulary, application of colour and sound.	expert judgment on user-friendliness	-
The experts decide to what extent user-friendliness of the software product matches the sample product.	user-friendliness compared to sample	-

Table 7.2: A fragment of the *usability* dimension of the Quint2 quality model.

On the other hand we believe that operability can be improved by employing effective HCI patterns [80]. An ontology can be used to describe basic HCI patterns and to assist in mapping them to domain concepts. For example, consistent terminology and message structure can be enforced by such an ontology improving the `consistency of terms in messages` and `message clearness` indicators.

A rich enough application ontology can supply default values to operating commands improving the `default value availability ratio` indicator. Skill level adaptability requires a rather detailed user model to be specified by a domain ontology. If such an ontology is available then we can use it to describe operations depending on the user skill level. For example, if a normal user interacts with an application then some operation parameters can be hidden with default values assigned. But, in the expert mode these parameters will be exposed to power users.

The *attractivity* quality characteristic reflects the ability of a software product to satisfy latent user desires through means beyond actual demand. Due to a highly informal nature of this quality sub-characteristic and the fact that it is the only indicator based on user judgment

we do not expect an ontology to have a considerable effect on it.

Clarity characterizes the ability of software to make the user aware of the functions it can perform. An application domain ontology is able to provide clear and recognizable descriptions for supported functionality. Moreover, the ontology can assist in organizing related functions, so the user can be advised on available relevant functions, how they can be combined, etc. We expect this to improve the `function recognition ratio` and `function use ratio` indicators.

The *helpfulness* sub-characteristic is determined by the availability of instructions to the user on how to interact with the software. Both of the defined indicators depend not only on the availability of explanatory instructions and messages but rather on the way these instructions are presented. An ontology can improve availability of explanatory texts by providing natural language descriptions for domain concepts, but this could as well clutter the UI with information that is likely to be already known to the user and, therefore not helpful. Hence, we foresee that a domain ontology is unable to provide much improvement for any of the defined indicators.

User-friendliness is determined by the level of users' satisfaction. This notion is extremely subjective. All the defined indicators depend on subjective judgment of either users or experts. The definitions of the indicators do not allow us to estimate the effect of ontologies no this sub-characteristic.

7.2.3 Maintainability

The *maintainability* quality dimension is characterized by the effort needed to introduce specified modifications into software. The sub-characteristics of this dimension are shown on table 7.3.

The *analysability* characteristic expresses by the effort needed to diagnose causes of failures[3] or to identify parts to be modified. An ontology of an application is able to support an explicit specification of restrictions on domains and ranges of implemented functions as well as state transitions. This can assist in automatic detection of failures and of their context.

Moreover, since an ontology facilitates consistent use of terminology in all development stages, we are able to trace failures manifested at the UI level back to the implementation and design levels. We expect this to reduce the amount of unrecognized failures and to

[3] A failure is externally recognizable erroneous behavior caused by the execution of a potential error in the software.

facilitate correct identification of corresponding faults, thus improving both analysability indicators.

Changeability is defined by the effort needed to modify the product. Changeability extends analysability beyond failure analysis, though most of the arguments applicable to analysability are also valid for changeability. Domain ontologies can facilitate software analysis, determine dependencies and assist in validation of a modified module. Therefore, we expect that all this can significantly reduce the `modification effort` indicator. Due to the improved analysability and reduced modification effort, `defect correction effort` and `mean correction time` are also likely to be reduced.

The `Mean failure treatment time` indicator characterizes how rapidly an application restores its normal state. This indicator depends on a variety of factors not affected by an ontology. Severity and nature of a failure together with the implemented failure handling mechanism determine these factors. Therefore, we do not expect any improvement on this indicator.

The *stability* quality sub-characteristic addresses the risk of unexpected effects of modifications. The single indicator defined in Quint2 measures the ratio of new faults made while modifying a (Web) application. An ontology is able to facilitate specification of dependencies (e.g., by supplying a shared and well-understood vocabulary to express them) between application components and between the application and external systems. We expect that such a specification will reduce the number of unexpected side effects.

Definition	*Indicator*	*Sens.*
Analysability		
The ratio of number of failures where users correctly recognised the fault positions to the number of detected failures caused by the faults of the software as a consequence of the maintainer analysing the failures.	fault position recognition ratio	+
Mean time needed to analyze a failure, and to discover any faults arising from this failure, and separate the positions to be repaired by the maintainer who received the failure report.	mean failure analysis time	+
Changeability		
Average amount of effort needed to modify the software product, per unit volume of the modification.	modification effort per unit volume	+

Mean effort needed to repair a defect in the software product.	correction effort per defect	+
Mean time from the failure occurrence to the restoration for end users.	mean failure treatment time	-
Testability		
Effort needed to test one unit volume of the software product with a certain testing coverage degree.	test effort per unit volume	+
Number of test cases that have to be made to test a unit volume of the software product with a certain testing coverage degree.	number of test cases per unit volume	+
Mean user's work time to verify the fault correction.	mean user's work time to verify the fault correction	+
Reusability		
Ratio of reusable parts of the software product to the total number of parts of the software product.	ratio reusable parts	+
Ratio of reused parts of the software product to the total number of parts of the software product.	ratio reused parts	+

Table 7.3: A fragment of the *maintainability* dimension of the Quint2 quality model.

The *testability* characteristic is defined by the effort needed to validate the modified software. A domain ontology can assist in creating restrictions on input parameters. Also such an ontology can provide a framework for organizing application functions making both functional and conceptual interdependencies more explicit. This we believe can reduce the overall test effort.

Additionally, domain ontologies can facilitate the specification of dependencies and data flow between sub-modules of a tested unit. This assists in development of an efficient test strategy. We also expect that verification and testing of fault corrections can significantly benefit from an ontology-based organization of functional requirements (verification) and operational restrictions (testing). For example, an ontology can help to identify software components related to the key conceptual elements (e.g., concepts that are connected to a

large number of domain concepts) of an application domain. By focusing effort on these critical software elements we can improve effectiveness of the verification and test process.

The benefits of an ontology-based approach to software verification are not limited to the formal techniques. With Onto⇔SOA we complement the formal techniques through service-enabled ontologies. Such ontologies can provide a reference implementation of coarsely-grained domain behavior. The reference implementation facilitates integration testing and verification.

Managability is defined by the ability of software to (re-)establish its running status. `Control effort ratio` is defined as "the ratio of effort put in controlling the software product (including maintenance) in man-hours to the number of hours the product is available to the users". This definition does not allow to make a conclusion about the effect of ontologies on this sub-characteristic.

The *reusability* characteristic evaluates a potential for complete or partial reuse in another software. Ontologies encourage and facilitate reuse of software modules by improving their analysability, customizability and interoperability. We expect both of the reusability indicators to significantly benefit from ontologies.

7.2.4 Summary

In the previous sections we argued that ontologies can (or cannot) have a significant positive impact on various quality indicators. In this section we summarize this argumentation.

We expect a positive impact of an ontology on a quality indicator if:

- The indicator depends on precision and unambiguity of communication between domain users and software developers. In this case we expect a positive effect because an ontology specifies a domain conceptualization that can be employed by agents with different background knowledge.

- The indicator benefits from consistent use of concepts across stages of development process. We see this case as an extension to the previous one. The communication now takes place among multiple parties (domain users, requirement engineers, software developers, testers, etc), has an asynchronous nature and can be prolonged in time. For example, a domain ontology provides concepts that are used to determine and describe application requirements. The same concepts can be used as a basis for software design, then employed in the implementation and will reappear in UI and final documentation.

- The indicator benefits from an increased agreement and standardization. In this case the scale and complexity of communication are further increased extending to software agents (interoperating application and services).
- The indicator depends on other indicators that are likely to be improved by an ontology. A wide applicability of ontologies amplifies the effect of ontologies.

On the other hand, we expect that an indicator is likely to be unaffected by an ontology if:

- It is unreasonable to expect availability of an ontology that covers a domain of the indicator. In this case, the indicator can be determined only via subjective judgments of users or experts making it difficult to confidently predict the effect of an ontology on it.
- An ontology improves the overall efficiency of development process freeing up resource that can now be devoted to improve the quality indicator. We do not consider such an indirect effect of an ontology on the quality indicator to be significant.

7.2.5 Sensitivity of Quality Sub-characteristics

Having analysed the effect of ontologies on indicators, we integrate the estimations for each quality sub-characteristic. Table 7.4 depicts the integrated sesnsitivity of the sub-characteristics ranked in the descending order.

From this table we can see that we expect ontologies to have a significant positive impact on the majority of analyzed quality sub-characteristics. The maintainability and functionality dimensions show more potential for improvement than the usability dimension. We believe that the main reason for this is that the former two dimensions benefit not only from the availability of domain-aligned conceptualization but also from the structured representation of this conceptualization.

Although we estimate that ontologies are able to significantly contribute to quality of software products, in order to evaluate the overall effectiveness of ontologies in Software Engineering we have to determine how ontologies effect development effort. In the next section we introduce ontologies into the WEBMO model to estimate their impact on Web application development effort.

Quality Sub-characteristic	*Quality Dimension*	*Sensitivity score*

Analysability	Maintainability	10
Compliance	Functionality	10
Customizability	Usability	10
Interoperability	Functionality	10
Suitability	Functionality	10
Testability	Maintainability	10
Traceability	Functionality	10
Reusability	Maintainability	10
Changeability	Maintainability	9
Accuracy	Functionality	5
Learnability	Usability	4
Understandability	Usability	4
Operability	Usability	3
Helpfulness	Usability	0
User-friendliness	Usability	0

Table 7.4: Integrated sesnsitivity score of the analysed quality sub-characteristics.

7.3 Effect on Development Effort

We may expect that the quality improvements described in the previous section will come at the expense of increased development effort. In this section we investigate how the introduction of ontologies can affect Web application development effort. By combining this investigation with the results of the previous section we expect to obtain an indication of the overall viability of ontologies for software development.

Although we aim to perform a general analysis, to quantify the arguments we will consider an average sized Web application (an information portal) to be developed by an average team of developers. We use a conventional ontology-free development process and the resulting product as a reference case to be compared against its ontology-aware counterpart. The ontology-aware case will be examined from two perspectives: 1) *the transition phase* that reflects short-term effects and the present state of the ontology engineering field and 2)

the maturity phase that represents optimistic long-term effects and expected development of the ontology engineering field.

To estimate the impact of ontologies on the development effort we employ the WEBMO methodology [1]. According to WEBMO, development effort of a Web application depends on three main factors:

- *application domain* – WEBMO distinguishes five application domains: e-Commerce, financial/trading applications, business-to-business applications, Web-based portals and Web-based utilities.
- *size of the application* measured in Web Objects (WO). WO is an extension to classical Function Points that takes into account Web-specific components of the application such as multimedia files, scripts, etc.
- *cost drivers* (Table 7.5) are concerned with the context of software development process (the development team, its experience, schedule constraints, etc) and with its efficiency.

For our analysis only the last two factors are relevant: we assume that the introduction of an ontology does not change the domain of the application. WEBMO estimates development effort (in person-months) as follows:

$$E = AS^P C,$$

where A and P are power law constants that depend on the application domain; S is the estimated size (in thousands of source lines of code) of the considered Web application and C is the product of all cost drivers.

We use subscripts to distinguish between the reference case (0-case) and the case when we apply ontologies (onto-case). The introduction of an ontology does not change the application domain (i.e. $A_0 = A_{onto}$) allowing us to discard A. Furthermore, since in WEBMO P takes values from $\{1, 1.03, 1.05\}$ depending on the application domain, we do not expect it to have significant impact on the ratio $\frac{S_{onto}}{S_0}$ to be around 1, therefore we can discard P as well. Finally, by assigning the nominal value of 1 to all cost drivers in the 0-case (C_{onto}) we can compute a relative effect of ontologies on development effort as follows:

$$\frac{E_{onto}}{E_0} = \frac{S_{onto}}{S_0} C_{onto}.$$

This equation allows us to compute the relative effort as a product of the relative change of application size (estimated in Section 7.3.1) on the product of the cost drivers that take place in the onto-case (analyzed in Section 7.3.2).

7.3.1 Application Size

In this subsection we estimate the impact of an ontology on the size of a Web application. Given a set of functional requirements we will consider the two above-mentioned cases: the onto-case and the reference case. In the onto-case the Web application is developed with ontologies applied whenever possible and justifiable by the requirements.

WEBMO uses Web Objects (WO) to estimate the size of a Web application and employs the following WO predictors [81]: Internal Logical Files, Multimedia Files, Web Building Blocks, Scripts, Links, External Interface Files, External Inputs, External Outputs and External Queries. The predictors employ ranks (low, average and high) to take into account the varying complexity of Web objects.

We assume that all the external predictors (External Interface Files, External Inputs, External Outputs and External Queries) are beyond the control of an application developer. Therefore, these predictors will stay the same in the onto-case and the 0-case.

We assume that the Multimedia Files and Links predictors are not affected by the ontology because they represent the content of the application and the way it is presented to the user. Therefore, we conclude that these factors do not change across the cases.

The Web Building Blocks predictor represents reusable software components employed in the application. The building blocks are used to directly provide the requested functionality or are required by the application design. By applying an ontology in a Web application an additional building block is required to account for ontology-related middleware. In the onto-case this block can be seen as a replacement for the corresponding middleware used in the reference case, thus causing no relative impact on application size. If the ontology-related middleware has no counterpart in the reference case then the impact on the Web application size will be marginal: in WEBMO one building block can be accounted for at most 6 WO (with an average Web application consisting of around 300 WO [82]).

An ontology can be applied as an internal logical file if an application directly employs the ontological representation to express its data structures and/or applicable functions. Such a solution usually imposes significant performance costs related to the generality of the representation mechanism employed by ontologies. However, it provides developers

with a structurally and semantically unified representation. The main consequence of the generality is that ontology-based Internal Logical Files (ILF) are likely to have a higher rank than those in the reference case. Since the difference between the neighboring ranks is 5 WO, we can expect the $5 \times number\_of\_affected\_ILF$ increase of Web application size in the onto-case.

Ontologies are able to decrease the Scripts predictor that estimates the effort needed to connect the Internal Logical Files to the Web Building Blocks. Ontologies supply internal files with a unified representation mechanism. This broadens the applicability of a script to a greater number of internal files. In WEBMO the average rank for a script is 3 WO potentially resulting in a $3 \times number\_of\_removed\_scripts$ decrease in size.

Now we can apply the above estimations to a typical Web application described in [1]. The described application contains 356 WO with: 3 internal logical files (2 with the average rank and 1 with the high rank); 4 scripts (3 with the low rank and 1 with the average rank). Assuming that 2 internal logical files and 1 (averagely ranked) script are affected in the onto-case we will get 6 (ontology middleware) $+2 \times 5$ (increased complexity of ILF) -1×3 (eliminated script) $= 13$ WO or $100\% \times 13/356 = 3.65\%$ increase in application size. Such an insignificant change of the size of the considered Web application provides an indication that ontologies do not considerably affect the size of an average Web application.

7.3.2 Cost Drivers

In the reference case we assign the nominal value of 1 to the cost drivers. In the onto-case we will consider each cost driver and will either change the nominal rank of the driver to the next lower/higher or leave it unchanged (Table 7.5).

The product complexity cost driver (CPLX) represents requirements to the reliability of a product and reflects the complexity of its architecture. We believe that the presence of an ontology does not change reliability requirements. Although an ontology (and related middleware) can cause an insignificant increase of the complexity of the architecture, this does not justify assignment of the high rank to this cost driver.

Platform difficulty (PDIF) estimates the difficulty of the target application platform. Ontology-oriented API's and middleware are available but still far from maturity. Beside that, the application of ontologies requires additional (often significant) computational and memory resources. Taking all this into account we have assigned the high rank to the PDIF driver. Similar arguments are applicable to the facilities (FCIL) driver which rank is likely

	Ratings			
Cost Driver	Low	Nominal	High	Very High
Product Reliability and Complexity (CPLX)	Client/server, some math, file management, limited distribution.	Client/server, full distribution, databases, integration.	Client/server, wide distribution, math intensive.	Client/server, full distribution, collaborative.
Values	0.85	1.0	1.30	1.67
Platform Difficulty (PDIF)	Few platform changes, few resource problems.	Stable platform, must watch resource usage.	Platform often changes, lack of resources a problem.	Platform unstable, resources limited.
Values	0.87	1.0	1.21	1.41
Personnel Capabilities (PERS)	35th percentile, minor delays due to turnover.	55th percentile, few delays due to turnover.	75th percentile, rare delays due to turnover.	90th percentile, no delays due to turnover.
Values	1.35	1.0	0.75	0.58
Personnel Experience (PREX)	$\leq$ 6 months, some experience.	$\leq$ 1 year, average experience.	$\leq$ 3 years, above average experience.	$\leq$ 6 years, lots of experience.
Values	1.19	1.0	0.87	0.71
Facilities (FCIL)	Multisite, some collaboration, basic CASE.	One complex, teams, good tools.	Same building, teamwork, integrated tools.	Co-located, integrated collaborative tools, etc.
Values	1.13	1.0	0.85	0.68
Schedule Constraints (SCED)	Must shorten, 85% of nominal value.	Keep as is, nominal value.	Can relax some, 120% of nominal value.	Can extend, 140% of nominal value.
Values	1.15	1.0	1.05	1.10
Planned Reuse (RUSE)	Not used.	Unplanned reuse.	Planned reuse of component libraries.	Systematic reuse based on architecture.
Values	–	1.0	1.25	1.48
Teamwork (TEAM)	Little shared vision, marginally effective teamwork.	Some shared vision, functional teams.	Considerable shared vision, strong team cohesion.	Extensive shared vision, exceptional team cohesion.
Values	1.31	1.0	0.75	0.62
Process Efficiency (PEFF)	Project-based process, rely on leadership.	Streamlined process, rely on process.	Efficient process, best way to do job.	Effective process, people want to use it.
Values	1.20	1.0	0.85	0.65

Table 7.5: Cost drivers in WEBMO (adapted from [1]).

to be reduced due to the lack of advanced tool support for ontology-related tasks and the lack of well-established methodologies.

Personnel capabilities (PERS) are unaffected by ontologies allowing us to keep the nominal rank for this driver. Unlike general personnel capabilities, we estimate that average experience of the personnel (PREX) will be affected. The reason for this is that due to the lack of ontology-related experience an average development team of 3-5 members [1] will be unable to sustain the nominal level of experience ($\leq$1 year) if exposed to the ontology engineering field. On the other hand, an ontology can facilitate transition of domain knowledge into the development team, and in this way compensate the lack of ontology-related expertise. Therefore, we have decided to leave the PREX cost driver unchanged for the early phase of ontology acceptance. We estimate that when the maturity phase is achieved the PREX cost driver is even likely to be improved by ontologies facilitating transfer of domain knowledge into the development team.

Ontologies do not affect schedule constraints (SCED) preserving the nominal value for this driver. Since we believe that ontologies will encourage reuse of software, we estimate that the overall amount of the planned reuse (RUSE) can be assigned the high rank. Moreover, an ontology facilitates sharing of knowledge, thus positively affecting teamwork and improving team cohesion (the TEAM driver).

We believe that the process efficiency (PEFF) cost driver is also affected by an ontology. Process efficiency is inversely proportional to the efforts associated with the development process. Our estimation of a variety of effort-related quality indicators (Table 7.1) has shown that many of them could be improved by applying ontologies to the development process.

Table 7.6 provides the estimated values for the cost drivers in the considered cases: the reference case, and two onto-cases that correspond to different stages of maturity of the Ontology Engineering field. We estimate that in the onto-case transition phase the product of the WEBMO cost drivers will slightly increase (by around 10%), and in the maturity phase – significantly decrease (by around 50%). We explain the initial increase of the cost drivers with an exposure of the development team to a novel technique. We believe that, although our estimation is based on many assumptions favorable to ontologies, it provides quite an assuring indication of the potential of ontologies in improving the efficiency of development process on long term.

7.4 Discussion

We have applied well-established software engineering models to estimate possible impact ontologies may have on software. Our analysis indicates that ontologies have the potential to both improve quality and decrease development effort. We believe that ultimately this implies that *a given level of quality can be achieved with less effort if ontologies are applied during software development*.

Cost-effectiveness of reuse of knowledge captured in ontologies was experimentally investigated in [83]. The authors concluded that in spite of significant translation and adaptation effort ontology reuse proved to be cost-effective. This agrees with our conclusions.

7.4.1 The Effect on Quality

There is a number of ways ontologies can contribute to quality of software. Ontologies can facilitate communication between domain users and software developers, and to provide a starting point for application design. We can employ ontological terminology across all stages of the development process establishing relationships between domain concepts and software design and implementation artifacts.

	PDIF	*PREX*	*FCIL*	*RUSE*	*TEAM*	*PEFF*	*C*
reference case	1	1	1	1	1	1	**100%**
onto-case, transition	1.21	1	1.13	1.25	0.75	0.85	**109%**
onto-case, maturity	1	0.87	1	1.48	0.62	0.65	**52%**

Table 7.6: The estimated effect of an ontology on the WEBMO cost drivers for an average Web application. Three cases are considered: 1) reference – no ontology is applied; 2) onto-case, transition phase – ontologies are applied for the first time to development process; 3) onto-case, maturity phase – ontologies have been applied for some time. The cost drivers that do not change across the cases are skipped.

Ontologies encourage use of structured (systematic) methods in software development. If such a method is already applied, an ontology can increase its effectiveness by promoting consistent use of application-specific concepts. If no such method is used then an ontology

still can be applied to semi-structured and informal techniques to improve their precision and consistency.

While analyzing the effect of ontologies on software quality we noted that many effort- and time-related indicators can be improved by ontologies. This observation has been confirmed by the dedicated study of the impact of an ontology on development effort model.

By integrating the impact on quality indicators we can conclude that the *functionality* and *maintainability* dimensions are most likely to be improved by ontologies. We expect that software characterized by an extensive functionality, long life span is most likely to benefit from ontologies. For example, applications for the e-Science domain belongs to this category.

Usability is considered to be one of the most important quality characteristic of a Web application [84] primarily due to rather limited representation capabilities of modern Web browsers. We expect that an ontology is less likely to have a positive effect on this quality dimension.

On the other hand, Web applications usually require extensive maintenance/updating raising the importance of the *maintainability* dimension. In the nearest future Web browsers will support rich UI presentation mechanisms (XUL [85], XForms [86], etc) reducing the importance of the usability issues and increasing the importance of the *functionality* and *maintainability* dimensions. One of the ways to manage complexity of future Web applications is through their modularization with further integration and this is where ontologies have the biggest potential.

7.4.2 The Effect on Development Effort

The maturity phase of the onto-case reflects a number of effects of the transition to the ontology-enabled Web application development. Although, at present we observe a lack of the technological support (tools, middleware, etc) for ontology-related tasks, we believe that in coming years these aspects will reach the level of the state of the art Web technologies.

Ontologies can enhance personnel experience by transferring domain knowledge into a development team. They also contribute to team work by providing a team with a unified conceptual view on an application domain. This improves communication within the team as well as between the team and the external stakeholders. And finally, ontologies can increase the overall efficiency of the development process through improvements in knowledge transfer, communication and reusability.

7.5 Conclusions

In this chapter we have addressed the research question *How do ontologies affect characteristics of software development such as software quality and development effort?* To find an answer we have investigated the effect of ontologies on software quality and development effort correspondently estimated by the Quint2 and WEBMO models.

We have applied the Quint2 model to analyze effect of an ontology on indicators associated with the *functionality*, *maintainability* and *usability* quality dimensions of a Web application. The analysis has indicated that the *functionality* and *maintainability* quality dimensions are likely to be improved by an ontology, but the *usability* dimension is less likely to be improved. We explain the expected positive effect with increased efficiency of traditional software engineering methods. The improvements are primarily attributed to application domain ontologies that supply conceptualization usable across all development stages amplifying the cumulative effect.

WEBMO indicated that a marginal increase in overall development effort is expected during the ontology adoption phase. This increase can be explained by novelty and developing nature of ontology-related methodologies and techniques. In the long run, WEBMO predicts an ontology to cause a significant decrease of development effort. This becomes possible due to the ability of ontologies to facilitate transfer of application domain knowledge into a development team, to provide a unified conceptual view improving communication within the development team as well as to external world, to improve reusability of design artifacts, and, ultimately, to improve efficiency of development techniques.

We acknowledge that the obtained results, both quantitative and qualitative, are based on assumptions favorable to ontologies and largely derived from our experience, and therefore, are not readily generalizeable. Nevertheless, we believe that they provide a sensible indication of a significant potential of ontologies in the field of Software Engineering.

Chapter 8

Conclusions

In this work we have proposed to improve usability and reusability of knowledge by combining ontologies and Service-Oriented Architectures. We have applied the proposed framework, called Onto⇔SOA, to a number of use cases from the e-Science domain: retrieving documents (publications, experimental data, reports, etc) matching a given query, conversion and consistency checking of units of measurement. We have addressed these use cases in a novel way by using the guidelines proposed in Onto⇔SOA to design and implement ontology-based and document-oriented services.

Onto⇔SOA establishes an integrated framework that describes how ontologies and services can be designed in a way that enables their natural integration. We maintain a consistent, simple and pragmatic approach that can be deployed without significant investment of effort. The proposed guidelines (or design constraints) are a combination of well-established design practices from the fields of Software Engineering and Service-Oriented Architectures.

The proposed approach complements the state of the art research in the fields of Ontology Engineering, Semantic Web, Semantic Web Services, Service-Oriented Architectures and Software Engineering.

On the left-hand side of Figure 8.1 the well-known vision of Semantic Web Services is depicted as an ontology-based extension of Web Services that leads to Intelligent Web Services. In this vision only the route from Web Services to Intelligent Web Services is explored. The path from the Semantic Web to Intelligent Web Services is not considered in the state of the art research. On the right-hand side of the same figure we depict the "Onto⇔SOA Vision". In this vision we exploit both paths:

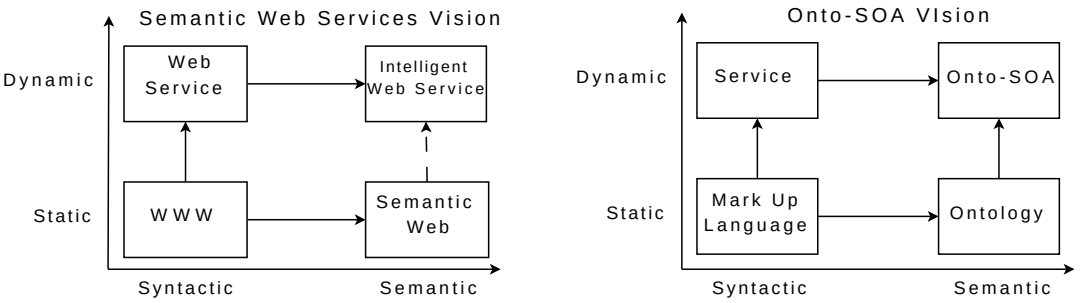

Figure 8.1: The "Full Potential Vision" of SWS and Onto⇔SOA.

- The Ontology-enabled Services path illustrates the extension of a service with an ontology and roughly corresponds to the route taken in Semantic Web Services. On this path we use ontologies to transfer domain conceptualization into services improving their domain (business) alignment characteristic.
- The Service-enabled Ontologies path represents the ability to attach a service to an ontology. On this path ontologies benefit from the services' ability to carry domain-specific procedural knowledge

In Onto⇔SOA these two paths converge enabling both services and ontologies to benefit from each other.

The main contributions reported in this dissertation are the answers to the research questions (RQ) raised in the introduction. In the rest of this chapter we summarize the answers to each of the questions and highlight the most notable discussion points.

RQ 1: How can typical e-Science tasks be facilitated with ontology-enabled software solutions? We have answered this question by applying Onto⇔SOA to design and implement solutions to a number of use cases from the e-Science domain:

- **1.1** The Document Retrieval use case is about finding documents that match a given query. We have covered this use case in Chapters 2 and 3.
- **1.2** The Unit Conversion use case is about finding a conversion factor between two units of measure. We have addressed this use case in Chapter 5.

- **1.3** The use case of Checking Consistency of Units (and dimensions) has been covered in Chapter 4.

We have employed the above-mentioned e-Science tasks primarily to evaluate the effectiveness and feasibility of the proposed Onto⇔SOA approach. The evaluation of the effectiveness of the provided solutions with respect to the e-Science field itself is outside the scope of this work.

In addition to the typical e-Science use cases, we have also addressed the more general problem of linking relational and ontology-friendly data models. We have formulated the corresponding research question:

RQ 1.4: How can we integrate tabular data with an ontology-ready data-model such as RDF? We have answered this question in Chapter 6 with the FDR2 approach that relies on three components: an automatically generated RDFS schema of tabular data, an RDF serialization of the data itself, and a manually created mapping of relational schema to a target ontology. The proposed approach is purely RDF/S-based and does not require any additional software components except an RDFS reasoner. Moreover, the technique underlying FDR2 is general enough to be applicable to data sources different from relational ones. By explicating the schema of the original data, serializing the data according to that schema and linking the schema to a target ontology we can semantically enrich the data and improve its accessibility by ontology-enabled software.

The main concern with the FDR2 approach is its significant space costs of serialization of virtual properties and corresponding data. Our pragmatic suggestion is to restrict the set of virtual properties only to those that are likely to be relevant to a case at hand.

Onto⇔SOA provided a uniform ontology- and service-oriented framework for the solutions to the target cases. As such, Onto⇔SOA itself is our answer to the *central research question* of this work:

RQ 2: How can ontologies and services (Service-Oriented Architectures) be integrated into a framework facilitating application of knowledge in e-Science (and beyond)? In Chapter 2 we have introduced Onto⇔SOA as an architectural framework for ontology-enabled services (Service-Oriented Architectures). This framework is based on a restricted service model that constrains internal properties of a service to induce *domain alignment* and *loose coupling* characteristics. These constraints allow, among others, to simplify the model of a service and to provide guidelines on the design of ontology-enabled services.

Onto⇔SOA addresses direct exchange of ontology-based messages between a *document-oriented* service and its consumer. It employs an ontology as a *service schema* (referred to as a *service ontology*). The main purpose of ontologies is to transfer domain conceptualization to services (Service-Oriented Architectures), thus enhancing their domain (business) alignment.

One of the distinguishing features of the proposed framework is that it relies on a *restricted* service model. One of the perceived drawbacks of such an approach is that this limits the range of application domains to which Onto⇔SOA can be applied. We consider this not "a drawback" (each approach has limitations to its applicability) but rather an explicit acknowledgement of the application domain boundaries of service-enabled ontologies. We can use these boundaries to either estimate how well the domain alignment and loose coupling characteristics can be met in a target application domain; or how this domain can be reshaped to achieve the target characteristics at the required level.

To facilitate the evaluation and application of Onto⇔SOA to the targeted use cases, we have further specialized it into *MoRe* – an operational framework and corresponding middleware based on RDF/S languages and REST Services. With *MoRe* we answer the following research (sub-)question:

RQ 2.1: How can we integrate RDFS ontologies with REST services? *MoRe*, introduced in Chapter 5, aims to provide a simple and pragmatic foundation for the development of ontology-based Web applications. We believe that it also bridges the gap between ontological domain knowledge and software development. On the one hand, it provides a pragmatic, application-driven view on domain ontologies. On the other hand, it facilitates software development by integrating domain-specific inference services into software solutions.

MoRe raises a number of discussion points at the operational level. One of these points is whether the directed labeled graph model underlying RDF and the ontological primitives defined in RDFS are capable of transferring domain conceptualization into services. Our experience from applying *MoRe* indicates that the underlying graph model is very flexible in representing domain concepts and relations. The interpretation of this graph model in terms of subjects, predicates and objects also provides a basis that is sufficiently neutral semantically and, hence, that aligns well with various application domains.

Another interesting discussion point is how well REST services actually integrate with Onto⇔SOA requirements. REST-full services offer four CRUD actions (PUT, GET, UPDATE and DELETE actions of HTTP) to manipulate Web resources. However, a service

in Onto⇔SOA is required to support one operation only (inference of new facts). In *MoRe* a consumer must use the HTTP POST action to submit an input document. This outlines the conceptual difference between the prescriptive REST-full actions and the descriptive, document-oriented interface to Onto⇔SOA services. Despite this difference, we still argue that REST services suit Onto⇔SOA better than standard (WSDL) services because REST services are able to provide sufficient support for document-oriented messaging in a simpler and less cumbersome way.

Collaboration between services enables construction of complex functionality from simple components, and therefore is an important component of any service-oriented approach. In Chapter 4 we have addressed this task in the context of Onto⇔SOA and answered the following research question:

RQ 2.2: How can ontology-enabled services work together? We have answered this question with an approach inspired by the ideas from Blackboard Systems. We have integrated Blackboard-style composition into the Onto⇔SOA approach. The devised composition mechanism utilizes an application-independent controller and a homogeneously structured ontology-based repository (a blackboard). The proposed approach requires neither an extensive service model nor an explicit workflow specification and enables composite functionality to emerge by bringing a number of ontology-enabled services together.

During the evaluation of the proposed service composition approach using the unit consistency checking task a number of potential performance bottlenecks have been spotted. The main reason for these bottlenecks is that composed Onto⇔SOA services may expose only very limited information, thus making it difficult for a composition controller to avoid unproductive invocations. A possible solution for this bottleneck is to allow the controller to learn from unproductive invocations and to dynamically adjust the invocation sequence accordingly. The same lack of information about a service (e.g., no preconditions or trigger procedures are exposed) also requires that the complete content of a blackboard has to be communicated to all composed services. A potential mitigation measure is to enable the controller to communicate only those concepts that are defined in a service ontology of a composed service.

Another key aspect of the proposed Blackboard-based composition mechanism is in deciding when composition process is complete. Since the proposed mechanism is data driven (rather than goal driven), the controller terminates composition when no new facts are inferred after a complete invocation sequence. Although we have introduced a number of requirements (facts can be only added but not removed or modified, services provide all

inferred facts at once, etc) that should prevent infinite composition, these requirements do not prevent it completely. So far, we have refrained from including a mechanism for abnormal termination into the proposed composition approach because we are not convinced in its viability in a general case. If required, such mechanism can be introduced on an implementation-specific basis.

By applying ontologies and services to the e-Science domain we have gained insights on how ontologies and services could benefit from each other. In addition to the main research questions we have addressed a number of more general research questions. The first of these questions is:

RQ 3: How can we attach a service to an ontology and what does this imply for ontologies? We have addressed this question in Chapter 3 where we have introduced Service-enabled Ontologies – an ontology-oriented perspective on Onto⇔SOA that shifts the focus to service ontologies. Service-enabled Ontologies re-interpret Onto⇔SOA as a mechanism that allows to attach an arbitrary service to an ontology, thus capturing *application semantics* of domain concepts.

These service attachments aim to facilitate practical application of ontologies, potentially at the costs of their overall reusability. The trade-off between declarative (general, reusable but difficult to utilize in practice), and procedural (application-specific but easy to exploit) knowledge still holds in Service-enabled Ontologies. Finding a suitable balance between these two ways of representing knowledge ultimately depends on the requirements of the specific application scenario. With Service-enabled Ontologies we provide a novel framework that contributes to flexibility in specifying a domain conceptualization.

The second of the two more general research questions is concerned with the overall impact of ontologies on software engineering practice. Or, more specifically:

4 How do ontologies affect characteristics of software development such as software quality and development effort? We have addressed this research question in Chapter 7 by investigating the effect of ontologies on existing models for estimating software quality (Quint2) and development effort (WEBMO). We estimated that ontologies can improve many quality dimensions by supplying domain conceptualization that can be employed across all development stages, thus amplifying the cumulative effect. We also estimated that in the long run, ontologies can cause a significant decrease of development effort. This becomes possible due to the ability of ontologies to facilitate transfer of application domain knowledge into a development team, to provide a unified conceptual view improving

communication within the development team as well as to the external world, to improve reusability of design artifacts, and, ultimately, to improve efficiency and effectiveness of development techniques.

The reported estimation is based on theoretical analysis and on our experience in employing ontologies in software and service engineering. Empirical validation is problematic to carry out due to lack of data on ontology-enabled software development projects. Nevertheless, we believe that not only our estimation can be used as an, admittedly optimistic, indication of overall viability of ontologies for software engineering practices; but also that the performed analysis revealed valuable insights on the interaction between ontologies and widely accepted quality characteristics of software products as well as of properties of the development process.

The results reported in this dissertation are based on the following publications (listed in chronological order):

1. Maksym Korotkiy and Jan L. Top: Blackboard-style Service Composition with Onto⇔SOA. In proceeding of the WWW/Internet 2007 conference. Vila Real, Portugal, 2007.

2. Maksym Korotkiy and Jan L. Top: Onto⇔SOA: From Ontology-enabled SOA to Service-enabled Ontologies. In proceedings of International Conference on Internet and Web Application and Services (ICIW'06). Guadeloupe, 2006.

3. Maksym Korotkiy and Jan L. Top: Designing a Document Retrieval Service with Onto⇔SOA. In proceedings of the Semantic Web Enabled Software Engineering workshop at ISWC 2006. Athens, GA, U.S.A., 2006.

4. Maksym Korotkiy: Towards an Ontology-enabled Service Oriented Architecture. In proceedings of the PhD Symposium in International Conference on Service Oriented Computing (ICSOC'05). Amsterdam, the Netherlands, 2005.

5. Maksym Korotkiy and Jan L. Top: MoRe Semantic Web Applications. In proceedings of the End-User Aspects of the Semantic Web Workshop. European Semantic Web Conference. Crete, 2005.

6. Maksym Korotkiy: On the Effect of Ontologies on Web Application Development Effort. In proceedings of the Knowledge Engineering and Software Engineering workshop. Koblenz, Germany, 2005.

7. Maksym Korotkiy: On the Effect of Ontologies on Quality of Web Applications. In proceedings of the Workshop on Building and Applying Ontologies for the Semantic Web. Portugal, 2005.

8. Maksym Korotkiy and Jan L. Top: From Relational Data to RDFS models. In proceedings of the International Conference on Web Engineering. Munich, 2004.

Summary

In this thesis we aim at improving usability and reusability of knowledge by combining ontologies and Service-Oriented Architectures. To achieve this we propose the Onto⇔SOA framework that describes how ontologies and services can be designed in a way that enables their *natural integration*. We maintain a consistent, simple and pragmatic approach that can be deployed without significant investment of effort. The design guidelines underlying Onto⇔SOA are a combination of well-established practices from the Software Engineering and Service-Oriented Architectures fields. Our approach complements state of the art research in the fields of Ontology Engineering, Semantic Web, Semantic Web Services, Service-Oriented Architectures and Software Engineering.

We apply Onto⇔SOA to a number of use cases from the e-Science domain – our target application domain. These use cases are: retrieving documents matching a given query (Chapters 2 and 3), conversion (Chapter 5) and consistency checking of units of measurement (Chapter 4). In these use cases we employ the guidelines proposed in Onto⇔SOA to design and implement ontology-based and document-oriented services that facilitate the above-mentioned e-Science tasks.

Onto⇔SOA provides a uniform ontology- and service-oriented framework for the solutions to the target cases. In Chapter 2 we introduce Onto⇔SOA as an architectural framework for Ontology-Enabled services. This framework is based on a restricted service model that constrains internal properties of a service to induce *domain alignment* and *loose coupling* characteristics. These constraints allow, among others, to simplify the model of a service and to provide guidelines on the design of ontology-enabled services.

Onto⇔SOA relies on direct exchange of ontology-based messages between a *document-oriented* service and its consumer; and employs an ontology as a *service schema* (referred to as a *service ontology*). The main purpose of ontologies is to transfer domain conceptualization to services (Service-Oriented Architectures), thus enhancing their domain (business) alignment.

In addition to the addressed e-Science use cases, in Chapter 6 we investigate the more general problem of linking relational and ontology-friendly data models. We propose the FDR2 approach that relies on three components: an automatically generated RDFS schema of tabular data, an RDF serialization of the data itself, and a manually created map. The proposed approach is purely RDF/S-based and does not require any additional software components except an RDFS reasoner. Moreover, the technique underlying FDR2 is general enough to be applicable to data sources different from relational ones. By explicating the schema of the original data, serializing the data according to that schema and linking the schema to a target ontology we can semantically enrich the data and improve its accessibility by ontology-enabled software.

To facilitate the evaluation and application of Onto⇔SOA to the targeted use cases, we further specialize it into *MoRe* (Chapter 5) – an operational framework and corresponding middleware based on RDF/S languages and REST Services. *MoRe* aims to provide a simple and pragmatic foundation for the development of ontology-based Web applications. We believe that it also facilitates bridging the gap between ontological domain knowledge and software development. On the one hand, it provides a pragmatic application-driven view on domain ontologies. On the other hand, it facilitates software development by integrating domain-specific inference services into software solutions.

Collaboration between services enables effective construction of complex functionality from simpler services and, thus is an important component of any service-oriented approach. In Chapter 4 we integrate a Blackboard-style composition into the Onto⇔SOA approach. The devised composition mechanism utilizes an application-independent controller and a homogeneously structured ontology-based repository (a blackboard). The proposed approach requires neither an extensive service model nor an explicit workflow specification and enables composite functionality to emerge by bringing a number of ontology-enabled services together.

By applying ontologies and services to the e-Science domain we have gained insights on how ontologies and services could benefit from each other. In Chapter 3 we introduce Service-enabled Ontologies – an ontology-oriented perspective on Onto⇔SOA that shifts the focus to service ontologies. Service-enabled Ontologies re-interpret Onto⇔SOA as a mechanism that allows to attach an arbitrary service to an ontology, thus capturing *application semantics* of domain concepts.

These service attachments can facilitate practical application of ontologies, potentially at the costs of their overall reusability. The trade-off between declarative (general, reusable

but difficult to utilize in practice), and procedural (application-specific but easy to exploit) knowledge still holds in Service-enabled Ontologies. Finding a suitable balance between these two ways of representing knowledge ultimately depends on the requirements of a certain application scenario. With Service-enabled Ontologies we provide a novel framework that contributes to flexibility in specifying a domain conceptualization.

In this dissertation we also investigate the overall impact of ontologies on software engineering practice. In Chapter 7 we analyze the effect of ontologies on existing models for estimating software quality (Quint2) and development effort (WEBMO). We estimate that ontology can improve many quality dimensions by supplying domain conceptualization. This conceptualization can be employed across all development stages amplifying the cumulative effect of ontologies. We also optimistically estimate that in the long run, ontologies can cause a significant decrease of development effort. This becomes possible due to the ability of ontologies to facilitate transfer of application domain knowledge into a development team, to provide a unified conceptual view improving communication within the development team as well as to external world, to improve reusability of design artifacts, and, ultimately, to improve efficiency and effectiveness of development techniques.

The reported estimation is based on theoretical analysis and our experience in employing ontologies in software and service engineering. Empirical validation is problematic to carry out due to lack of data on ontology-enabled software development projects. Nevertheless, we believe that not only our estimation can be used as an, admittedly optimistic, indication of overall viability of ontologies for software engineering practices; but also that the performed analysis reveals valuable insights on the interaction between ontologies and widely accepted quality characteristics of software products as well as of properties of the development process.

Samenvatting

In dit proefschrift streven wij ernaar de bruikbaarheid en herbruikbaarheid van kennis te verbeteren door ontologieën te combineren met service-oriented architectures. Om dit doel te bereiken hebben wij het raamwerk Onto⇔SOA ontwikkeld. Dit raamwerk beschrijft hoe ontologieën en services op *natuurlijke wijze* kunnen worden gecombineerd. Het doel is om tot een eenvoudige en pragmatische oplossing te komen die geen significante investeringen vraagt. De ontwerpregels die aan Onto⇔SOA ten grondslag liggen combineren reeds lang gevestigde praktijken uit de software engineering met de SOA-aanpak. Onze werkwijze is aanvullend op die van bestaande methoden uit ontology engineering, Semantisch Web onderzoek en software engineering per se.

In dit werk passen wij Onto⇔SOA toe op een aantal use cases uit het e-science domein. Deze use cases zijn het zoeken naar informatie in documenten (Hoofdstukken 2 en 3) en de conversie (Hoofdstuk 5) en controle op consistentie van eenheden (Hoofdstuk 4). In deze use cases passen wij de Onto⇔SOA ontwerpregels toe om ontologie-gebaseerde en document-georiënteerde services te ontwerpen en implementeren voor de genoemde e-science taken. We zien dat de gekozen aanpak de oplossingen voor deze use cases vereenvoudigt.

Onto⇔SOA definieert een uniform raamwerk voor het behandelen van de use cases, gebaseerd op het combineren van ontologieën en SOA. In Hoofdstuk 2 introduceren wij Onto⇔SOA als een architectuurmodel voor ontologie-ondersteunde services. Dit raamwerk is gebaseerd op bepaalde randvoorwaarden die worden opgelegd aan de interne eigenschappen van het servicemodel. Deze randvoorwaarden maken *losse koppeling* en *domeinaansluiting* mogelijk. In het bijzonder maken deze randvoorwaarden het mogelijk om het servicemodel te vereenvoudigen en om ontwerpregels van ontologie-ondersteunde services op te stellen.

Onto⇔SOA baseert zich op de directe uitwisseling van op ontologie-gebaseerde berichten tussen *document-georiënteerde* services en de gebruiker van die services. Het raamwerk

gebruikt de ontologie als *de serviceschema* (dat als *serviceontologie* wordt bedoeld). De belangrijkste functie van de ontologie is om domeinconceptualisering naar de services (en SOA) over te brengen. Hierdoor sluiten de services beter aan op het beoogde domein (de business).

Wij ontwikkelen Onto⇔SOA verder in *MoRe* (Hoofdstuk 5) – een operationeel raamwerk en middleware gebaseerd op RDF/S en REST-Services. Het doel van *MoRe* is om een eenvoudige en pragmatische basis voor de ontwikkeling van op ontologie-gebaseerde toepassingen te verschaffen. Wij geloven dat het daarnaast ook het gat overbrugt tussen het modelleren van ontologisch domeinkennis en softwareontwikkeling. Enerzijds geeft het een pragmatische interpretatie van de toegepaste domeinontologieën. Anderzijds vergemakkelijkt het softwareontwikkeling door aan het vakgebied verbonden redeneersystemen in software te integreren.

Samenwerking tussen services is een belangrijk element van SOA. In Hoofdstuk 4 ontwikkelen wij een methode voor het combineren van Onto⇔SOA services gebaseerd op Blackboard. De voorgestelde aanpak gebruikt een applicatie-onafhankelijk besturingsmechanisme en een op ontologieën gebaseerde tussenopslag (het "schoolbord"). De voorgestelde benadering vereist geen uitgebreid servicemodel, noch een expliciete specificatie van de werkstroom. Het maakt samengestelde functionaliteit mogelijk door simpelweg een aantal ontologie-ondersteunde services bij elkaar te brengen.

Door ontologieën en services in het e-science domein toe te passen hebben we inzicht gekregen over hoe ontologieën en services van elkaar kunnen profiteren. In Hoofdstuk 3 introduceren we service-ondersteunende ontologieën – een ontologie-georiënteerd perspectief op Onto⇔SOA dat de nadruk legt op serviceontologieën. Service-ondersteunde ontologieën herinterpreteren Onto⇔SOA als een mechanisme om een 'willekeurige' service aan een ontologie te verbinden. Deze verbinding legt de *toepassingssemantiek* van de gebruikte domeinconcepten vast.

Dergelijke ontologie-gebonden services kunnen de praktische toepassing van ontologieën vergemakkelijken, hoewel dat mogelijk ten koste gaat van hun herbruikbaarheid. De afweging tussen het gebruik van declaratieve kennis (algemeen, hergebruikbaar maar moeilijk in de praktijk te gebruiken), en procedurele kennis (applicatie-afhankelijk maar gemakkelijk te in te zetten) moet nog steeds gemaakt worden. Het bepalen van de trade-off tussen deze twee manieren om kennis te representeren hangt uiteindelijk af van het specifieke toepassingsscenario. Met service-ondersteunde ontologieën vergroten wij de flexibiliteit bij het specificeren van domeinkennis.

In dit proefschrift onderzoeken wij ook het algemene effect van ontologieën op de praktijk van software engineering. In Hoofdstuk 7 analyseren we het effect van het gebruik van ontologieën op bestaande modellen voor het schatten van softwarekwaliteit (Quint2) en ontwikkelingskosten (WEBMO). Wij verwachten dat ontologieën langs veel dimensies de kwaliteit kunnen verbeteren door het leveren van domeinmodellen. Deze modellen kunnen in alle ontwikkelingsstadia worden gebruikt en zo het cumulatieve effect vergroten. Wij zijn ook optimistisch over het feit dat ontologieën uiteindelijk een significante daling van de ontwikkelingskosten kunnen veroorzaken.

De gegeven schatting is gebaseerd op een theoretische analyse en op onze eigen ervaring in het toepassen van ontologieën in software en service engineering. De empirische bevestiging ervan is problematisch door een gebrek aan praktijkgegevens over ontologie-ondersteunde softwareontwikkeling. Niettemin geloven wij dat onze schatting kan worden gezien als een aanwijzing dat het gebruik van ontologieën in software engineering zin heeft. Bovendien geeft de uitgevoerde analyse zicht op mogelijke specifieke effecten van ontologieën op de kwaliteitskenmerken van softwareproducten en op het ontwikkelingsproces.

Bibliography

[1] Donald J. Reifer, "Estimating Web Development Costs: There Are Differences," *The Journal of Defense Software Engineering*, June 2002.

[2] Plato, *Plato Complete Works*, Hackett Publishing Company, 1997.

[3] Robert Nozick, *Philosophical Explanations*, Belknap Press, 2006.

[4] Ludwig Wittgenstein, *On Certainty*, Wiley-Blackwell, 1991.

[5] Peter Drucker, *The Age of Discontinuity: Guidelines to our Changing Society*, Harper and Row, New York, 1969.

[6] Guus Schreiber, Hans Akkermans, Anjo Anjewierden, Robert Dehoog, Nigel Shadbolt, Walter Vandevelde, and Bob Wielinga, *Knowledge Engineering and Management: The CommonKADS Methodology*, The MIT Press, December 1999.

[7] W. K. C Guthrie, *A History of Greek Philosophy*, Cambridge University Press, Melbourne, 1986.

[8] Rene Descartes, *Discourse on Method and Related Writings*, Penguin Classics, 2000.

[9] Kevin H. O'Rourke, Ahmed S. Rahman, and Alan M. Taylor, "Trade, Knowledge, and the Industrial Revolution," *NBER Working Papers*, vol. 13057, 2007.

[10] Tim Berners-Lee, James Hendler, and Ora Lassila, "The semantic web," *Scientific american*, vol. 279(5), pp. 35–43, 2001.

[11] Thomas. R. Gruber, "Towards Principles for the Design of Ontologies Used for Knowledge Sharing," *Formal Ontology in Conceptual Analysis and Knowledge Representation*, 1993.

[12] Nicola Guarino and Pierdaniele Giaretta, "Ontologies and Knowledge Bases: Towards a Terminological Clarification," *Towards Very Large Knowledge Bases: Knowledge Building and Knowledge Sharing*, pp. 25–32, 1995.

[13] W3C, "RDF Semantics," W3C Recommendation. http://www.w3.org/TR/rdf-mt/, February 2004.

[14] Dan Brickley and R.V. Guha, "RDF Vocabulary Description Language 1.0: RDF Schema," W3C Recommendation. http://www.w3.org/TR/rdf-schema/, February 2004.

[15] W3C, "OWL Web Ontology Language Semantics and Abstract Syntax," www.w3.org.

[16] W3C, "Semantic Web Recommendations and Specifications: RDF/S, OWL, etc," http://www.w3.org/sw.

[17] Myo-Myo Naing, Ee-Peng Lim, and Dion Goh Hoe-Lian, "Ontology-based Web Annotation Framework for HyperLink Structures," in *WISEW '02: Proceedings of the Third International Conference on Web Information Systems Engineering (Workshops) - (WISEw'02)*, Washington, DC, USA, 2002, p. 184, IEEE Computer Society.

[18] Abhijit A. Patil, Swapna A. Oundhakar, Amit P. Sheth, and Kunal Verma, "Meteor-s web service annotation framework," in *WWW '04: Proceedings of the 13th international conference on World Wide Web*, New York, NY, USA, 2004, pp. 553–562, ACM.

[19] Zhang Duo, Li Juan-Zi, and Xu Bin, "Web Service Annotation Using Ontology Mapping," in *SOSE '05: Proceedings of the IEEE International Workshop*, Washington, DC, USA, 2005, pp. 243–250, IEEE Computer Society.

[20] David Parry, "A fuzzy ontology for medical document retrieval," in *ACSW Frontiers '04: Proceedings of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation*, Darlinghurst, Australia, Australia, 2004, pp. 121–126, Australian Computer Society, Inc.

[21] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, Eds., *The description logic handbook: theory, implementation, and applications*, Cambridge University Press, New York, NY, USA, 2003.

[22] Ziv Baida, Jaap Gordijn, Borys Omelayenko, and Hans Akkermans, "A Shared Service Terminology for Online Service Provisioning," in *Proceedings of the Sixth International Conference on Electronic Commerce (ICEC04)*. 2004, ACM Press.

[23] John Taylor, "Research Council e-Science Core Programme," http://www.research-councils.ac.uk/escience/.

[24] Karey Harrison and Kathleen Fahy, *Methods of research in sports science: quantitative and qualitative approaches.*, Chapter Constructive research: methodology and practice, pp. 660–700, Meyer and Meyer, 2005.

[25] W3C, "Web Services: Recommendations, Specifications and Documents (WSDL, SOAP, etc)," http://www.w3.org/2002/ws.

[26] Thomas Erl, *SOA Design Patterns*, Prentice Hall PTR, New Jersey, USA, 2008.

[27] Michael Bell, *Service-Oriented Modeling (SOA): Service Analysis, Design, and Architecture*, Wiley, New Jersey, USA, 2008.

[28] W3C, "OWL-S: Semantic Markup for Web Services," http://www.w3.org.

[29] D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel, "Web Service Modeling Ontology," *Applied Ontology*, vol. 1, no. 1, pp. 77–106, 2005.

[30] M. Hotle, "A conceptual evolution: From process to web services," Gartner Group Research Note TU-16-1420, May 2003.

[31] Roy Thomas Fielding, *Architectural Styles and the Design of Network-based Software Architectures*, Ph.D. thesis, UNIVERSITY OF CALIFORNIA, 2005.

[32] Dewayne E. Perry and Alexander L. Wolf, "Foundations for the Study of Software Architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, 1992.

[33] Mary Shaw and Paul C. Clements, "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems," in *COMPSAC '97: Proceedings of the 21st International Computer Software and Applications Conference*, Washington, DC, USA, 1997, pp. 6–13, IEEE Computer Society.

[34] Grady Booch, *Object-Oriented Analysis and Design with Applications*, Benjamin–Cummings, Redwood City, Calif., 2nd Edition, 1994.

[35] Hugo Haas and Allen Brown, "Web Services Glossary," W3C Working Group Note. http://www.w3.org/TR/ws-gloss/, February 2004.

[36] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns : Elements of Reusable Object-Oriented Software*, Addison Wesley, March 1995.

[37] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana, "Web Services Description Language (WSDL) 1.1," W3C Note. http://www.w3.org/TR/wsdl, March 2001.

[38] Daniel Corkill, "Blackboard Systems," *AI Expert*, vol. 6, no. 9, January 1991.

[39] Paul A. Strassmann, *The squandered computer: evaluating the business alignment of information technologies*, Information Economics Press, New Canaan, CT, USA, 1997.

[40] P.J. Gawthrop and D.J. Ballance, "Symbolic computation for manipulation of hierarchical bond graphs," *Symbolic Methods in Control System Analysis and Design, N. Munro (ed*, 1999.

[41] Maksym Korotkiy and Jan Top, "MoRe Semantic Web Applications," in *Proceedings of the ESWC'05 workshop on User Aspects of the Semantic Web*, 2005.

[42] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard, "Web Services Architecture.," W3C Working Group Note. http://www.w3.org/TR/ws-arch, February 2004.

[43] L. Cabral, J. Domingue, and et al, "Approaches to Semantic Web Services: an Overview and Comparisons.," in *ESWS*, Christoph Bussler, John Davies, Dieter Fensel, and Rudi Studer, Eds. 2004, vol. 3053 of *Lecture Notes in Computer Science*, pp. 225–239, Springer.

[44] D Martin, M Paolucci, S McIlraith, M Burstein, D McDermott, D McGuinness, B Parsia, T Payne, M Sabou, M Solanki, N Srinivasan, and K Sycara, "Bringing Semantics to Web Services: The OWL-S Approach.," in *Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition*, 2004.

[45] Namyoun Choi, Il-Yeol Song, and Hyoil Han, "A survey on ontology mapping," *SIGMOD Rec.*, vol. 35, no. 3, pp. 34–41, September 2006.

[46] Alexander Borgida, Ronald J. Brachman, Deborah L. McGuinness, and Lori Alperin Resnick, "CLASSIC: a structural data model for objects," 1989, pp. 58–67.

[47] R. G. Brachman and J. G. Schmolze, "An overview of the KL-ONE knowledge representation system," *Cognitive Science*, vol. 9, pp. 171–216, 1985.

[48] NASA, "CLIPS Basic Programming Guide," JSC-25012.

[49] Fensel D. Studer R., Benjamins V. R., "Knowledge engineering: Principles and methods," *Data Knowledge Engineering*, vol. 25(1-2), pp. 161–197, 1998.

[50] D. McGuinness, *Spinning the Semantic Web: bringing the World Wide Web to Its Full Potential*, Chapter Ontologies Come of Age, MIT Press, 2002.

[51] M. Fernandez, A. Gomez-Perez, and N. Juristo, "METHONTOLOGY: From Ontological Arts Towards Ontological Engineering," in *In Proceedings of the AAAI-97 Spring Symposium Series on Ontological Engineering*, 1997.

[52] N. Noy and D. McGuinness, "Ontology Development 101: A Guide to Creating Your First Ontology," Tech. Rep., Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report SMI-2001-0880, 2001.

[53] B. McBride, "Four steps towards the widespread adoption of a semantic web.," in *Proceedings of the First International Semantic Web Conference*, Sardinia, Italy, 2002.

[54] Oren Etzioni, Steven Gribble, Alon Halevy, Hank Levy, and Luke McDowell, "An evolutionary approach to the semantic web.," In Poster Presentation at the First International Semantic Web Conference, 2002.

[55] Jr. Fred P. Brooks, "The Mythical Man-Month," in *Proceedings of the international conference on Reliable software*, New York, NY, USA, 1975, p. 193, ACM Press.

[56] Karl Pfleger and Barbara Hayes-Roth, "An Introduction to Blackboard-style Systems Organization," Tech. Rep., Stanford University, 1997.

[57] HP Labs Semantic Web Activity, "Jena Semantic Web Toolkit," http://www.hpl.hp.com/semweb/.

[58] W3C, "SPARQL Query Language for RDF. W3C Candidate Recommendation.," http://www.w3.org/TR/rdf-sparql-query/.

[59] Farshad Hakimpour, Denilson Sell, Liliana Cabral, John Domingue, and Enrico Motta, "Semantic Web Service Composition in IRS-III: The Structured Approach," in *CEC '05: Proceedings of the Seventh IEEE International Conference on E-Commerce Technology (CEC'05)*, Washington, DC, USA, 2005, pp. 484–487, IEEE Computer Society.

[60] Robert Engelmore and Tony Morgan, Eds., *Blackboard Systems*, Addison-Wesley, 1988.

[61] Dennis Quan, David Huynh, and David R. Karger, "Haystack: A Platform for Authoring End User Semantic Web Applications.," *International Semantic Web Conference*, pp. 738–753, 2003.

[62] Borislav Popov, Atanas Kiryakov, Angel Kirilov, Dimitar Manov, Damyan Ognyanoff, and Miroslav Goranov, "KIM - Semantic Annotation Platform.," *International Semantic Web Conference*, pp. 834–849, 2003.

[63] Martin Dzbor, Enrico Motta, and John Domingue, "Opening Up Magpie via Semantic Services.," in *International Semantic Web Conference*, Sheila A. McIlraith, Dimitris Plexousakis, and Frank van Harmelen, Eds. 2004, vol. 3298 of *Lecture Notes in Computer Science*, pp. 635–649, Springer.

[64] Enrico Motta, John Domingue, Liliana Cabral, and Mauro Gaspari, "IRS-II: A Framework and Infrastructure for Semantic Web Services.," *International Semantic Web Conference*, pp. 306–318, 2003.

[65] J. Broekstra, A. Kampman, and F. van Harmelen, "Sesame: An Architecture for Storing and Querying RDF Data and Schema Information," *In D. Fensel, J. Hendler, H. Lieberman, and W. Wahlster, editors, Semantics for the WWW. MIT Press.*, 2001.

[66] Roy Thomas Fielding, "Principled Design of the Modern Web Architecture," in *Proceedings of the 2000 International Conference on Software Engineering*, 2000.

[67] W3C, "Resource Description Framework (RDF)," http://www.w3.org/RDF/.

[68] C. Fahner and G. Vossen, "A survey of database design transformations based on the entity-relationship model," *Data and Knowledge Engineering*, vol. 15, pp. 213–250, 1995.

[69] R. H. L. Chiang, T. M. Barron, and V. C. Storey, "Performance evaluation of reverse engineering relational databases into extended entity-relationship models.," in *In Proceedings of the 12th international conference on entity relationship approach*, USA, Texas, 1993, pp. 352–363, Springer.

[70] Chris Bizer, "D2R MAP - Database to RDF Mapping Language and Processor," http://www.wiwiss.fu-berlin.de/suhl/bizer/d2rmap/D2Rmap.htm.

[71] C. Ramanathan, *Providing Object-Oriented Access to relational Databases*, Ph.D. thesis, Mississippi State University, 1997.

[72] objectarchitects.de, "ARCUS Project," http://www.objectarchitects.de.

[73] A. M. Keller and G. Wiederhold, "Penguin: Objects for Programs, Relations for Persistence," *Succeeding With Object Databases*, 2001.

[74] Dave Beckett and Jan Grant, "Semantic Web Scalability and Storage: Mapping Semantic Web Data with RDBMSes," SWAD-Europe deliverable, W3C, 2003.

[75] B. Omelayenko, "RDFT: A Mapping Meta-Ontology for Business Integration," in *Proceedings of the Workshop on Knowledge Transformation for the Semantic Web at the 15th European Conference on Artificial Intelligence*, Lyon, France, 2002, pp. 77–84.

[76] ICS-FORTH, "The ICS-FORTH RDFSuite: High-level Scalable Tools for the Semantic Web," http://139.91.183.30:9090/RDF/.

[77] R. Paulussen en J. Trienekens B. van Zeist, P. Hendriks, "Quint2: The Extended ISO Model of Software Quality," http://www.serc.nl/quint-book/, 1996.

[78] Mike Uschold and Michael Grüninger, "Ontologies: principles, methods, and applications," *Knowledge Engineering Review*, vol. 11, no. 2, pp. 93–155, 1996.

[79] M. Musen, "Ontology-oriented design and programming," *In: Cuena, J., Demazeau, Y., Garcia, A., and Treur, J., eds. Knowledge Engineering and Agent Technology. Amsterdam: IOS Press, in press*, 2000.

[80] Sari A. Laakso, "User Interface Design Patterns," http://www.cs.helsinki.fi/u/salaakso/patterns/index.html.

[81] Donald J. Reifer, "Web Objects Counting Conventions," http://www.reifer.com.

[82] Donald J. Reifer, "Let the Numbers Do the Talking," *The Journal of Defense Software Engineering*, March 2002.

[83] M. Uschold, M. Healy, K. Williamson, P. Clark, and S. Woods, "Ontology reuse and application," *In proceedings of the International Conference on Formal Ontology and Information Systems - FOIS'98*, 1998.

[84] Jakob Nielsen, "User interface directions for the Web," *Communications of the ACM*, vol. 42, no. 1, pp. 65–72, 1999.

[85] The Mozilla Organization, "XUL," http://www.mozilla.org/projects/xul/.

[86] W3C, "XForms 1.0 Recommendation," http://www.w3.org/TR/xforms/.

SIKS Dissertation Series

1998-1 Johan van den Akker (CWI)
DEGAS - An Active, Temporal Database of Autonomous Objects

1998-2 Floris Wiesman (UM)
Information Retrieval by Graphically Browsing Meta-Information

1998-3 Ans Steuten (TUD)
A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective

1998-4 Dennis Breuker (UM)
Memory versus Search in Games

1998-5 E.W.Oskamp (RUL)
Computerondersteuning bij Straftoemeting

1999-1 Mark Sloof (VU)
Physiology of Quality Change Modelling;
Automated modelling of Quality Change of Agricultural Products

1999-2 Rob Potharst (EUR)
Classification using decision trees and neural nets

1999-3 Don Beal (UM)
The Nature of Minimax Search

1999-4 Jacques Penders (UM)
The practical Art of Moving Physical Objects

1999-5 Aldo de Moor (KUB)
Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems

1999-6 Niek J.E. Wijngaards (VU)
Re-design of compositional systems

1999-7 David Spelt (UT)
Verification support for object database design

1999-8 Jacques H.J. Lenting (UM)
Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation.

2000-1 Frank Niessink (VU)
Perspectives on Improving Software Maintenance

2000-2 Koen Holtman (TUE)
Prototyping of CMS Storage Management

2000-3 Carolien M.T. Metselaar (UVA)
Sociaal-organisatorische gevolgen van kennistechnologie; een procesbenadering en actorperspectief.

2000-4 Geert de Haan (VU)
ETAG, A Formal Model of Competence Knowledge for User Interface Design

2000-5 Ruud van der Pol (UM)
Knowledge-based Query Formulation in Information Retrieval.

2000-6 Rogier van Eijk (UU)
Programming Languages for Agent Communication

2000-7 Niels Peek (UU)
Decision-theoretic Planning of Clinical Patient Management

2000-8 Veerle Coup (EUR)
Sensitivity Analyis of Decision-Theoretic Networks

2000-9 Florian Waas (CWI)
Principles of Probabilistic Query Optimization

2000-10 Niels Nes (CWI)
Image Database Management System Design Considerations, Algorithms and Architecture

2000-11 Jonas Karlsson (CWI)

Scalable Distributed Data Structures for Database Management

2001-1 Silja Renooij (UU)
Qualitative Approaches to Quantifying Probabilistic Networks

2001-2 Koen Hindriks (UU)
Agent Programming Languages: Programming with Mental Models

2001-3 Maarten van Someren (UvA)
Learning as problem solving

2001-4 Evgueni Smirnov (UM)
Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets

2001-5 Jacco van Ossenbruggen (VU)
Processing Structured Hypermedia: A Matter of Style

2001-6 Martijn van Welie (VU)
Task-based User Interface Design

2001-7 Bastiaan Schonhage (VU)
Diva: Architectural Perspectives on Information Visualization

2001-8 Pascal van Eck (VU)
A Compositional Semantic Structure for Multi-Agent Systems Dynamics.

2001-9 Pieter Jan 't Hoen (RUL)
Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes

2001-10 Maarten Sierhuis (UvA)
Modeling and Simulating Work Practice
BRAHMS: a multiagent modeling and simulation language for work practice analysis and design

2001-11 Tom M. van Engers (VUA)
Knowledge Management:
The Role of Mental Models in Business Systems Design

2002-01 Nico Lassing (VU)
Architecture-Level Modifiability Analysis

2002-02 Roelof van Zwol (UT)
Modelling and searching web-based document collections

2002-03 Henk Ernst Blok (UT)
Database Optimization Aspects for Information Retrieval

2002-04 Juan Roberto Castelo Valdueza (UU)
The Discrete Acyclic Digraph Markov Model in Data Mining

2002-05 Radu Serban (VU)
The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents

2002-06 Laurens Mommers (UL)
Applied legal epistemology;
Building a knowledge-based ontology of the legal domain

2002-07 Peter Boncz (CWI)
Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications

2002-08 Jaap Gordijn (VU)
Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas

2002-09 Willem-Jan van den Heuvel(KUB)
Integrating Modern Business Applications with Objectified Legacy Systems

2002-10 Brian Sheppard (UM)
Towards Perfect Play of Scrabble

2002-11 Wouter C.A. Wijngaards (VU)
Agent Based Modelling of Dynamics: Biological and Organisational Applications

2002-12 Albrecht Schmidt (Uva)
Processing XML in Database Systems

2002-13 Hongjing Wu (TUE)
A Reference Architecture for Adaptive Hypermedia Applications

2002-14 Wieke de Vries (UU)
Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems

2002-15 Rik Eshuis (UT)
Semantics and Verification of UML Activity Diagrams for Workflow Modelling

2002-16 Pieter van Langen (VU)
The Anatomy of Design: Foundations, Models and Applications

2002-17 Stefan Manegold (UVA)
Understanding, Modeling, and Improving Main-Memory Database Performance

2003-01 Heiner Stuckenschmidt (VU)
Ontology-Based Information Sharing in Weakly Structured Environments

2003-02 Jan Broersen (VU)
Modal Action Logics for Reasoning About Reactive Systems

2003-03 Martijn Schuemie (TUD)
Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy

2003-04 Milan Petkovic (UT)
Content-Based Video Retrieval Supported by Database Technology

2003-05 Jos Lehmann (UVA)
Causation in Artificial Intelligence and Law - A modelling approach

2003-06 Boris van Schooten (UT)
Development and specification of virtual environments

2003-07 Machiel Jansen (UvA)
Formal Explorations of Knowledge Intensive Tasks

2003-08 Yongping Ran (UM)
Repair Based Scheduling

2003-09 Rens Kortmann (UM)
The resolution of visually guided behaviour

2003-10 Andreas Lincke (UvT)
Electronic Business Negotiation: Some experimental studies on the interaction between medium, innovation context and culture

2003-11 Simon Keizer (UT)
Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks

2003-12 Roeland Ordelman (UT)
Dutch speech recognition in multimedia information retrieval

2003-13 Jeroen Donkers (UM)
Nosce Hostem - Searching with Opponent Models

2003-14 Stijn Hoppenbrouwers (KUN)
Freezing Language: Conceptualisation Processes across ICT-Supported Organisations

2003-15 Mathijs de Weerdt (TUD)
Plan Merging in Multi-Agent Systems

2003-16 Menzo Windhouwer (CWI)
Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouses

2003-17 David Jansen (UT)
Extensions of Statecharts with Probability, Time, and Stochastic Timing

2003-18 Levente Kocsis (UM)
Learning Search Decisions

2004-01 Virginia Dignum (UU)
A Model for Organizational Interaction: Based on Agents, Founded in Logic

2004-02 Lai Xu (UvT)
Monitoring Multi-party Contracts for E-business

2004-03 Perry Groot (VU)
A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving

2004-04 Chris van Aart (UVA)
Organizational Principles for Multi-Agent Architectures

2004-05 Viara Popova (EUR)
Knowledge discovery and monotonicity

2004-06 Bart-Jan Hommes (TUD)
The Evaluation of Business Process Modeling Techniques

2004-07 Elise Boltjes (UM)
Voorbeeldig onderwijs; voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes

2004-08 Joop Verbeek(UM)

Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale politiële gegevensuitwisseling en digitale expertise

2004-09 Martin Caminada (VU)
For the Sake of the Argument; explorations into argument-based reasoning

2004-10 Suzanne Kabel (UVA)
Knowledge-rich indexing of learning-objects

2004-11 Michel Klein (VU)
Change Management for Distributed Ontologies

2004-12 The Duy Bui (UT)
Creating emotions and facial expressions for embodied agents

2004-13 Wojciech Jamroga (UT)
Using Multiple Models of Reality: On Agents who Know how to Play

2004-14 Paul Harrenstein (UU)
Logic in Conflict. Logical Explorations in Strategic Equilibrium

2004-15 Arno Knobbe (UU)
Multi-Relational Data Mining

2004-16 Federico Divina (VU)
Hybrid Genetic Relational Search for Inductive Learning

2004-17 Mark Winands (UM)
Informed Search in Complex Games

2004-18 Vania Bessa Machado (UvA)
Supporting the Construction of Qualitative Knowledge Models

2004-19 Thijs Westerveld (UT)
Using generative probabilistic models for multimedia retrieval

2004-20 Madelon Evers (Nyenrode)
Learning from Design: facilitating multidisciplinary design teams

2005-01 Floor Verdenius (UVA)
Methodological Aspects of Designing Induction-Based Applications

2005-02 Erik van der Werf (UM))
AI techniques for the game of Go

2005-03 Franc Grootjen (RUN)
A Pragmatic Approach to the Conceptualisation of Language

2005-04 Nirvana Meratnia (UT)
Towards Database Support for Moving Object data

2005-05 Gabriel Infante-Lopez (UVA)
Two-Level Probabilistic Grammars for Natural Language Parsing

2005-06 Pieter Spronck (UM)
Adaptive Game AI

2005-07 Flavius Frasincar (TUE)
Hypermedia Presentation Generation for Semantic Web Information Systems

2005-08 Richard Vdovjak (TUE)
A Model-driven Approach for Building Distributed Ontology-based Web Applications

2005-09 Jeen Broekstra (VU)
Storage, Querying and Inferencing for Semantic Web Languages

2005-10 Anders Bouwer (UVA)
Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments

2005-11 Elth Ogston (VU)
Agent Based Matchmaking and Clustering - A Decentralized Approach to Search

2005-12 Csaba Boer (EUR)
Distributed Simulation in Industry

2005-13 Fred Hamburg (UL)
Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen

2005-14 Borys Omelayenko (VU)
Web-Service configuration on the Semantic Web; Exploring how semantics meets pragmatics

2005-15 Tibor Bosse (VU)
Analysis of the Dynamics of Cognitive Processes

2005-16 Joris Graaumans (UU)
Usability of XML Query Languages

2005-17 Boris Shishkov (TUD)
Software Specification Based on Re-usable Business Components

2005-18 Danielle Sent (UU)
Test-selection strategies for probabilistic networks

2005-19 Michel van Dartel (UM)
Situated Representation

2005-20 Cristina Coteanu (UL)
Cyber Consumer Law, State of the Art and Perspectives

2005-21 Wijnand Derks (UT)
Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics

2006-01 Samuil Angelov (TUE)
Foundations of B2B Electronic Contracting

2006-02 Cristina Chisalita (VU)
Contextual issues in the design and use of information technology in organizations

2006-03 Noor Christoph (UVA)
The role of metacognitive skills in learning to solve problems

2006-04 Marta Sabou (VU)
Building Web Service Ontologies

2006-05 Cees Pierik (UU)
Validation Techniques for Object-Oriented Proof Outlines

2006-06 Ziv Baida (VU)
Software-aided Service Bundling - Intelligent Methods & Tools for Graphical Service Modeling

2006-07 Marko Smiljanic (UT)
XML schema matching – balancing efficiency and effectiveness by means of clustering

2006-08 Eelco Herder (UT)
Forward, Back and Home Again - Analyzing User Behavior on the Web

2006-09 Mohamed Wahdan (UM)
Automatic Formulation of the Auditor's Opinion

2006-10 Ronny Siebes (VU)
Semantic Routing in Peer-to-Peer Systems

2006-11 Joeri van Ruth (UT)
Flattening Queries over Nested Data Types

2006-12 Bert Bongers (VU)
Interactivation - Towards an e-cology of people, our technological environment, and the arts

2006-13 Henk-Jan Lebbink (UU)
Dialogue and Decision Games for Information Exchanging Agents

2006-14 Johan Hoorn (VU)
Software Requirements: Update, Upgrade, Redesign - towards a Theory of Requirements Change

2006-15 Rainer Malik (UU)
CONAN: Text Mining in the Biomedical Domain

2006-16 Carsten Riggelsen (UU)
Approximation Methods for Efficient Learning of Bayesian Networks

2006-17 Stacey Nagata (UU)
User Assistance for Multitasking with Interruptions on a Mobile Device

2006-18 Valentin Zhizhkun (UVA)
Graph transformation for Natural Language Processing

2006-19 Birna van Riemsdijk (UU)
Cognitive Agent Programming: A Semantic Approach

2006-20 Marina Velikova (UvT)
Monotone models for prediction in data mining

2006-21 Bas van Gils (RUN)
Aptness on the Web

2006-22 Paul de Vrieze (RUN)
Fundaments of Adaptive Personalisation

2006-23 Ion Juvina (UU)
Development of Cognitive Model for Navigating on the Web

2006-24 Laura Hollink (VU)
Semantic Annotation for Retrieval of Visual Resources

2006-25 Madalina Drugan (UU)
Conditional log-likelihood MDL and Evolutionary MCMC

2006-26 Vojkan Mihajlovic (UT)
Score Region Algebra: A Flexible Framework for Structured Information Retrieval

2006-27 Stefano Bocconi (CWI)
Vox Populi: generating video documentaries from semantically annotated media repositories

2006-28 Borkur Sigurbjornsson (UVA)
Focused Information Access using XML Element Retrieval

2007-01 Kees Leune (UvT)
Access Control and Service-Oriented Architectures

2007-02 Wouter Teepe (RUG)
Reconciling Information Exchange and Confidentiality: A Formal Approach

2007-03 Peter Mika (VU)
Social Networks and the Semantic Web

2007-04 Jurriaan van Diggelen (UU)
Achieving Semantic Interoperability in Multi-agent Systems: a dialogue-based approach

2007-05 Bart Schermer (UL)
Software Agents, Surveillance, and the Right to Privacy: a Legislative Framework for Agent-enabled Surveillance

2007-06 Gilad Mishne (UVA)
Applied Text Analytics for Blogs

2007-07 Natasa Jovanovic' (UT)
To Whom It May Concern - Addressee Identification in Face-to-Face Meetings

2007-08 Mark Hoogendoorn (VU)
Modeling of Change in Multi-Agent Organizations

2007-09 David Mobach (VU)
Agent-Based Mediated Service Negotiation

2007-10 Huib Aldewereld (UU)
Autonomy vs. Conformity: an Institutional Perspective on Norms and Protocols

2007-11 Natalia Stash (TUE)
Incorporating Cognitive/Learning Styles in a General-Purpose Adaptive Hypermedia System

2007-12 Marcel van Gerven (RUN)
Bayesian Networks for Clinical Decision Support: A Rational Approach to Dynamic Decision-Making under Uncertainty

2007-13 Rutger Rienks (UT)
Meetings in Smart Environments; Implications of Progressing Technology

2007-14 Niek Bergboer (UM)
Context-Based Image Analysis

2007-15 Joyca Lacroix (UM)
NIM: a Situated Computational Memory Model

2007-16 Davide Grossi (UU)
Designing Invisible Handcuffs. Formal investigations in Institutions and Organizations for Multi-agent Systems

2007-17 Theodore Charitos (UU)
Reasoning with Dynamic Networks in Practice

2007-18 Bart Orriens (UvT)
On the development an management of adaptive business collaborations

2007-19 David Levy (UM)
Intimate relationships with artificial partners

2007-20 Slinger Jansen (UU)
Customer Configuration Updating in a Software Supply Network

2007-21 Karianne Vermaas (UU)
Fast diffusion and broadening use: A research on residential adoption and usage of broadband internet in the Netherlands between 2001 and 2005

2007-22 Zlatko Zlatev (UT)
Goal-oriented design of value and process models from patterns

2007-23 Peter Barna (TUE)
Specification of Application Logic in Web Information Systems

2007-24 Georgina Ramrez Camps (CWI)
Structural Features in XML Retrieval

2007-25 Joost Schalken (VU)
Empirical Investigations in Software Process Improvement

2008-01 Katalin Boer-Sorbn (EUR)
Agent-Based Simulation of Financial Markets: A modular, continuous-time approach

2008-02 Alexei Sharpanskykh (VU)
On Computer-Aided Methods for Modeling and Analysis of Organizations

2008-03 Vera Hollink (UVA)
Optimizing hierarchical menus: a usage-based approach

2008-04 Ander de Keijzer (UT)
Management of Uncertain Data - towards unattended integration

2008-05 Bela Mutschler (UT)
Modeling and simulating causal dependencies on process-aware information systems from a cost perspective

2008-06 Arjen Hommersom (RUN)
On the Application of Formal Methods to Clinical Guidelines, an Artificial Intelligence Perspective

2008-07 Peter van Rosmalen (OU)
Supporting the tutor in the design and support of adaptive e-learning

2008-08 Janneke Bolt (UU)
Bayesian Networks: Aspects of Approximate Inference

2008-09 Christof van Nimwegen (UU)
The paradox of the guided user: assistance can be counter-effective

2008-10 Wauter Bosma (UT)
Discourse oriented summarization

2008-11 Vera Kartseva (VU)
Designing Controls for Network Organizations: A Value-Based Approach

2008-12 Jozsef Farkas (RUN)
A Semiotically Oriented Cognitive Model of Knowledge Representation

2008-13 Caterina Carraciolo (UVA)
Topic Driven Access to Scientific Handbooks

2008-14 Arthur van Bunningen (UT)
Context-Aware Querying; Better Answers with Less Effort

2008-15 Martijn van Otterlo (UT)
The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for the Markov Decision Process Framework in First-Order Domains.

2008-16 Henriette van Vugt (VU)
Embodied agents from a user's perspective

2008-17 Martin Op 't Land (TUD)
Applying Architecture and Ontology to the Splitting and Allying of Enterprises

2008-18 Guido de Croon (UM)
Adaptive Active Vision

2008-19 Henning Rode (UT)
From Document to Entity Retrieval: Improving Precision and Performance of Focused Text Search

2008-20 Rex Arendsen (UVA)
Geen bericht, goed bericht. Een onderzoek naar de effecten van de introductie van elektronisch berichtenverkeer met de overheid op de administratieve lasten van bedrijven

2008-21 Krisztian Balog (UVA)
People Search in the Enterprise

2008-22 Henk Koning (UU)
Communication of IT-Architecture

2008-23 Stefan Visscher (UU)
Bayesian network models for the management of ventilator-associated pneumonia

2008-24 Zharko Aleksovski (VU)
Using background knowledge in ontology matching

2008-25 Geert Jonker (UU)
Efficient and Equitable Exchange in Air Traffic Management Plan Repair using Spender-signed Currency

2008-26 Marijn Huijbregts (UT)
Segmentation, Diarization and Speech Transcription: Surprise Data Unraveled

2008-27 Hubert Vogten (OU)
Design and Implementation Strategies for IMS Learning Design

2008-28 Ildiko Flesch (RUN)
On the Use of Independence Relations in Bayesian Networks

2008-29 Dennis Reidsma (UT)
Annotations and Subjective Machines - Of Annotators, Embodied Agents, Users, and Other Humans

2008-30 Wouter van Atteveldt (VU)
Semantic Network Analysis: Techniques for Extracting, Representing and Querying Media Content

2008-31 Loes Braun (UM)
Pro-Active Medical Information Retrieval

2008-32 Trung H. Bui (UT)
Toward Affective Dialogue Management using Partially Observable Markov Decision Processes

2008-33 Frank Terpstra (UVA)
Scientific Workflow Design; theoretical and practical issues

2008-34 Jeroen de Knijf (UU)
Studies in Frequent Tree Mining

2008-35 Ben Torben Nielsen (UvT)
Dendritic morphologies: function shapes structure

2009-01 Rasa Jurgelenaite (RUN)
Symmetric Causal Independence Models

2009-02 Willem Robert van Hage (VU)
Evaluating Ontology-Alignment Techniques

2009-03 Hans Stol (UvT)
A Framework for Evidence-based Policy Making Using IT

2009-04 Josephine Nabukenya (RUN)
Improving the Quality of Organisational Policy Making using Collaboration Engineering

2009-05 Sietse Overbeek (RUN)
Bridging Supply and Demand for Knowledge Intensive Tasks - Based on Knowledge, Cognition, and Quality

2009-06 Muhammad Subianto (UU)
Understanding Classification

2009-07 Ronald Poppe (UT)
Discriminative Vision-Based Recovery and Recognition of Human Motion

2009-08 Volker Nannen (VU)
Evolutionary Agent-Based Policy Analysis in Dynamic Environments

2009-09 Benjamin Kanagwa (RUN)
Design, Discovery and Construction of Service-oriented Systems

2009-10 Jan Wielemaker (UVA)
Logic programming for knowledge-intensive interactive applications

2009-11 Alexander Boer (UVA)
Legal Theory, Sources of Law & the Semantic Web

2009-12 Peter Massuthe (TUE, Humboldt-Universitaet zu Berlin)
Operating Guidelines for Services

2009-13 Steven de Jong (UM)
Fairness in Multi-Agent Systems

2009-14 Maksym Korotkiy (VU)
From Ontology-enabled Services to Service-enabled Ontologies (making ontologies work in e-Science with Onto⇔SOA)

2009-15 Rinke Hoekstra (UVA)
Ontology Representation - Design Patterns and Ontologies that Make Sense

2009-16 Fritz Reul (UvT)
New Architectures in Computer Chess

2009-17 Laurens van der Maaten (UvT)
Feature Extraction from Visual Data