Zur Erlangung des akademischen Grades eines
Doktors der Wirtschaftswissenschaften (Dr. rer. pol.)
von der Fakultaet fuer Wirtschaftswissenschaften
der Universitaet Fridericiana zu Karlsruhe
genehmigte Dissertation.

Methods and Tools for Ontology Evolution

M.Sc. Ljiljana Stojanovic

Referent:
Prof. Dr. Rudi Studer, Universitaet Karlsruhe (TH)
1. Korreferent:
Prof. Dr. Christof Weinhardt, Universitaet Karlsruhe (TH)
2. Korreferent:
Prof. Dr. Asuncion Gomez-Perez, Universidad Politecnica de Madrid

Tag der muendlichen Pruefung: 05. August 2004

To my parents

Acknowledgements

This thesis is the result of my work as a research assistant at the FZI – Research Center for Information Technologies at the University of Karslruhe, Germany. Many people supported me and made the successful completion of this thesis possible.

First of all, I would like to express my sincere thanks to Prof. Dr. Rudi Studer, my supervisor, for his encouragement and support during my research. He created a balanced environment that allowed me the necessary freedom to pursue my ideas, while at the same time offering the right amount of guidance to keep me focused.

I would like to thank Prof. Dr. Christof Weinhardt (University of Karlsruhe) and Prof. Dr. Asuncion Gomez-Perez (Universidad Politécnica de Madrid), who were willing to serve on my dissertation committee as co-referents. Furthermore, I thank Prof. Dr. Wolffried Stucky (University of Karlsruhe) and Prof. Dr. Siegfried Berninghaus (University of Karlsruhe), who served on the examination committee.

A PhD thesis is to some degree the product of a synergetical environment. I had the luck to work in a great team, the knowledge management group at the Research Center for Information Technologies (FZI/WIM) at University of Karlsruhe and at the AIFB Institute at the University of Kalrsruhe. Thanks to the all members for the many interesting discussions. I am especially grateful to those members who read earlier drafts of this work and for the useful comments they provided.

I especially thank Dr. Alexander Maedche who directed my initial ideas and thoughts to the right research area. Without Boris Motik this thesis would not have the same degree of profoundness. Our countless intensive discussions and his always very critical, but extremely useful and constructive comments produced valuable ideas for the overall research contribution of this thesis. He also provided the core technology for making the implementation part of this work possible. Thanks to Dr. Andreas Abecker for providing a very fruitful and stimulating working atmosphere. He deserves special gratitude for giving me opportunities to promote my work and for teaching me how to write project proposals.

My parents and my brothers receive my deepest gratitude and love for complete support throughout my life and especially for always believing in me. Last, but not least, I thank my husband, Nenad for his love, support and understanding that strongly encouraged me and, in the end, made this thesis possible.

August 2004, Karlsruhe

Ljiljana Stojanovic

Abstract

With the rising importance of knowledge interchange, many industrial and academic applications have adopted ontologies as their conceptual backbone. Business dynamics and changes in the operating environment often give rise to continuous changes in application requirements that may be fulfilled only by changing the underlying ontologies. This is especially true for Semantic Web applications, which are based on heterogeneous and highly distributed information resources and therefore need efficient mechanisms to cope with changes in the environment.

In the thesis we defined requirements for an efficient ontology evolution system and introduced a process model that fulfils them. The ontology evolution process (i) enables handling the required ontology changes; (ii) ensures the consistency of the underlying ontology and all dependent artefacts; (iii) supports the user to manage changes more easily; and (iv) offers advice to the user for continual ontology reengineering.

Our primary intention was to enable the customisation of the ontology-evolution process to the current need (i.e. knowledge, preferences) of an ontology engineer. It is achieved by allowing the declarative specification of a request for a change, as well as by providing means (i.e. evolution strategies) for guiding the change resolution.

Another aspect we considered is the applicability of the proposed approach to the Semantic Web. We developed a multi-dimensional approach for the ontology evolution that takes into account the number of evolving ontologies and their physical distribution.

Our approach goes beyond a standard change management process; rather it is a continual improvement process. To improve the usability of an ontology with respect to the needs of end-users, we proposed methods for the discovery of the changes by analysing the end-users' behaviours.

A substantial part of the results from the thesis is a system for realizing ontology evolution, implemented in the KAON ontology engineering framework, as well as several case studies that show benefits of the proposed approach.

Furthermore, the thesis identifies a number of promising areas for future work. Finally, it gives a comprehensive overview of related, similar and subsumed approaches.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Overview

An important characteristic of today's business systems is their ability to adapt themselves efficiently to the changes in their environment, as well as to the changes in their internal structures and processes. The continual reengineering of a business system, i.e. the need to be better and better, is becoming a prerequisite for surviving in the highly changing business world. Although changes encompass several dimensions of a business system (e.g. people, processes, technologies), most of them are reflected on its IT infrastructure. For example, the establishment of a new department in the organisational structure will require the corresponding changes in the enterprise portal, underlying groupware system, skill management system, etc. Therefore, the adaptability of the implemented IT solutions directly defines the efficiency of a business system.

However, building and maintaining long-living applications that will be "open for changes" is still a challenge for the entire software engineering community. Even though there are ongoing attempts to address this problem by providing IT systems with powerful concepts for self-management [59], they focus only on changes caused by malfunctioning of a system. Indeed, most of today's management tasks are performed manually. This can be time-consuming and error prone. Moreover, it requires a growing number of highly skilled personnel, making the maintenance of applications costly.

It is clear that an ad hoc management of changes in applications might work only for particular cases. Moreover, it can scale neither in space nor in time. Therefore, in order to avoid drawbacks in the long run, the change management must be treated in a more systematic way. It is especially important for the applications that are distributed over different systems. Examples of such applications are knowledge management applications that enable integration of various, physically distributed knowledge sources differing in the structure and the level of formality.

In order to avoid unnecessary complexity and possible failures and/or even to ensure the realisation of a request for a change, the *change management should deal with the conceptual model* of such an application. For example, a more efficient retrieval of knowledge items in a knowledge management system requires the establishment of the (hierarchical) relationships between their conceptual descriptions.

Ontologies have recently become a key technology for semantics-driven modelling, especially for the ever-increasing need for knowledge interchange and integration. Many industrial and academic applications have adopted ontologies as their conceptual backbone. The usage of ontologies has several advantages [31], [122], [146]:

- Ontologies facilitate *interoperability between applications* by capturing a shared understanding of a problem domain. They provide comfortable means for explicating implicit design decisions and underlying assumptions at the system building time. This makes it easier to reason about the intended meaning of the information interchanged between two systems.
- Ontologies provide a *formalization* of a shared understanding which makes them *machine-processable*. Machine processibility is the basis for the next generation of the WWW, the so-called *Semantic Web* [9], [31], which is based on using ontologies for enhancing (i.e. annotating) content with formal semantics.
- The explicit representation of the semantics of data through ontologies enables applications to provide a qualitatively *new level of services,* such as verification, justification, gap analysis, etc.

Ontology-based applications are subject to a continual change. Thus, to improve the speed and to reduce costs of their modification, the changes have to be reflected on the underlying ontology. Moreover, as ontologies grow in size, the complexity of change management increases significantly. If the underlying ontology is not up-to-date, then the reliability, accuracy and effectiveness of the system decrease significantly [66]. For example, an obsolete classification of knowledge items in an ontology-based knowledge management system decreases the precision of the knowledge retrieval process. A typical example is the MEDLINE system[1], the largest medical knowledge base available over the Internet, which is based on the MESH medical ontology. In order to stay in line with the state-of-the art in medical research, MESH is frequently updated. However, the ontology is not only extended with new terms (e.g. new diseases and medicaments); rather, the terms are often reclassified according to the latest research results. Therefore, in case the MESH is obsolete, not only that some relevant information will be missing, but also some wrong answers will be delivered.

Since an ontology has to be continually changed, the need for the ontology evolution[2] is inevitable. The task of the ontology evolution is to formally interpret all requests for changes coming from different sources (e.g. users, internal processes, business environment) and to perform them on the ontology and depending artefacts while keeping consistency of all of them. Figure 1 illustrates the role of the ontology evolution in a business system.

Although the importance of such a change management approach is demonstrated in the industrial practice [50], as known to the authors the corresponding methods and tools are still missing. The primary reason has been the immaturity of the large-scale and long-living ontology-based applications in the industry, so that the long run pay-off could not be easily accounted. However, the explosive development in the research and implementation of the second generation of the WWW, the so-called Semantic Web, in the last 5 years, has made the idea of the large-scaled ontology-based applications in a business context real. It has opened hundreds of opportunities to apply ontologies in real world contexts. Furthermore, it has opened a dozen of challenges for the efficient development and the maintenance of ontologies that underlie ontology-based applications, which is researched in a lot of research groups as a part of various research and industry projects. This thesis is one of the results of such efforts to make the Semantic Web real. Particularly, in this thesis, we make the theoretical foundations for a systematic approach for the ontology evolution, and present its concrete realisation in the KAON[3] ontology management system.

[1] http://www.nlm.nih.gov/pubs/factsheets/medline.html

[2] The word "evolution" merely means "change through time". It implies neither a direction, nor, necessarily, improvement, but merely a change.

[3] kaon.semanticweb.org

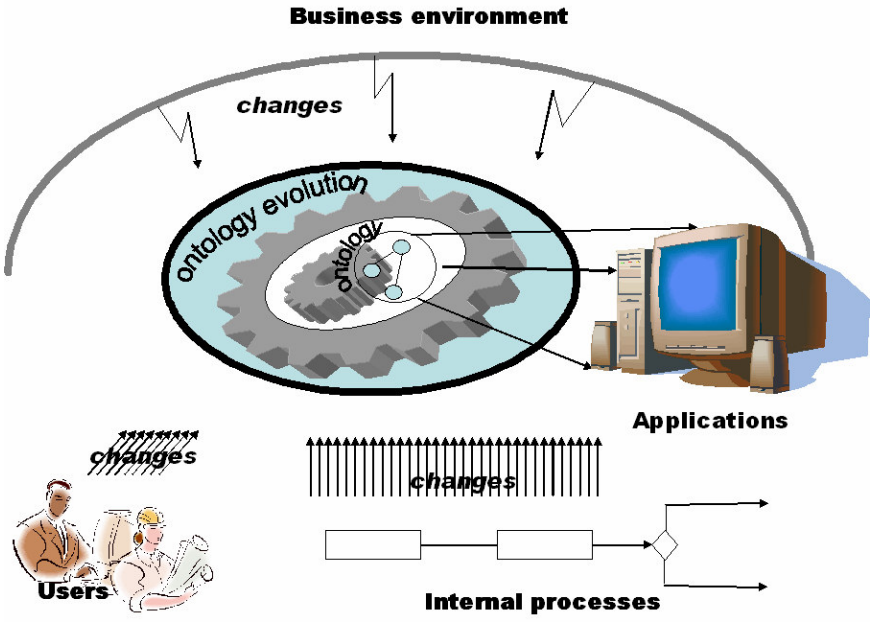


Figure 1. The role of an ontology evolution in a business system

1.2 Ontology Evolution

1.2.1 Definition

Through this thesis, we will use the following definition of the term ontology evolution: "**Ontology Evolution** is the timely adaptation of an ontology to the arisen changes and the consistent propagation of these changes to dependent artefacts." Since a change in the ontology can cause inconsistencies in other parts of the ontology, as well as in the dependent artefacts, the ontology evolution has to be considered as a process. It encompasses the set of activities, both technical and managerial, that ensures that the ontology continues to meet organizational objectives and users' needs in an efficient and effective way.

The distinction between management, modification, evolution and versioning of the ontologies has been, in some cases, confused. In the rest of the thesis, we use the following characterization by adapting the terminology from the database community [113]:

- *Ontology management* is the whole set of methods and techniques that is necessary to efficiently use multiple variants of ontologies from possibly different sources for different tasks. Therefore, an ontology management system should be a framework for creating, modifying, versioning, querying, and storing ontologies. It should allow an application to work with an ontology without worrying about how the ontology is stored and accessed, how queries are processed, etc.;
- *Ontology modification* is accommodated when an ontology management system allows changes to the ontology that is in the use, without considering the consistency;
- *Ontology evolution* is accommodated when an ontology management system facilitates the modification of an ontology by preserving its consistency;

- *Ontology versioning* is accommodated when an ontology system management allows handling of ontology changes by creating and managing different versions of it.

1.2.2 Importance of ontology evolution

Most of the work conducted so far in the field of ontologies has focused on ontology construction issues. It is assumed that domain knowledge encapsulated in an ontology does not evolve in time. However, in a more open and dynamic business environment, the domain knowledge evolves continually [34]. These changes include accounting the modification in the application domain or in the business strategy; incorporating additional functionality according to changes in the users' needs; organizing information in a better way, etc. Figure 1 depicts three basic sources that can cause changes in a business system:

- *The environment*: The environment in which systems operate can change, thereby invalidating assumptions made when the system was built. For example, acquiring a new subsidiary in an enterprise adds new business areas and functionalities to the existing system;
- *Users*[4]: Users' requirements often change after the system has been built, warranting system adaptation. For example, hiring new employees might lead to new competencies and greater diversity in the enterprise, which the system must reflect;
- *Internal processes*: The business applications are coupled around the business processes that should be continually reengineered, in order to achieve better performances.

Therefore, ontology development is a dynamic process starting with an initial rough ontology, which is later revised, refined and filled in with the details [95]. Further, the ontology must be used, and, during its period of use, the knowledge on which it relies will change and develop. An ontology that has not become rapidly obsolete must change and adapt to the changes in environments, users' needs, etc. Therefore, if an ontology aims at being useful, it is essential that it is able to accommodate the changes that will inevitably occur. It this thesis, we address this neglected area of the ontology evolution.

Ontology evolution is very important nowadays. The major reason for this is the increasing number of ontologies in use and the increasing costs associated with adapting them to changing requirements. Developing ontologies and their applications is expensive, but evolving them is even more expensive. The experiences show that the traditional software systems maintenance costs exceed the development costs by a factor of between two and four. There is not a reason to assume this should be any different for ontologies, when they are used during a longer period of time. The costs can be even higher, due to the collaborative development of ontologies and their physical distribution.

1.2.3 Problems in realizing ontology evolution

Ontology evolution is not a trivial process, due to the variety of sources and consequences of changes. It cannot be done manually by an ontology engineer, since she is not able to

[4] We differ between two types of users: ontology engineers who develop an ontology and end-users who use an ontology-based application.

comprehend all side-effects of a change. Therefore, a system that is responsible for maintaining consistency is needed. Building such a system has proven to be a difficult task, since there is almost a complete lack of suitable methodology, techniques and tools.

Particularly, there are three challenges for the efficient realisation of the ontology evolution:

- *Complexity* – an ontology model is rich and, therefore, an ontology has an interwoven structure. Each change leads to a change specific workaround. Even when the effects of a change are minor, the cumulative effect of all changes realizing a user's request can be enormous;
- *Dependencies* - ontologies often reuse and extend other ontologies. Changes in an ontology may affect the ontologies that are based on it. Therefore, changes between dependent ontologies are interrelated, and the immediate synchronisation between dependent ontologies is required. Obviously, the complexity of the ontology evolution increases with the number of dependent ontologies being evolved;
- *Physical distribution* - ontology development is a de-centralized and collaborative process. Therefore, the physical distribution of the dependent ontologies has to be taken into account. The ontology evolution requires tracking the changes applied to an ontology and broadcasting the group of changes when an explicit request arises.

1.3 Contributions

In the thesis, we make several contributions:

1. *Process-oriented ontology evolution*: We define requirements for an efficient ontology evolution system and introduce the process model that fulfils them. The ontology evolution process (i) enables handling the required ontology changes; (ii) ensures the consistency of the underlying ontology and all dependent artefacts; (iii) supports the user to manage changes more easily; and (iv) offers advice to the user for continual ontology reengineering.

2. *User-driven ontology evolution*: Since there are many ways to resolve an ontology change, we aim at allowing ontology engineers to control and customise this process. Evolution strategies are developed as a method of "finding" the consistent ontology that meets the needs of the ontology engineer. Moreover, to allow the resolution of the complex request for a change, the ontology engineer is able to represent her request declaratively, and to choose the way of the resolution that is the most suitable to her need.

3. *Multi-dimensional ontology evolution*: We identify two dimensions of the overall ontology evolution problem. The first dimension defines the number of the ontologies that have to be updated for a change request. The second dimension specifies the physical location of evolved ontologies. We extend the ontology evolution of a single ontology in two ways: (i) we define the solution for evolution problems incurred between dependent ontologies within one node; and (ii) we extend this approach for the distributed environment.

4. *Usage-driven ontology evolution*: To improve the usability of an ontology with respect to the needs of end-users (i.e. the users of the ontology-based applications), we investigate the possibility of the continual ontology improvement. We propose methods for the discovery of the changes by analysing the end-users' behaviours. In order to anticipate the end-users' needs, we define several measures that combine the usage-data and the information about the ontology structure. The interpretation of these measures results in new ontology changes.

5. *Ontology evolution framework*: The proposed ontology evolution approach has been implemented in the KAON framework. Since our goal has been to support the evolution of

the large ontologies on the Semantic Web, the special focus of the implementation was given to the scalability issue. In several case studies realized in different research projects, we evaluated the implemented system.

1.4 Thesis Overview

The remainder of the thesis proceeds as follows.

Chapter 2 provides the background information on ontologies, and defines notions needed for understanding the ontology evolution. We introduce the KAON ontology language that is used through the thesis. Further, we define the ontology consistency and present the taxonomy of ontology changes. Finally, we survey the work from a number of related fields.

In Chapter 3, we define the requirements for the ontology evolution system and derive a six-phase evolution process that satisfies them. The process systematically analyses the causes and the consequences of the changes, and ensures the consistency of the ontology and depending artefacts after resolving these changes. Further, we discuss the ontology changes at different levels of abstractions, and introduce the evolution ontology that is used as a backbone of the process.

Chapter 4 explores the complexity of the change resolution problem and defines two ways to support the user-driven ontology evolution. We introduce a procedural and declarative approach, and discuss its advantages and disadvantages regarding the needs of ontology engineers who use an ontology evolution system.

In Chapter 5, we tackle the problem of the evolution of dependent ontologies. We define two ways for the reuse of ontologies. We extend the single ontology evolution process by taking into account the dependency between ontologies and their physical distribution. The evaluation of the proposed approaches on the MEDLINE dataset is presented, as well.

Chapter 6 is dedicated to the usage-driven ontology evolution. We introduce the usage ontology that is used to capture the end-users' interactions with the ontology-based application. We define several measures combining the frequency of usage of an ontology entity and the structure of the ontology, in order to recommend changes. Two evaluation studies show the usability of the proposed measures.

In Chapter 7, we present the conceptual architecture of the KAON ontology engineering framework, focusing on the aspects relevant for the ontology evolution. We present the application of the ontology evolution system in the EU-IST OntoGov project.

Chapter 8 contains some concluding remarks and the outlook for the future work.

1.5 Publications

Parts of the thesis have been published before:

Chapter 3 that defines a process model for the ontology evolution is based on several publications. The discussion of the requirements for the ontology evolution and the process itself is published in "A. Maedche, B. Motik, L. Stojanovic, R. Studer, R. Volz, *Ontologies for Enterprise Knowledge Management*, IEEE Intelligent System, Volume 18, Number 2, pp. 26-34, March/April 2003".

The phases of this process are further elaborated in:

- ”N. Stojanovic, L. Stojanovic, *Evolution in the ontology-based knowledge management system*, In Proceedings of the Xth European Conference on Information Systems - ECIS 2002, Gdañsk, Poland, 2002”;
- “A. Maedche, L. Stojanovic, R. Studer, R. Volz, *Managing Multiple Ontologies and Ontology Evolution in Ontologging*, In Proceedings of the Conference on Intelligent Information Processing (IIP-2002), Montreal, Canada, pp. 51-63, 2002”;
- ”L. Stojanovic, *An approach for Continual Ontology Improvement*, to appear in Proceedings of the First International Conference on Knowledge Engineering and Decision Support (ICKEDS'2004), Porto, Portugal, 2004”.

Two different ways of exploring the complexity of the “semantics of change” problem in Chapter 4 are described in:

- “L. Stojanovic, A. Maedche, B. Motik, N. Stojanovic, *User-driven Ontology Evolution Management*, In Proceedings of the 13th European Conference on Knowledge Engineering and Knowledge Management (EKAW’02), Siguenza, Spain, LNCS 2473, pp. 285-300, 2002” and
- “L. Stojanovic, A. Maedche, N. Stojanovic, R. Studer, *Ontology Evolution as Reconfiguration-design Problem Solving*, In Proceedings of the international conference on Knowledge capture (K-CAP’03), Sanibel Island, FL, USA, pp. 162-171, 2003”.

Parts of Chapter 5 are partially published in:

- “A. Maedche, B. Motik, L. Stojanovic, *Managing multiple and distributed ontologies on the Semantic Web*, theVLDB Journal (2003) - Special Issue on Semantic Web, Volume 12, pp. 286-302, 2003” and
- “A. Maedche, B. Motik, L. Stojanovic, R. Studer, R. Volz, *An Infrastructure for Searching, Reusing and Evolving Distributed Ontologies*, In Proceedings of the Twelfth International World Wide Web Conference (WWW 2003), Budapest, Hungary, ACM, pp. 439-448, 2003”.

Chapter 6 is dedicated to the usage-driven ontology evolution. It is published in:

- “L. Stojanovic, N. Stojanovic, A. Maedche, *Change Discovery in Ontology-Based Knowledge Management Systems*, In Proceedings of 21st International Conference on Conceptual Modelling (ER’2002), Workshop on Evolution and Change in Data Management (ECDM’02), Tampere, Finland, 2002, Revised Papers, LNCS 2784, ISBN 3-540-20255-2, pp. 51-62, 2003” and
- “L. Stojanovic, N. Stojanovic, J. Gonzalez, R. Studer*, OntoManager - A System for the Usage-Based Ontology Managemen*t, In Proceedings of the 2st International Conference on Ontologies, Databases and Application of Semantics (ODBASE 2003), Catania, Sicily, Italy, LNCS 2888, pp. 858-875, 2003”.

The tools and applications that are presented in Chapter 7 are also described in

- “L. Stojanovic, B. Motik, *Ontology Evolution within Ontology Editors*, In Proceedings of the OntoWeb-SIG3 Workshop Evaluation of Ontology-based Tools (EON2002) at the 13th International Conference on Knowledge Engineering and Knowledge Management (EKAW 2002), Siguenza, Spain, CEUR-WS Volume 62, pp. 53-62, 2002” and

- "L. Stojanovic, J. Schneider, A. Maedche, S. Libischer, R. Studer, T. Lumpp, A. Abecker, G. Breiter, J. Dinger *The Role of Ontologies in Autonomic Computing Systems*, To appear in IBM Systems Journal, Volume 43, Number 3, 2004".

2 Basics of Ontology Evolution

Before speaking about evolving ontologies, several notions have to be clarified. Since the ontology evolution is the process of changing an ontology while maintaining its consistency, in the rest of this chapter we define the notions of ontology (see section 2.1), of its consistency (section 2.3) and its changes (section 2.4). The process itself is elaborated in chapter 3. The approach we present throughout this thesis is based on the KAON ontology model. We briefly review the key concepts of this model in section 2.2.

2.1 Ontology

2.1.1 Definition of Ontology

The term ontology is borrowed from philosophy, where an ontology is a systematic account of existence. For computer science, what "exists" is that which can be represented. Thus, in the context of computer science, the following definition is adopted [48]:

Definition 1 *An ontology* is a *formal*, *explicit* specification of *a shared conceptualisation* of a *domain* of interest.

Conceptualisation is an abstract, simplified view of the world that we wish to represent for some purpose. Ontologies have set out to overcome the problem of implicit and hidden knowledge by making the conceptualisation of a domain explicit. Ontology is used to make assumptions about the meaning of a specific concept. It can also be seen as an explication of the context for which the concept is normally used.

Moreover, everything (i.e., any knowledge-based system or any knowledge-level agent) is liable to some conceptualisation, explicitly or implicitly. Therefore, since there is consensus of terms, it is ***a shared conceptualisation***.

Next, the purpose of an ontology is not to model the whole world, but rather a part of it - so-called domain. A ***domain*** is just a specific subject area or area of knowledge, like medicine, tool manufacturing, real estate, automobile repair, financial management, etc. Therefore, in order to define a domain, it is important to know what an ontology is *for*.

Further, ontologies serve as a means for establishing a conceptually concise basis for communicating knowledge for many purposes. In order to achieve this, an ontology has to be

a ***formal*** description of the meaning of concepts and relationships between them. Therefore, the formal specification means that an ontology is specified by means of a formal language, e.g. first order logic.

Finally, this formal model is readable, understandable and processable not only for the people, but also for the machines. This is achieved through the ***explicit*** specification, while there is formal semantics of all statements, i.e. the semantics of the used language is formally specified as well. Therefore, ontologies have to be specified in a language that comes with formal semantics. Only in this way can the detailed, accurate, consistent, sound, and meaningful description be made.

The study of ontologies and their use is no longer just one of the fields in the computer science literature. Ontologies are now ubiquitous in many enterprise-wide information-systems: they are used in e-commerce, knowledge management and in various application fields such as bioinformatics and medicine. Moreover, they constitute the backbone for the Semantic Web, which is discussed in the next section.

2.1.2 Ontologies on the Semantic Web

The Semantic Web [9], [31] is the next generation of the WWW, which is based on using ontologies for enhancing (i.e. annotating) content with formal semantics. "Expressing meaning" of resources that can be found on the web is the main task of the Semantic Web. In order to achieve that objective, several layers of representational structures are needed [9]. They are presented in Figure 2.

These layers have the following role:

- the XML layer represents the *structure* of data;
- the RDF layer represents the *meaning* of data;
- the Ontology layer represents the *formal common agreement* about meaning of data;
- the Logic layer enables intelligent *reasoning* with meaningful data;
- the Proof layer supports the exchange of "*proofs*" in an inter-agent communication enabling the common understanding of how the desired information is derived;
- the Trust layer ranges from digital signatures and security to social network analysis.

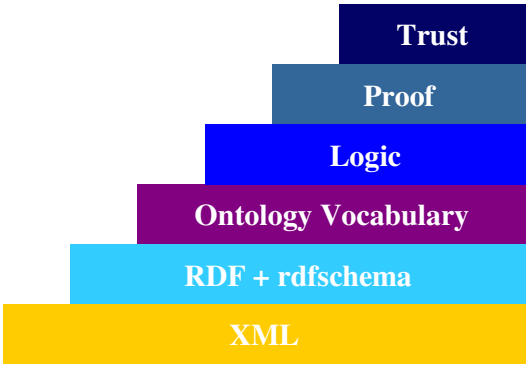


Figure 2. Layers of the Semantic Web Architecture

It is worth noticing that the real power of the Semantic Web is realised when many systems that collect Web content from diverse sources, integrate and process the information as well as exchange the results with other human or machine agents are created. Thereby, the effectiveness of the Semantic Web will increase drastically as more machine-readable Web contents and automated services become available. This level of inter-agent communication will require the exchange of "proofs". Furthermore, the Semantic Web will also be the basis for the Web of Trust, which will provide mechanisms to handle authentication, permission, and validation of attribution in a Web where, by design, anyone can contribute content, links, and services.

Two important technologies for developing the Semantic Web are already in place: the eXtensible Markup Language (XML) and the Resource Description Framework (RDF).

XML[5] lets everyone create his own tags that annotate Web pages or sections of text on a page. The systems can make use of these tags in sophisticated ways, but the programmer has to know the pages, which the author uses each tag for. In short, XML allows users to add arbitrary structure to their documents, but says nothing about what the structures mean. The meaning of XML-documents is intuitively clear to humans since the “semantic” mark-up and tags are domain-terms. However, computers do not have intuition. Tag-names per se do not provide semantics.

Data Type Definitions (DTDs) are a possibility to structure the content of the documents. However, structure and semantics are not always aligned, they can be orthogonal. Therefore, a DTD is not an appropriate formalism to describe the semantics of an XML document [29]. The same holds for XML-Schema[6] – it only defines structure, though with a richer language. In essence, XML lacks a semantic model: it has only a “surface model", a tree. So, XML is not the solution for propagating semantics through the Semantic Web. It can only play the role of a "transport mechanism", viz. as an easily machine-processable data format.

The Resource Description Framework[7] (RDF) provides a means for adding semantics to a document. RDF is an infrastructure that enables encoding, exchange and reuse of structured metadata. Principally, information is stored in the form of RDF statements, which are machine understandable. Search engines, intelligent agents, information brokers, browsers and human users can understand and use that semantic information. RDF is implementation independent and may be serialised in XML (i.e., its syntax is defined in XML). Adding semantic information to web documents is called *semantic annotation*. RDF, in combination with RDFS,[8] offers modelling primitives that can be extended according to the needs at hand. Basic class hierarchies and relations between classes and objects are expressible in RDFS. Some parts of the RDF and RDFS vocabularies are not assigned any formal meaning, and in some cases, notably the reification and container vocabularies, it assigns less meaning than one might expect[9]. Therefore, interpretation of how to use RDF(S) properly is an error-prone process.

A solution to this problem is provided by the third basic component of the Semantic Web, via ontologies. In philosophy, an ontology is a theory about the nature of existence, about what types of things exist; ontology as a discipline studies such theories. Artificial intelligence and web researchers have co-opted the term for their own purposes, and for them an ontology describes a formal, shared conceptualisation of a particular domain of interest, as defined in the previous section.

[5] http://www.w3.org/XML
[6] http://www.w3.org/XML/Schema
[7] http://www.w3.org/RDF/
[8] http://www.w3.org/TR/PR-rdf-schema
[9] http://www.w3.org/TR/rdf-mt/

Ontologies are well suited for describing heterogeneous, distributed and semi-structured information sources (e.g. XML documents) that can be found on the web or in the intranets. By defining shared and common domain theories, ontologies help both people and machines to communicate concisely, by supporting the exchange of semantics, rather than only syntax. It is therefore important that any semantics for the web is based on an explicitly specified ontology. In this way consumer and producer agents can reach a shared understanding by exploiting ontologies that provide the vocabulary needed for negotiation.

The Semantic Web needs ontologies with a significant degree of structure. This structure consists of the following kinds of concepts:

- the concepts (general things) in a domain of interest;
- the relationships that can exist among things;
- the attributes that belong to things;
- the instances that correspond to the concrete individuals in the domain.

The ontologies, which already exist on the Semantic Web, range from simple taxonomies (such as the Yahoo hierarchy), to metadata schemes (such as the Dublin Core), to logical theories [87]. Even though ontologies are often equated with taxonomic hierarchies of concepts, ontologies need not be limited to these forms. Besides, ontologies are not limited to the definitions in the traditional logic sense that only introduce terminology and do not add any knowledge about the world. To specify a conceptualisation one needs to state axioms that do constrain the possible interpretations of the defined terms.

2.1.3 Ontology Languages

To be useful, ontologies must be expressed in a concrete notation. An *ontology language* is a formal language by which an ontology is built. There have been a number of languages for ontologies ([19], [33], [80], [85]) both proprietary and standards-based. Based on their formal semantics they can be split into two groups of languages [51]:

- ***Frame-based ontology languages*** – They have a long history in artificial intelligence. Their central modelling primitives are classes (known as frames) with properties (known as slots). A frame provides a context for modelling a class, which is generally defined as a subclass of one or more other classes, with slot-value pairs being used to specify additional constraints on instances of the new class. Many frame-based systems and languages with many additional refinements of these modelling primitives have been developed [26]. Moreover, adapted to the object-oriented paradigm they have been very successfully applied in the software engineering. For example, the KAON ontology language [80], which is used in this thesis, incorporates the essential modelling primitives of frame-based systems, being based on the notion of a concept and the definition of its superclasses and slots. It also treats slots as first class objects that can have their own properties (e.g. domain and range) and can be arranged in a hierarchy;
- ***Description logic based ontology languages*** – They have been developed in knowledge-representation research, and describe knowledge in terms of concepts (comparable to classes, or frames) and roles (comparable to slots in frame systems). An important aspect of these languages is that they have very well understood theoretical properties. Description logic enables reasoning with concept descriptions and the automatic derivation of classification taxonomies. There are now efficient

implementations of description logic reasoners able to perform these tasks. For example, the Ontology Web Language - OWL [103] inherits from description logic both their formal semantics and efficient reasoning support.

In the rest of this thesis we focus on the KAON ontology language.

2.2 KAON Ontology Language Definition

When KAON[10] development started, RDF and RDFS were the de-facto standard languages for ontology modelling in the Semantic Web [31]. Hence, these languages were chosen to be implemented by the platform. However, as development progressed, certain features of these languages were found to be inadequate for practice. Also, the languages in question have undergone a transformation themselves. Hence, the currently implemented ontology language is based on RDF(S), but contains many additions and changes to the standard [80]. To avoid the pitfalls of self-describing RDFS primitives[11] such as subClassOf, the KAON ontology language has the clean separation of modelling primitives from the ontology itself. Moreover, it provides means for modelling metaclasses and incorporating several commonly used modelling primitives, such as transitive, symmetric and inverse properties, or cardinalities. These differences are justified by practical requirements and thus should be taken into account when designing new versions of the standards. In the rest of this section we recapitulate these differences and point out the lessons learned.

According to the KAON ontology language, all information is organised in so-called OI-models[12] (ontology-instance models), containing both ontology entities (concepts and properties) as well as their instances. This allows grouping concepts with their well-known instances into self-contained units. An OI -model may include another OI-model, thus making all definitions from the included OI-model automatically available. The mathematical definition of OI-model is given below.

Definition 2 *An OI-model* (*ontology-instance model*) is a tuple *OIM := (E, INC)*, where:

- *E* is the set of entities of the OI-model;
- *INC* is the set of included OI-models.

An OI-model represents a self-contained unit of structured information that may be reused. It consists of entities (the set *E* in previous definition) and may include other OI-models (represented through the set *INC*). Different OI-models may talk about the same entity, so the set of entities of these models need not to be disjoint.

Note that the set of ontology entities E contains the ontology entities and an instance pool associated with it. Both of them are defined in the rest of this section.

[10] http://kaon.semanticweb.org

[11] RDF does not make a distinction among normal resources representing instances and resources used as modeling primitives. Thus, a resource can be used as an instance and can have user-defined meta-data. At the same time, a resource can be used also as a modeling primitive. Such situations in practice usually give models with ambiguous semantics.

[12] In the remainder of this thesis we use the terms OI-model and ontology interchangeable.

Definition 3 *An ontology structure* of an OI-model is an 11-tuple:

$$O(OIM) := (C, P, S, T, INV, H_C, H_P, domain, range, mincard, maxcard)$$

where:

- $C \subseteq E$ is a set of concepts;
- $P \subseteq E$ is a set of properties;
- $R \subseteq P$ is a set of relational properties, i.e. relations;
- $A = P \setminus R$ is a set of attribute properties, i.e. attributes;
- $S \subseteq R$ is a subset of symmetric properties;
- $T \subseteq R$ is a subset of transitive properties;
- $INV \subseteq R \times R$ is a symmetric relation that relates inverse properties:

  if $(p_1, p_2) \in INV$, then p_1 is an inverse property of p_2;
- $H_C \subseteq C \times C$ is an acyclic relation called concept hierarchy:

  If $(c_1, c_2) \in H_C$ then c_1 is subconcept (or child) of c_2, c_2 is superconcept (or parent) of c_1, H_C^* is the reflexive, antisymmetric and transitive closure of H_C;
- $H_P \subseteq P \times P$ is an acyclic relation called property hierarchy:

  If $(p_1, p_2) \in H_P$ then p_1 is subproperty (or child property) of p_2, p_2 is a superproperty (or parent property) of p_1, H_P^* is the reflexive, antisymmetric and transitive closure of H_P;
- function $domain: P \rightarrow 2^C$ gives the set of domain concepts for some property $p \in P$;
- function $range: R \rightarrow 2^C$ gives the set of range concepts for some property $p \in R$;
- function $mincard: C \times P \rightarrow N_0$ gives the minimum cardinality for each concept-property pair;
- function $maxcard: C \times P \rightarrow (N_0 \cup \{\infty\})$ gives the maximum cardinality for each concept-property pair.

An ontology example, which will be used in the rest of this thesis, is shown in Figure 3. The ontology models the university domain. It contains the set of concepts such as "*Professor*", "*Student*", "*Project*", and a set of properties between them (e.g. "*hasFirstName*", "*includes*", etc.). Note that concepts are interpreted as sets of elements whereas properties establish the relations between these elements. Each property may have domain concepts as well as range concepts. For example, the domain concept for the property "*includes*" is the concept "*Project*" whereas the range concept is the concept "*Person*". Property domain and property range both constrain the types of instances to which the properties may be applied. Note that the properties are first class citizens. This means that a property can exist without being attached to any concept.

Some properties may be marked as symmetric and/or transitive, and it is possible to say that two properties are inverse of each other. For each concept-property pair, it is possible to specify the minimum (mincard) and maximum (maxcard) cardinalities, defining how many times a property may be specified for instances of that concept. Concepts and properties can be arranged in a hierarchy, as specified by *Hc* and *Hp* respectively. This relation relates directly connected concepts (properties) whereas Hc^* (Hp^*) is the transitive closure of *Hc* (*Hp*). In the example shown in Figure 3, the concept "*PhD Student*" is a direct child of the

concept "*Student*", and an indirect child of the concept "*Person*". Therefore, *("PhDStudent","Student")*$\in H_C$ and *("PhD Student","Person")*$\in H_C^*$.

Definition 4 *An instance pool* associated with an OI-model is a 4-tuple:

$$IP(OIM) := (I, L, instconc, instprop)$$

where:

- $I \subseteq E$ is a set of instances;
- L is a set of literal values, $L \cap E = \varnothing$;
- function *instconc:* $C \rightarrow 2^I$ relates a concept with a set of its instances;
- function *instprop:* $P \times I \rightarrow 2^{I \cup L}$ assigns to each property-instance pair a set of instances related through given property.

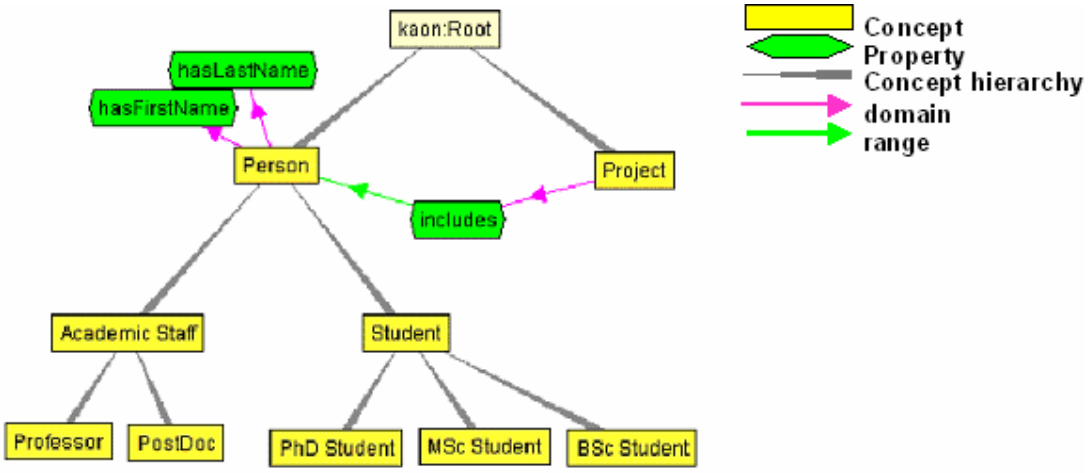


Figure 3. An ontology example

Figure 4 shows the extension of the ontology shown in Figure 3 with an instance pool. It is constructed by specifying the instances, which are instantiated by each concept, and by establishing property instantiation between instances. Property instantiations must follow the domain and range constraints, and must obey the cardinality constraints. For example, "*SteffenWezler*" is an instance of the concept "*BSc Student*", "*OntoLogging*" is an instance of the concept "*Project*", and there is a property instance that relates these two instances through the property "*includes*" i.e. "*SteffenWezler*"$\in$*instprop(*"*includes*", "*OntoLogging*"*)*. This information can be also represented as a triplet: *("OntoLogging","includes","SteffenWezler").*

An OI-model represents a self-contained unit of structured information that may be reused. It consists of entities and may include a set of other OI-models (represented through the set *INC*, see Definition 2). The entities defined in one OI-model are inherited in all OI-models that include it. Different OI-models can talk about the same entity, so the sets of entities E of these OI-models do not need to be disjoint. Therefore, the KAON ontologies build on or extend other ontologies, forming a graph of dependent ontologies.

Definition 5 *A root OI-model* is defined as a particular, well-known OI-model with the structure *ROIM:=({KAON:ROOT}, ∅)*, where *KAON:ROOT* is the root concept and every other concept must be either directly or indirectly a subconcept of the *KAON:ROOT*.

Every other ontology must include ROIM and thus gain visibility to the root concept. Many knowledge representation languages contain the Top concept that is a superconcept of all other concepts.

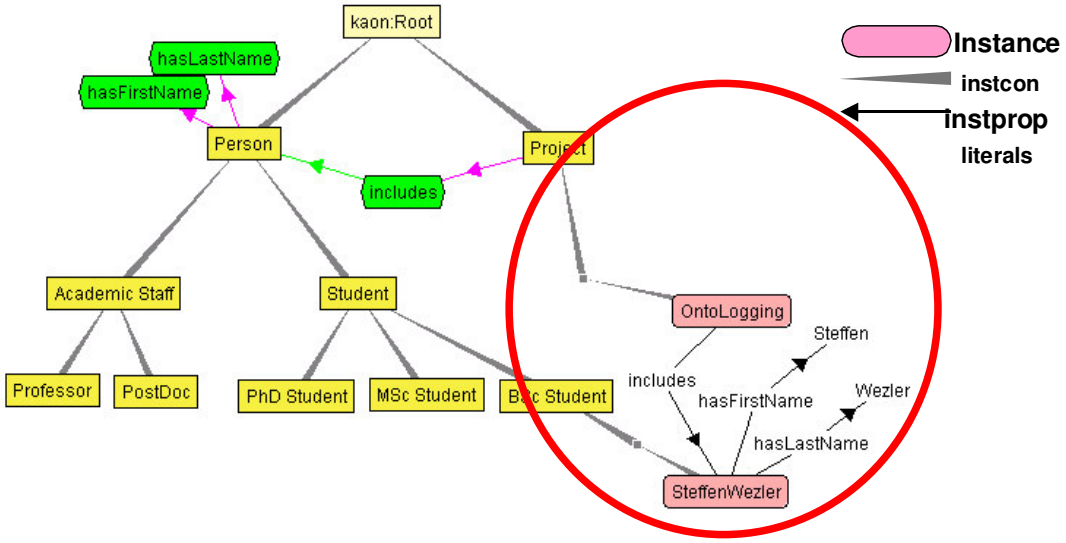


Figure 4. The ontology from Figure 3 and an instance pool associated with it

An example of multiple OI-models is shown in Figure 5. The basic ontology *BO* (see Figure 3 and Figure 4) modelling the university domain is open to be extended in other ontologies. To create an ontology containing information about project, one should be able to reuse as many definitions as possible from existing ontologies. Thus, the project ontology *PO* reuses all definition from the *BO* ontology. Moreover, it contains its own entities. For example, both the subconcepts of the concept "*Project*", namely the concepts "*Research Project*" and "*Industrial Project*", are defined in the *PO* ontology.

It is worth mentioning that the KAON ontology language is defined in a way to adjust the expressiveness of traditional logic-based languages and to sustain tractability. As a side effect, this enables the realisation of ontology-based systems using existing and well-established technologies such as relational databases.

The rest of this thesis is based on the KAON model. However, we emphasise that the proposed approach is applicable to most of the existing ontology models. This is because our model incorporates all the basic entities of the ontology models for which there is a wide acceptance [150]. Due to differences in ontology models, in the rest of this thesis we concentrate on the "common" features of ontology models, namely concepts, properties, instances, as well as concept inheritance. Beside the number of primitives of an ontology model, the other difference lies in the diverse semantics of ontology models. This is mostly reflected in the ontology consistency definition (see section 2.3).

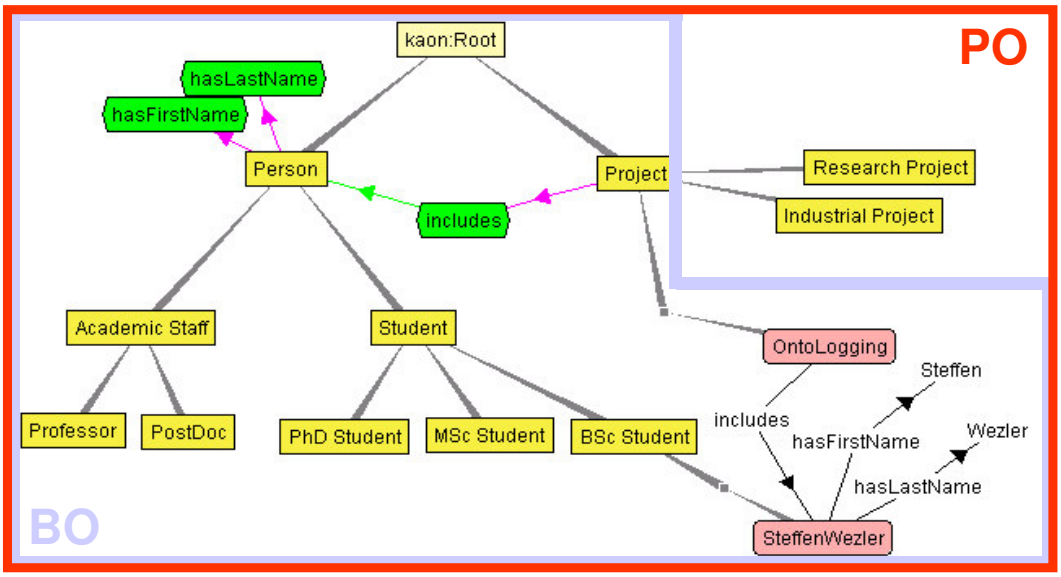


Figure 5. An example of multiple ontologies. The PO ontology reuses the BO ontology

2.2.1 KAON vs. OWL

In this subsection we discuss the distinction between our ontology model and the OWL which should be a standard ontology language. We focus only on the features which are relevant from the ontology evolution point of view. The most important aspects are (i) semantics of domains and ranges and (ii) cardinalities. Both of them are in the KAON interpreted as constraints, and not as inference rules like in the OWL.

Domains and Ranges

The KAON definition of domains and ranges differs from that of RDFS and OWL. In these languages, domain and range specifications are axioms specifying sufficient conditions for an instance to be a member of some class. For example, although for an instance "*instanceI*" is not explicitly stated to be an instance of a concept "*conceptC*", because it has a property "*propertyP*" instantiated and because "*propertyP*" has the concept "*conceptC*" as domain, it can be inferred that the instance "*instanceI*" is an instance of the concept "*conceptC*".

Regarding our experience, while such inferencing may sometimes be truly useful, it is not often needed or even desired in closed environments, such as e.g. presented by most knowledge management applications. Most users without a formal background in logic, but with strong background in databases and object-oriented systems intuitively expect domains and ranges to specify the constraints that must be fulfilled while populating ontology instances. In other words, unless "*instanceI*" is known to be an instance of the concept "*conceptC*", the "*propertyP*" cannot be instantiated for the instance "*instanceI*" in the first place or the ontology becomes inconsistent.

This approach has the following benefits:

- Treating domains and ranges as constraints makes it possible to guide the user in the process of providing information about instances. It is easy to compute the set of

properties that can be applied to an instance, and then to ask the user to provide values for them. On the other hand, if domains and ranges are treated as axioms, any property can be applied to any instance, which makes it difficult to constrain user's input;

- Similar problems occur when evolving the ontology. For example, if the instance "*SteffenWezler*" is removed from the extension of the concept "*BSc Student*", it can be computed that the "*includes*" property between "*OntoLogging*" and "*SteffenWezler*" instances must be removed. On the other hand, if domains and ranges are axioms, then it is not clear how to change the ontology so that it still makes sense;
- Treating domains and ranges as axioms introduces significant performance overhead in query answering. For example, to compute the extension of some concept, it is not sufficient to take the union of the extension of all subconcepts -- one must examine a larger part of the instance pool to see which instances may be classified under the concept according to the domain and range axioms. Therefore, if only the constraint semantics is needed, the system will suffer from unnecessary performance overhead.

Cardinalities

In the KAON ontology language cardinalities are treated as constraints regulating the number of property instances that may be specified for instances of each concept. This is different from the OWL and other description logic languages, where cardinalities are axioms specifying that for the instances with particular number of property instances can be inferred to be instances of some concept. We find that constraining the number of property instances that are allowed for some instance is extremely useful for guiding the user in providing ontology instances. By knowing how many property instances can be provided for instances of some concept, the user can be asked to provide the appropriate number of values. Similar arguments as in the case of domain and range semantics apply here as well.

2.3 Ontology Consistency Model

According to the [56], consistency is the degree of uniformity, standardisation, and freedom from contradiction among the parts of a system or component. From the point of view of logic, consistency is an attribute of a (logical) system that is so constituted that none of the facts deducible from the model contradict one other. Therefore, the ontology consistency can be considered as an agreement among ontology entities with the respect to the semantic of the underlying ontology language.

In this section we define the ontology consistency constraints related to the single ontology[13]. The fact that an ontology might include other ontologies puts additional constraints on the ontology model. These constraints are defined in section 5.2.1 that is dedicated to the evolution of multiple ontologies.

Definition 6 A single ontology *OI* is defined to be *consistent* with the respect to its model if and only if it preserves the constraints defined for underlying ontology model.

Since the ontology consistency is defined for a particular ontology model, the set of constraints heavily depends on the semantics of the underlying ontology model. Based on the

[13] A single ontology *OI* is an ontology that includes only the root ontology *ROIM* and does not include other ontologies, i.e. *OI:=(E, {ROIM})* (see Definition 5).

semantics of the KAON ontology language, we define the ontology consistency model M as:

$$M = IC \cup SC \cup UC,$$
$$IC=\{IC_i\},\ 1\leq i\leq 16,\ SC\subseteq\{SC_j\},\ 1\leq j\leq 2,\ UC\subseteq\{UC_k\},\ 1\leq k\leq 4,$$

where IC_i are *invariants* of the model, SC_j are *soft-constraints* and UC_k are *user-defined constraints*. We note that although an ontology must satisfy all invariants, it does not have to satisfy either all soft or all user-defined constraints The methods enforcing the ontology consistency constraints are elaborated in chapter 4.

Invariants are consistency rules that must hold for every ontology [3]. Every change in an ontology must maintain the correctness of the invariants. The following invariants are defined for our ontology model:

- IC_1: *Distinct Identity Invariant*: All entities (concepts, properties and instances) have a distinct identity:

$$C\cap P=\phi \ \wedge\ C\cap I=\phi \ \wedge\ P\cap I=\phi$$

This consistency constraint enforces a strict separation of concepts, properties and instances, i.e. disjointness of concepts, properties and instances is required. This means that, for example, a concept cannot be at the same time an instance.

- IC_2: *Concept Hierarchy Invariant*: The concept hierarchy is a directed acyclic graph:

$$\neg\exists\, c\in C\ (c,\ c)\in Hc^{*}$$

Returning to the example in Figure 3, it is not allowed to make the concept "*PhD Student*" a superconcept of the concept "*Person*" since the concept "*Person*" is an indirect parent (through the concept "*Student*") of the concept "*PhD Student*".

Note that this constraint does not say anything about multiple inheritance, i.e. in our ontology model multiple inheritance is allowed.

- IC_3: *Rootedness Invariant*: There is a single concept $Root\in C$ that is the superconcept of all concepts in C. This concept *Root* is called the root of the ontology:

$$\exists Root\in C\ (\forall c_1\in C\backslash\{Root\}\ (c_1,Root)\in Hc^{*})\wedge(\neg\exists c_2\in C\ (Root,\ c_2)\in Hc^{*})$$

All concepts are either directly or indirectly subconcepts of the "*Root*" concept that must be included in each ontology. The deletion of the concept "*Root*" in Figure 3 would cause the invalidation of the rootedness invariant.

- IC_4: *Concept-Closure Invariant*: Every concept in C, excluding the root of the ontology, has at least one superconcept in the set C, giving closure to C:

$$\forall c_1\in C\backslash\{Root\}\ \ \exists c_2\in C\ \ (c_1,c_2)\in Hc$$

This constraint prevents the existence of the orphaned concepts, i.e. concepts that do not have any parent concept. For example, the removal of the subconcept relationship between the concept "*Person*" and the concept "*Root*" (see Figure 3) would cause no parent concept to be defined for the concept "*Person*" any longer, which has to prevented or resolved.

- IC_5: *Concept-Hierarchy Closure Invariant*: The parent concepts as well as the child ones have to be elements of the set C:

$$\forall c_1\ \forall c_2\ (c_1,c_2)\in Hc \Rightarrow\ c_1,c_2\in C$$

This constraint requires that the concept hierarchy can be established only between concepts. For example, the addition of the subconcept relationship between concepts "*Project*" and "*Industrial Project*" would also provoke an inconsistency since the "*Industrial Project*" concept is not yet defined as a member of conceptual set *C*.

- IC_6: *Property-Closure Invariant*: The domain relationship can be established between a concept and a property. The range relationhip can be established between a concept and a relation:

$$\forall p \ \forall c \ c \in domain(p) \Rightarrow c \in C \wedge p \in P$$

$$\forall p \ \forall c \ c \in range(p) \Rightarrow c \in C \wedge p \in R$$

For example, the addition of the domain relationship between the concept "*Person*" and the entity "*worksIn*" would provoke inconsistency, in the case that the entity "*worksIn*" is not defined as a property.

- IC_7: *Attribute Invariant*: An attribute must not have a range concept:

$$\forall p \in P \setminus R \ \neg \exists c \in C \ c \in range(p)$$

- IC_8: *Property-Hierarchy Closure Invariant*: The parent properties as well as the child ones have to be elements of the set *P*:

$$\forall p_1 \ \forall p_2 \ (p_1,p_2) \in Hp \Rightarrow p_1,p_2 \in P$$

- IC_9: *Sub-Property Closure Invariant*: A subproperty of some property may add additional domain and range restrictions, but cannot remove existing ones:

$$\forall p_1 \ \forall p_2 \ (p_1,p_2) \in Hp \Rightarrow domain(p_2) \subseteq domain(p_1)$$

$$\forall p_1 \ \forall p_2 \ (p_1,p_2) \in Hp \Rightarrow range(p_2) \subseteq range(p_1)$$

- IC_{10}: *Instance Invariant*: Every instance in *I* is associated to a concept in the set *C*:

$$\forall i \in I \ \exists c \in C \ i \in instconc(c)$$

This constraint prevents the existence of the orphaned instances, i.e. instances that are not attached to any concept. Returning to the example shown in Figure 3, the removal of the instanceOf relationship between the concept "*BSc Student*" and the instance "*SteffenWezler*" would cause the instance "*SteffenWezler*" not to belong to any concept, which has to be either prevented or resolved.

- IC_{11}: *Instance-Closure Invariant*: Every instance is defined as an instance of a concept from the set *C*. Property instantiations must follow the property and instance constraints:

$$\forall c,i \ \ i \in instconc(c) \Rightarrow i \in I \wedge c \in C$$

$$\forall i_1,i_2,p \ i_2 \in instprop(p,i_1) \Rightarrow i_1 \in I \wedge \ i_2 \in I \cup L \wedge p \in P$$

For example, it is not possible to define the concept "*Student*" as an instance of the concept "*Person*" and the concept "*Student*" since the instanceOf relationship can be established only between instances and concepts.

- IC_{12}: *Full Inheritance Invariant*: Only properties defined for a concept associated to an instance or for the parent concepts of that concept can be instantiated for the instance:

$$\forall i_1 \in I \ \forall i_2 \in I \cup L \ \forall p \in P \ i_2 \in instprop(p,i_1) \Rightarrow$$

$$\exists c_1 \in C \ i_1 \in instconc(c_1) \wedge (c_1 \in domain(p) \vee$$

$$(\exists c_2 \in C \ (c_1,c_2) \in Hc^* \wedge (c_2 \in domain(p)))$$

$$\forall i_1 \in I \ \forall i_2 \in I \cup L \ \forall p \in P \ i_2 \in instprop(p,i_1) \Rightarrow$$

$$(\exists c_1 \in C \ i_2 \in instconc(c_1) \ \wedge (c_1 \in range(p) \vee$$

$$(\exists c_2 \in C \ (c_1,c_2) \in Hc^* \wedge (c_2 \in range(p)))) \vee \ (i_2 \in L)$$

- IC_{13}: *Property Instance Invariant*: A property instantiated between instances must have a range concept. A property instantiated between an instance and a literal value must not have a range concept.

$$\forall p \in P \ \forall i_1 \in I \ \forall i_2 \in I \ i_2 \in instprop(p,i_1) \Rightarrow \exists c \in C \ c \in range(p)$$

$$\forall p \in P \ \forall i_1 \in I \ \forall l \in L \ l \in instprop(p,i_1) \Rightarrow \neg \exists c \in C \ c \in range(p)$$

- IC_{14}: *Cardinality Invariant*: Minimal cardinality has to be less than maximal cardinality:

$$\forall c \in C \ \forall p \in P \ 0 \leq mincard(c,p) \leq maxcard(c,p)$$

- IC_{15}: *Cardinality Closure Invariant*: Minimal and maximal cardinality must be specified for concept-property pairs:

$$\forall c,p \ mincard(c,p) \Rightarrow \ c \in C \ \wedge p \in P$$

$$\forall c,p \ maxcard(c,p) \Rightarrow \ c \in C \ \wedge p \in P$$

- IC_{16}: *Property Specification Invariant*: Property axioms must be defined for relations:

$$\forall p \ \ p \in S \Rightarrow p \in R$$

$$\forall p \ \ p \in T \Rightarrow p \in R$$

$$\forall p_1, p_2 \ \ (p_1, p_2) \in INV \Rightarrow p_1, p_2 \in R$$

This implies that the three property characteristics (i.e. symmetric, transitive and inverse of) can never be specified for attributes.

We note that the set of invariants contains the set of consistency constraints (e.g. IC_5) that are directly derived from the KAON ontology model definition (see section 2.2). They represent necessary but not sufficient conditions for establishing well-formed KAON ontologies. Therefore, we define the additional consistency constraints (e.g. IC_4) that put further restrictions on the KAON ontologies, i.e. they restrict the use of ontology entities.

The previously introduced invariants are "hard" constraints since they must be satisfied in any stable state of the ontology; that is before and after an ontology change. However, there are so-called *soft constraints* that can be temporarily invalidated in order to make the ontology evolution easier. We have introduced the following set of soft constraints:

- *SC_1:Min-cardinality Soft Constraint*:

$$\forall c \in C \ \forall p \in P \ mincard(c,p) \text{ is defined} \Rightarrow$$

$$\forall i \in I \ |instprop(p,i)| \geq mincard(c,p)$$

- *SC_2: Max-cardinality Soft Constraint*:

$$\forall c \in C \ \forall p \in P \ maxcard(c,p) \text{ is defined} \Rightarrow$$

$$\forall i \in I \ |instprop(p,i)| \leq maxcard(c,p)$$

To explain the importance of soft constraints, let's consider that minimal cardinality 2 is defined for the property "*includes*" and the concept "*Project*" that are shown in Figure 3. An ontology engineer cannot add any instance of the concept "*Project*" or of its subconcepts since the SC_1 can never be satisfied due to the fact that this ontology contains only one instance "*SteffenWezler*" of the concept "*Person*". The higher minimal cardinality implies fewer chances to realise the request of an ontology engineer. Thus, we argue that it is better to allow that SC_1 and SC_2 constraints are temporarily unsatisfied, but to inform the ontology engineer about this inconsistency (e.g. by using a different colour to mark this part of the ontology).

In addition to the invariants and soft constraints, we have introduced the so-called *user-defined constraint*s. They represent guidelines for building well-formed ontologies. The following set of the user-defined constraints is defined:

- *UC_1: Domain/Range Property User-defined Constraint*: A property with a domain/range concept is considered as consistent:

$$\forall p \in P \ \exists c \in C \ \ c \in domain(p)$$

$$\forall p \in P \ \exists c \in C \cup L \ \ c \in range(p)$$

At least one domain concept has to be specified for each property. Furthermore, a property has to be defined as an attribute (i.e. its range is a literal) or a property has to have at least one range concept. Returning to the example shown in Figure 3, the removal of the concept "*Project*" which is the only element of the domain set for the property "*includes*" results in such an inconsistency.

- *UC_2: Domain/Range Property Repetition User-defined Constraint*: A domain (range) of a property cannot contain a concept that is at the same time a subconcept of some other domain (range) concept:

$$\forall p \in P \ \forall c_1 \in C \ c_1 \in domain(p) \ \Rightarrow \neg (\exists c_2 \in C \ (c_2,c_1) \in Hc^* \wedge c_2 \in domain(p))$$

$$\forall p \in P \ \forall c_1 \in C \ \ c_1 \in range(p) \Rightarrow \neg (\exists c_2 \in C \ (c_2,c_1) \in Hc^* \wedge c_2 \in range(p))$$

For the ontology shown in Figure 3, the concept "*Student*" cannot be a range concept of the property "*includes*" since it is a subconcept of the concept "*Person*" and inherits all properties from its parents including the property "includes".

- *UC_3: Concept Hierarchy Shape User-defined Constraint*: It is not allowed to have an alternative path to the direct parent concept:

$$\forall c_1,c_2 \in C \ (c_1,c_2) \in Hc \Rightarrow \neg (\exists c_3 \in C \ (c_1,c_3) \in Hc^* \wedge (c_3,c_2) \in Hc^*)$$

Returning to the ontology shown in Figure 3, the concept "*PhD Student*" cannot be a direct child of the concept "*Person*" since there is a path between them through the concept "*Student*".

- *UC_4: One Leaf Concept User-defined Constraint*: A concept cannot contain a single subconcept:

$$\forall c_1,c_2 \in C \ (c_1,c_2) \in Hc \Rightarrow \exists c_3 \in C \setminus \{c_1\} \ (c_3,c_2) \in Hc$$

By assuming that the concept "*PostDoc*" in Figure 3 is removed, the concept "*Professor*" would be superfluous since classification with only one subclass beats the original purpose of classification. Our experience shows that this constraint is very useful for improvement of an existing ontology due to the fact that it points to the "weak" place in the ontology. However, it should not be used for the ontology development since one leaf concept can be considered as a point for the future extensions.

2.4 Ontology Changes

There are two major issues involved in the ontology evolution. The first issue is the understanding how an ontology can be changed since the ontology evolution is realised by means of applying ontology changes. The second issue involves deciding when and how to modify an ontology to keep its consistency, which is elaborated in chapter 4.

To resolve the first issue a possible set of changes has to be defined. This set of ontology changes heavily depends on the underlying ontology model and thus it varies from one ontology system to another. Thereupon, one of the fundamental issues concerning the introduction of changes in a data model is the semantics of a change. It refers to the effects of the change on the ontology itself, and, in particular the checking and maintenance of the ontology consistency after the change application. Only in this way can the correct use of ontology changes be completely independent of the ontology engineer's responsibility, with automatic system aid and control.

In the rest of this section we first establish the taxonomy of changes under the KAON ontology model (see section 2.4.1). Then in section 2.4.2 we define the semantics of these changes, which is a prerequisite for preserving the ontology consistency.

2.4.1 Taxonomy of Ontology Changes

One of the first approaches for the object-oriented schema evolution was proposed in [3]. The given taxonomy of changes was adjusted in most other schema evolution research and represents the most frequently used set of changes. In this section we adopt this approach by taking into account the underlying ontology model. Namely, this taxonomy does not consider all aspects of the ontology model. For example, the characteristics that can be defined for the properties (e.g. symmetry) are specific for the ontology model and the changes related to them are not defined in [3]. Further, changes related to the property hierarchy, cardinality constraints, included ontology models, etc., are specific for ontology models in general. Therefore, we extend the taxonomy with the corresponding changes. On the other hand, we use only a subset of this taxonomy by excluding changes related to the methods since they are not a part of the ontology model.

Through the enrichment of the referent taxonomy [3] with the additional changes related to the ontology model and through the elimination of changes that are not relevant to the ontology model, the "standard" set of schema changes is completely adjusted to the KAON

ontology model. Moreover, the semantics of all changes (including the changes existing in the referent taxonomy) is defined on the basis of the underlying ontology model. For example, a property is a first-class citizen and therefore, it can exist without being attached to any concepts.

Each entity of the ontology model can be updated by one of the meta-change transformations: add and remove [55]. A full set of changes can thus be defined by the cross product of the set of entities of the ontology model, which form meta schema, and the set of meta-changes. A complete set of changes, determined by the KAON ontology language (see section 2.2), is given in Table 1. These changes represent the ontology modifications at the lowest level of the complexity since they can add or remove *one* and *only one* entity in an ontology. Therefore, they build the backbone of an ontology evolution system.

Table 1. The taxonomy of ontology changes

Meta Entities/Meta Changes	Add	Remove
Concept	AddConcept	RemoveConcept
Concept Hierarchy	AddSubConcept	RemoveSubConcept
Property	AddProperty	RemoveProperty
Property Hierarchy	AddSubProperty	RemoveSubProperty
Property Domain	AddPropertyDomain	RemovePropertyDomain
Property Range	AddPropertyRange	RemovePropertyRange
Property Symmetric	AddPropertySymmetric	RemovePropertySymmetric
Property Transitive	AddPropertyTransitive	RemovePropertyTransitive
Property Inverse	AddPropertyInverse	RemovePropertyInverse
Max Cardinality	AddMaxCardinality	RemoveMaxCardinality
Min Cardinality	AddMinCardinality	RemoveMinCardinality
Instance	AddInstance	RemoveInstance
InstanceOf	AddInstanceOf	RemoveInstanceOf
InstProp	AddPropertyInstance	RemovePropertyInstance
OI-model	AddOI-model	RemoveOI-model

Table 1 shows that ontology changes can be thought of as *additive* and *subtractive* as is illustrated through the second and the third column, respectively. The formal distinction of these two types of changes, adopted from [114], is given below.

Definition 7 *An ontology change OntoCh* is a total mapping[14] between ontologies, i.e. $OIM_2=OntoCh(OIM_1)$, where OIM_1 and OIM_2 are ontologies.

We introduce the following function:

[14] A mapping *f* between two sets *A* and *B*, $f:A \rightarrow B$, is *total* if *f* is defined on every element of *A*.

- the function *extractSet(OIM, i)* that gives *i-th* set of elements for a given ontology *OIM*, 1≤i≤15.

There are 15 meta entities in the KAON ontology model. They are derived from the KAON ontology language definition (see Definition 2, Definition 3 and Definition 4) and are shown in Table 1. For example, *extractSet(OIM,1)* returns a set of concepts *C* or *extractSet(OIM,15)* returns a set of included ontologies *INC*.

Definition 8 An ontology change *OntoCh* is an *additive* (*capacity augmenting*) ontology change, if *OntoCh* satisfies the following condition:

$$\exists i\ i=1,..,15\ \ Set_{i1} = extractSet(OIM_1,i) \wedge Set_{i2} = extractSet(OIM_2,i) \wedge$$
$$(\exists el_1 \in Set_{i2} \wedge el_1 \notin Set_{i1}) \wedge (\neg\exists el_2 \in Set_{i2} \setminus \{el_1\} \wedge el_2 \notin Set_{i1}) \wedge (\forall el_3 \in Set_{i1} \wedge el_3 \in Set_{i2}) \wedge$$
$$\forall k\ k \neq i\ \ Set_{k1} = extractSet(OIM_1,k) \wedge Set_{k2} = extractSet(OIM_2,k)$$
$$\forall el \in Set_{k1} \leftrightarrow el \in Set_{k2}$$

Definition 9 An ontology change *OntoCh* is an *subtractive* (*capacity reducing*) ontology change if it satisfies the following condition:

$$\exists i\ i=1,..,15\ \ Set_{i1} = extractSet(OIM_1,i) \wedge Set_{i2} = extractSet(OIM_2,i) \wedge$$
$$(\exists el_1 \in Set_{i1} \wedge el_1 \notin Set_{i2}) \wedge (\neg\exists el_2 \in Set_{i1} \setminus \{el_1\} \wedge el_2 \notin Set_{i2}) \wedge (\forall el_3 \in Set_{i2} \wedge el_3 \in Set_{i1}) \wedge$$
$$\forall k\ k \neq i\ \ Set_{k1} = extractSet(OIM_1,k) \wedge Set_{k2} = extractSet(OIM_2,k)$$
$$\forall el \in Set_{k1} \leftrightarrow el \in Set_{k2}$$

We note that in Definition 8 and Definition 9 OIM_1 is a given ontology and OIM_2 is a changed ontology, i.e. $OIM_2 = OntoCh(OIM_1)$.

An additive ontology change is an ontology change where new entities of the ontology model (e.g. a concept), are added to an ontology without altering the existing ones. A subtractive ontology change involves the removal of some piece of entities. Thus, more precisely, the ontology evolution can be defined as the creation and removal of ontology entities. An ontology is formed from a set of ontology entities and it evolves by adding or removing them.

The previously defined set of changes is complete and minimal. Completeness refers to the possibility of transforming an arbitrary ontology in any other, i.e. all changes are required for the evolution of the given ontology model. Minimality refers to the achievement of completeness with a minimal set of changes.

The proposed taxonomy of changes does not include "*modify*" changes. We found that these changes can be split into "*rename*" changes (e.g. "*rename*" a concept) or "*set*" changes (e.g. "*set*" the subconcept relationship between two concepts), which depends on the ontology entity the "*modify*" change should be applied to. These two interpretations of "*modify*" changes imply two reasons for their elimination from the proposed taxonomy of ontology changes.

The "*rename*" changes (such as rename a concept, rename a property, rename an instance) would be superfluous since there is lexical information about ontology entities (e.g. labels in different languages). In the KAON ontology language, lexical information is thought of as

meta-information about an ontology given that it talks about ontology entities. We defined Lexical OI-model[15] [80] as a well-known OI-model that models various lexical properties of ontology entities, such as labels, synonyms, stems or textual documentation. This Lexical OI-model can be included in other OI-models, which allows associating lexical entries with the ontology entities. Therefore, we consider ontology entities as URIs[16] – unique resource identifiers, which provide simple and extensible means for identifying a resource. Since a URI referring an ontology entity is a generic name, it does not need to be changed. Consequently, the "*rename*" changes can be kept out. The desired effect (e.g. renaming) is achieved by removal followed by an addition of lexical entries.

Figure 6 shows the lexical information[17] defined for the concept "*Person*" from the ontology shown in Figure 3. Two types of lexical information are specified: (i) labels such as "*Die Person*", "*Person*" and (ii) synonyms such as "*human being*", "*individual*". Note that all lexical information is multilingual. To rename a concept "*Person*" one has to create a new label and to remove the old one. Since labels are defined using the property instances for the property *KAON:VALUE*, the following two changes have to be applied: the *AddPropertyInstance* change and the *RemovePropertyInstance* change.

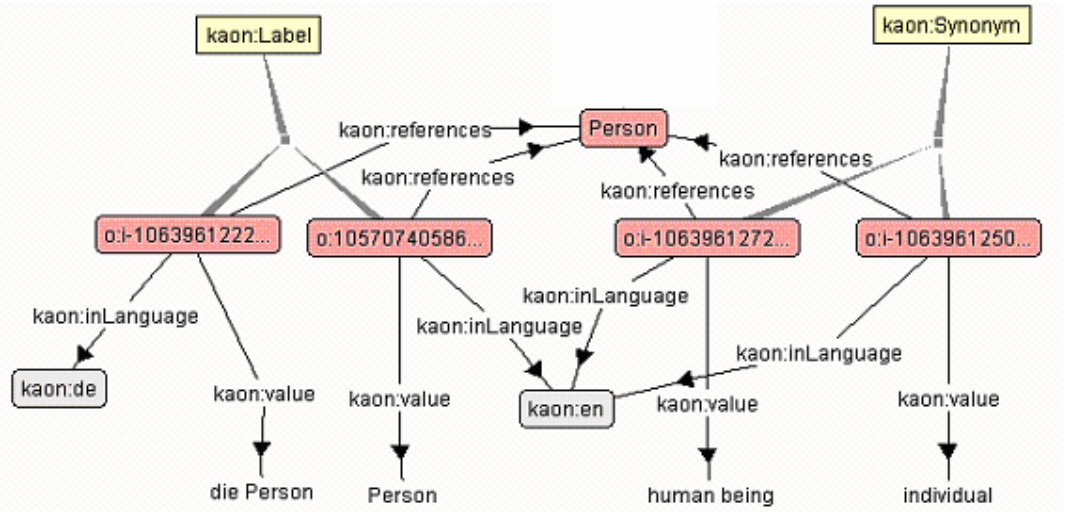


Figure 6. A part of the lexical layer for the concept "*Person*" from the ontology shown in Figure 3

The second reason for excluding "*modify*" changes is related to the second interpretation of these changes, i.e. "*set*" changes. We aim at a minimal set of changes, which means that no change defined in the taxonomy subsumes the functionality of the other primitives defined in

[15] Essentially an ontology can be thought of as a tuple *(V,A)*, where *V* is the vocabulary and *A* is the set of assertions governing the theory [54]. Interpreting this from the point of view of the KAON ontology language, the vocabulary *V* consists of a set of identifiers of ontology entities whereas all other elements from ontology definition form the set *A*. For example, the fact that "*Person*" is a concept is represented as word "*Person*" in the vocabulary and as a logical statement *concept(Person)* in A (i.e. *"Person"∈C*). We extended this approach by defining the Lexical OI-model. Therefore, the vocabulary does not contain only identifiers, but also synonyms, labels, etc.

[16] Uniform Resource Identifier (URI) is a compact string of characters for identifying an abstract or physical resource.

[17] The value of the lexical entry is given by the property *KAON:VALUE*, whereas the language of the value is specified through the *KAON:INLANGUAGE* property. The property *KAON:REFERENCES* establishes n : m relationship between lexical entries and ontology entities. The lexical structure consists of the several subconcepts, such as *KAON:Label*, *KAON:Synonym*, etc.

the taxonomy nor can it be specified by a sequence of the proposed changes. For example, the set up of the inheritance relationship between two concepts (e.g. *SetSubConcept* change) can be achieved trough two changes: the *AddSubConcept* change establishing the subconcept relationship between two concepts and the *RemoveSubConcept* change removing the subconcept relationship between concepts. Therefore, since the "*set*" changes can be realised as an addition followed by a removal, they would be considered as redundant changes. Their existence would cause the set of changes not to be minimal with respect to completeness.

On the other hand, these changes can improve the usability of an ontology evolution system significantly since an ontology engineer does not need to select a set of changes in order to realise her request for an update. We consider these changes as coarse-grained changes and discuss them in section 3.2.1.

All valid changes to manipulate an ontology can be specified by one ontology change or by a sequence of changes. Changes can be applied to an ontology in a consistent state, and once all the changes are made, the ontology and dependent artefacts must pass into another consistent state. It means that every change is guaranteed to maintain ontology consistency constraints. This is elaborated in the next subsection.

2.4.2 Semantics of Ontology Changes

The goal of the ontology evolution is to ensure that the application of ontology changes should result in an ontology conforming to the set of ontology consistency constraints introduced in section 2.3.

Definition 10 An ontology change preserves the ontology consistency if and only if it preserves all constraints of the underlying ontology model.

However, applying an ontology change alone will not always leave an ontology in a consistent state. As shown in Figure 7, making the concept "*PhD Student*" a parent concept of the concept "*Person*" causes the inconsistency since the invariant IC_2 - *Concept Hierarchy Invariant* related to the cycle in a concept hierarchy would be violated. Namely, the concept "*Person*" would be in the same time the parent concept of the concepts "*PhD Student*" (through the concept "*Student*") as well as its child concept.

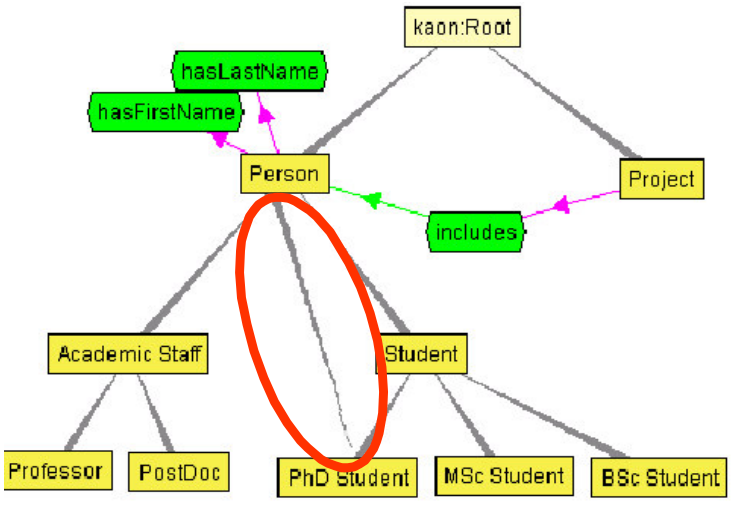


Figure 7. Inconsistent ontology due to a cycle in the concept hierarchy

This example shows that the ontology consistency has to be examined. Since verification in general concerns the correctness, the ontology verification is checking of the correctness of an ontology with the respect to the ontology consistency definition. There are two approaches to confirm ontology consistency [55]:

1. *a posteriori verification*;
2. *a priori verification*.

The first approach first executes a change, and then validates the updated ontology to check whether it satisfies the consistency constraints. The advantage is that all changes use a single check for their verification. The disadvantage lies in the fact that the check is performed after the change is completed. In the case that the check fails, the roll back of the ontology into the initial state is needed. Furthermore, checking the entire ontology is costly, even more so when the changes are small, local and mostly unnecessary. It is preferable to have an incremental checking mechanism where only the modified part of an ontology is checked. Moreover, when the validation is performed after a batch of changes and some inconsistency is detected, it is impossible to find out which change caused the inconsistency.

The second approach defines a respective set of preconditions for each change. It must be proven that, for each change, the consistency will be maintained if (1) an ontology is consistent prior to an update and (2) the preconditions are satisfied. Since the preconditions are specified for each change, checking the consistency can be limited to the local range affected by the change. Moreover, no roll back mechanism needs to be supported since inconsistencies are discovered prior to the modification. A disadvantage of using preconditions is that they need to be formulated in addition to the consistency constraints.

A summary of pros and cons of both approaches is shown in Table 2.

Table 2. A posteriori verification vs. a priori verification. "+" indicates positive aspect whereas "-" represents disadvantage.

	A posteriori verification	**A priori verification**
Roll-back	Necessary (-)	Not necessary (+)
Explanation	Impossible (-)	Possible (+)
Additional effort	No (+)	Yes (preconditions) (-)

In order to avoid getting inconsistent ontologies and reverting them back into a consistent state, we have decided to apply the second approach. Therefore, for each ontology change it is required to specify the necessary *preconditions* to describe the applicability conditions of a change. Moreover, both the approaches require the specification of sufficient *postconditions* to describe the effect of a change.

Definition 11 *Preconditions* of an ontology change comprise a set of assertions that must be true to be able to apply the change. If a precondition fails, and a change is nevertheless performed, the resulting ontology will be in an inconsistent state. For example, the precondition for the removal of a concept *c* is $c \in C \setminus \{Root\}$[18];

Definition 12 *Postconditions* of an ontology change comprise a set of assertions that must be true after applying a change. They describe the result of a change. For example, the removal of a concept *c* results in an assertion $c \notin C$.

In the rest of this thesis we use the following formal definition of an ontology change:

Definition 13 *An ontology change Ch* is a 4-tuple:

$$Ch := (name, args, preconditions, postconditions),$$

where:

- *name* is the identifier of the change - Table 1 contains the all possible change identifiers that are derived from the KAON ontology model;
- $args \in (C \cup P \cup I \cup L)^n$, $1 \leq n \leq 3$, is a list of one or more change arguments. There are changes with one, two or three arguments. For example, to remove the concept "*Person*" from an ontology, the only argument of the change *RemoveConcept* has to be "*Person*". To remove the range "*Person*" of the property "*includes*", the change *RemovePropertyRange("includes", "Person")* has to be applied. All changes related to property instances have three arguments. For example, the change *AddPropertyInstance("OntoLogging", "includes", "SteffenWezler")* is required to specify that the instance "*OntoLogging*" has the value "*SteffenWezler*" for the property "*includes*";
- *preconditions* – see Definition 11;
- *postconditions* – see Definition 12.

In order to simplify the notation of changes, in the rest of this thesis we use the following simplified syntax: *name(args, preconditions, postconditions)*. Moreover, to specify the request for a change the notation *name (args)* is used since the preconditions and the postconditions are general and do not depend on the concrete application of a change.

The role of an ontology change in the evolution of an ontology is shown in Figure 8. The application of a change (cf. *Ch* in Figure 8) to the ontology *O* (cf. *O*) results in an updated ontology (cf. *O'*). We assume that the ontology that has to be changed (cf. *O*) is a consistent ontology. A change will not be executed unless the corresponding preconditions are satisfied and a change will not be committed unless the corresponding postconditions are accomplished. Therefore, preconditions and postconditions are enforced on each ontology change.

[18] Note that *C* is a set of ontology concepts as introduced in the Definition 3 whereas *Root* is a predefined concept (*Root*$\in$*C)* that is, according to the IC_3 - *Rootedness Invariant*, contained in all ontologies.

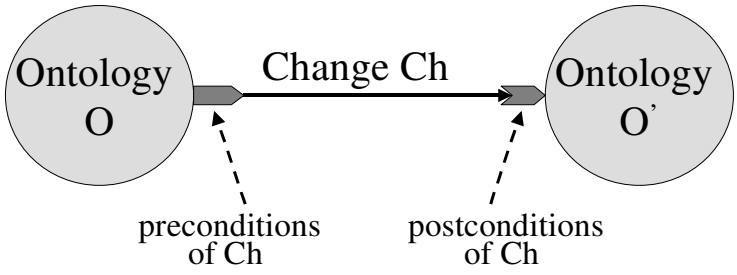


Figure 8. The application of an ontology change Ch

In order to formalise this process we introduce the following definitions:

- the function $preconditions(O, Ch) = \begin{cases} true, & \text{if an ontology } O \text{ satisfies a set of preconditions of a change } Ch \text{ (see Definition 11)} \\ false, & \text{otherwise} \end{cases}$
- the function $postconditions(O, Ch) = \begin{cases} true, & \text{if an ontology } O \text{ satisfies a set of postconditions of a change } Ch \text{ (see Definition 12)} \\ false, & \text{otherwise.} \end{cases}$

Definition 14 Given an ontology *O* and the request for a change *Ch*, *the application of an ontology change Ch* to the ontology *O* results in an ontology *O'*:

$$O' = Ch \circ O = Ch(O)$$

where ° is an operator performing the application of changes from *Ch* on the ontology *O*, under the following conditions:

$$preconditions(O,Ch)=true \wedge postconditions(O',Ch)=true$$

Interpreting Definition 14 on the Figure 8, it can be concluded that the change *Ch* can be applied if and only if the ontology *O* satisfies the preconditions of this change and there exists at least one ontology *O'* that satisfies the set of postconditions of this change. Therefore, the preconditions of a change (cf. *Ch* in Figure 8) indicate the initial state prior to its execution (i.e. the ontology *O*) and the postconditions indicate the state after it is applied (i.e. the ontology *O'*). Postconditions allow us to specify what the result of an update is. Indeed, the postconditions represent a subset of the ontology consistency constraints that cannot be invalidated after the change application.

Note that the postconditions do not include the effects on the ontology change on the other parts of the ontology. Therefore, it may happen that the ontology *O'* is not in a consistent state. To resolve that problem, additional changes have to be generated in order to satisfy the consistency of the ontology *O'* (see chapter 4). These induced changes may introduce new inconsistencies. However, after finishing the resolution of all changes (including all induced changes) the consistency of an ontology must be true. In that sense, the consistency can be viewed as universal postconditions for all changes.

In the rest of this section we define the precise syntax and semantics of two ontology changes, namely *AddSubConcept* and *RemoveSubConcept*. A similar strategy is applied to all other changes as well. The syntax defines the sentences in the language whereas semantics defines the "meaning" of changes. Note that preconditions and postconditions contain only constraints related to the invariants since they are mandatory consistency constraints. In the case that the consistency definition includes the soft- and/or user-defined constraints, the preconditions and the postconditions have to be extended.

Change:	**AddSubConcept**
Syntax:	*AddSubConcept* (c_1, c_2)
Semantics:	Create a subconcept relationship between the child concept c_1 and the parent concept c_2
Preconditions:	$c_1 \in C \setminus \{Root\}$
	$c_2 \in C \setminus \{c_1\}$
	$(c_1,c_2) \notin Hc$
	$(c_2,c_1) \notin Hc^*$
Postconditions:	$(c_1,c_2) \in Hc$

Change:	**RemoveSubConcept**
Syntax:	*RemoveSubConcept*(c_1,c_2)
Semantics:	Breaks the subconcept relationship between concepts c_1 and c_2
Preconditions:	$(c_1,c_2) \in Hc$
	$\exists c \in C/\{c_2\}\ (c_1,c) \in Hc$
Postconditions:	$(c_1,c_2) \notin Hc$

As can be concluded from the definitions of ontology changes, they are strongly related to the notion of the consistency since the preconditions as well as postconditions of each change represent the subset of ontology consistency constraints. One of the challenges of the ontology evolution framework is to ensure that all consistency constraints are fulfilled after the application of a change. For example, the postcondition for the removal of a concept c is that the concept is not in the ontology anymore (i.e. $c \notin C$). However, for an ontology to be consistent the following conditions have to be satisfied as well:

$\neg\exists c_1 \in C\ (c,c_1) \in Hc$

$\neg\exists c_1 \in C\ (c_1,c) \in Hc$

$\neg\exists p \in P\ \ c \in domain(p)$

$\neg\exists p \in P\ \ c \in range(p)$

$\neg\exists i \in I\ i \in instconc(c)$

The methods for guarantying that are discussed in chapter 4.

The definitions of ontology changes justify the distinctions of *additive* and *subtractive* changes. As can be seen, the changes related to the addition and the removal of an ontology

entity have the opposite sets of conditions. Therefore, for each change we introduce its inverse change.

Definition 15 Let $Ch(args, preconditions_1, postconditions_1)$ be an ontology change. *An inverse change* of a change, denoted as $Ch^{-1}(args, postconditions_2, preconditions_2)$ is an ontology change that satisfies the following condition:

$$preconditions(O, Ch)=true \wedge postconditions(O, Ch^{-1})=true \wedge$$
$$postconditions(Ch(O), Ch)=true \wedge preconditions(Ch(O), Ch^{-1})=true$$

where O is an ontology.

For example, the concept removal and the concept addition are inverse changes. After the removal of the concept "*Student*" the preconditions for the addition of the concept "*Student*" are satisfied and vice versa.

It is important to note that the application of a change and then its inverse change (or vice versa) to an ontology result in the same ontology, i.e. $Ch^{-1} \circ Ch(O) = O$.

2.5 Related Work

Evolution in general is a research issue that has been thoroughly investigated. This objective of this section is the presentation of the relevant state of the art for the ontology evolution. Since an ontology is a conceptual model of a domain, we discuss the evolution of other conceptual models. Indeed, we discuss the following areas:

- schema evolution in (relational and object-oriented) databases;
- XML schema evolution;
- maintenance of knowledge-based systems.

These areas come closest to the approach presented in this thesis. However, there are many differences that will be pointed out. The comparison of our approach with the existing approaches for the ontology management is given at the end of each section of this thesis separately.

2.5.1 Ontology evolution vs. database schema evolution

Evolution in databases addresses the problem that the logical schema of a database is likely to undergo changes, even if the database is already populated [113]. The schema evolution in the relational databases is the starting point for all evolution issues. In the case study related to health management system [117], the authors have shown that the modification of a database schema happens quite often. The result of this study revealed that, during the lifetime of the system based on this schema, the number of relations increased by 139%, the number of attributes by 274% and that every relation has been modified.

We came to the similar results during the developing ontologies. For example, the ontology that has been developed as a backbone of the portal of our institute in January 2001 is still a subject of changes (see section 6.3.1). The initial version of this ontology contained concepts such as "*Person*", "*Project*", "*ResearchGroup*", the relations between them such as "*worksIn*", "*manages*", attributes such as "*hasName*" etc. To fulfil the business requirements (e.g. the information about research topics is relevant), the users' needs (e.g. they would like

to be able to search for projects related to the particular research topic) or to organise the information in a better way (e.g. research topics are not instances but rather concepts related in a topic hierarchy), this ontology has been modified regularly, once in a month. The last change that occurred in April 2004 was to extend the ontology with the information about news. The actual version of this ontology contains only a small part of this initial ontology, not to mention instances whose number has been increased drastically.

The schema evolution in the relational databases is only poorly supported. The standard SQL DDL (Data Definition Language) allows for changes in the table definition (e.g. adding attributes). However, the related consistency problems have not been considered so far. The administrators are responsible for maintaining consistency. It means that the instance adaptation has to be done manually, e.g. by using SQL UPDATE queries. On the contrary, our intention was:

- to help an ontology engineer to modify an ontology by providing changes at the right level of the abstraction;
- to automate the change resolution or at least to inform an ontology engineer about all consequences of a change.

Since ontology management systems are mostly implemented on top of the relational database management system, the ontology evolution could only profit from resolving the aforementioned problems in the relational databases.

With the appearance of object-oriented database systems, the schema evolution became a research issue. The object-oriented schema evolution is more relevant for the ontology evolution due to two reasons:

- the object-oriented database models provide a semantically richer model than the relational database model and, therefore, can be considered as an extension of the relational database evolution;
- the object-oriented database models are more similar to the ontology models due to the complex inheritance hierarchies.

There are many approaches dealing with the object-oriented schema evolution issue ([3], [36], [55], etc.). They address two main questions: the effects of a schema change on the schema itself and the effects of the schema change on the underlying instances. The first problem is resolved either by defining rules that must be followed to maintain the schema consistency ([3], [154]) or by introducing axioms (with an inference mechanism) that guarantee the consistency ([104]). If the schema modification of populated database is allowed, the second problem is how to propagate the changes to the instances. The proposed solutions include: (i) the data migration for the immediate adaptation of the existing instances to the changed schema; (ii) the mechanisms for the synchronisation between data and schema and (iii) the combination of them. The synchronisation mechanisms are realised as: (a) delayed (lazy) conversion, when instances are only converted on demand; (b) screening, when changes are propagated via deferred object conversion; (c) versioning, when changes are never propagated and objects are indeed assigned to different schemas.

Our goal is to build a general framework for the ontology evolution that resolves the aforementioned problems. Regarding the first problem, in chapter 4 we propose the combination of the two complementary approaches, which seems to be promising. As far as the change propagation is concerned, we distinguish the centralised and distributed scenario (see chapter 5) and apply the methods similar to the data migration and the delayed (lazy) conversion, respectively. The decision is driven by the contradictory objectives (i.e. global consistency vs. runtime performance) that favour one or other solution.

However, we do not have only to adapt the existing approaches to the ontology evolution, but rather to extend them due to many reasons. They are a consequence of the different knowledge model and different usage paradigm, which is elaborated in [94]. Here we summarise the main differences between ontologies and schemas that have direct implication on the ontology evolution.

There are many differences between ontology engineering and object-oriented modelling. An ontology reflects the structure of the world, it is often about the structure of concepts; besides, the actual physical representation is not an issue. On the other hand, an object-oriented structure reflects the structure of the data and code. It is usually about behaviour since the integral part of an object-oriented model comprises methods. The physical representation of data (int, char, etc) is also a part of a model.

The evolution is driven by the set of changes that have to preserve the consistency. Therefore, each approach requires (i) the definition of the consistency and (ii) the explicit specification of changes that can be applied. Both the consistency definition and the set of changes heavily depend of the underlying model and, thus, they vary from one system to another. Since the ontology model is richer than an object-oriented model [94], (i) the consistency definition includes more consistency constraints; and (ii) the number of possible changes is much richer than in a typical (relational or object-oriented) database schema.

Regarding the consistency definition, we adjust of the object-oriented consistency definition, which consists of the set of invariants of the object-oriented model, to the formal semantics of the ontology model (see section 2.3). For example, a property is a first-class citizen and therefore, it can exist without being attached to any concepts. Next, we extend the ontology consistency definition with the *soft constraints* and the *user-defined constraint*s since our goal is to enable an ontology engineer to adapt an ontology to her needs in the easiest manner.

Moreover, the model is reflected in the set of changes. Each modelling primitive requires at least two additional necessary changes – one for the addition of the entity and one for its removal (see section 2.4.1). For example, changes related to the property hierarchy, cardinality constraints, included ontology models, etc., are specific for ontology models and are not contained in the taxonomy of the object-oriented schema changes. Further, it is not sufficient to enumerate changes based on the underlying model. It is required to specify the semantics of changes in the form of preconditions, postconditions and possible actions that are necessary to preserve the consistency (see section 4.2). Due to the interwoven structure of an ontology, the number of inconsistencies that a change may cause is larger than the number of inconsistencies that arise during the database schema evolution. Therefore, the actions that have to be specified for each change are much more complex.

Many ontology languages allow the ontology reuse. Therefore, there is a need for changes that permit the including/excluding an ontology in/from other ontology. Consequently, methods for propagating changes not only to ontology instances, but also to the ontologies that reuse it are necessary. Moreover, the distributed environment such as the Semantic Web has to be taken into account.

Further differences between ontologies and schemas stem from the explicit semantics that ontologies provide [94]. This allows for the usage of reasoning methods for checking the consistency. Moreover, there is no clear difference between the conceptual level and the instances. A concept may be treated as an instance and vice versa. This has serious consequence on the ontology evolution. Further, all ontology entities and not only ontology instances (which correspond to the database objects) may be used in the same way. For example, the results of the ontology-based queries do not include only ontology instances. Thus, the ontology evolution has to take into account the effect of an ontology change on queries [134].

2.5.2 Ontology evolution vs. XML schema evolution

XML as a data storage becomes more and more popular in recent years. It is used not only as a Web interface to traditional databases but as standalone data storage as well. In both cases it is appropriate to talk about XML database with structure defined by a collection of XML Schemas (or DTDs). XML data is stored in a collection of valid XML documents with structure regulated by corresponding XML Schemas.

Research and practical experience show that XML and XML Schema evolution are of critical importance to the successful development and management of any complex XML-based applications, especially content management systems. As XML schema is changing, the XML schema evolution becomes an important area of research. However, there are only few approaches that take this problem into consideration.

Many similarities between database schema and the XML schema allow for the building the XML schema evolution on the extensive research in the database schema evolution. The first step regarding schema evolution issues from a scientific viewpoint is to fix the meaning of schemas and its changes. Thus, when investigating the XML schema evolution, the starting point is a formal model of the XML Schema and the taxonomy of XML(S) changes. The first attempt in this direction is given in [136]. To manage the evolution of DTDs and XML documents, the XML Evolution Manager (XEM) was developed. XEM provides a minimal yet complete taxonomy of basic change primitives. For a change in the XML document, XEM ensures that the modified XML document conforms to its DTD both in structure and constraints. For a schema change, it ensures that the new DTD is well formed, and all existing XML documents are also transformed to conform to the modified DTD. However, it does not take into account that a DTD change can corrupt the DTDs that include it as well as application programs running against the DTD or its XML documents. On the contrary, in chapter 5 we propose an approach that brings automatically all dependent artefacts into a consistent state after an ontology update has been performed.

In [17] the author investigates the management of data changes on the Web in general. More precisely, he focuses on semi-structured data such as XML. He distinguishes between changes at a “microscopic” scale, such as an XML document as well as at a “macroscopic” scale, the scale of the graph of the Web. The author proposes an algorithm to detect changes, and a formalism to represent these changes. Similarly, we introduce three levels of ontology changes: elementary, composite and complex changes (see section 3.2.1). Further, we allow an ontology engineer to specify an arbitrary complex request for a change as a composition of existing changes and propose an approach to resolving it. Finally, we propose an approach for the continuous ontology improvement by semi-automatic discovery of changes based on the analysis of the end-users’ behaviours.

A number of projects and tools have emerged to map XML and similar semi-structured data formats to traditional database systems. In [78] the authors investigate semi-structured data in relational databases. The data managed in Lore is not confined to a schema, and it may be irregular or incomplete. In general, Lore attempts to take advantages of structure where it exists, but also handles irregular data as gracefully as possible. Since we agree that the global consistency on the (Semantic) Web can never be achieved, we define the ontology consistency on three different levels. Further, we propose methods for the synchronisation as elaborated in chapter 5.

Oracle’s XML SQL Utility (XSU) and IBM’s DB2 XML Extender are well-known commercial relational database products extended with XML support. They allow implementing changes to XML schema by mapping the existing data to the new schema. Instead of having to export and re-import all of the XML data, the user has to create the

XSLT (XSL Transformation) style sheet to transform old documents into the new schema, and the database takes care of the rest. However, this only partially eases the management of XML data since it requires users to be aware of the underlying storage system, its data model, and the mapping mechanism between XML, XML Schema and the underlying storage model. It prevents users from expressing desired changes independent of the underlying data management system. On the contrary, we pay a special attention to the usability issue. We take into account the users with the different background regarding the ontology management and enable the customisation of the ontology evolution to the current need of the user.

The research regarding the XML schema evolution and the corresponding tool support are still in early stages. Even though the full maturity is not achieved, the experiences from this area are useful. XML schema evolution considers the schema changes as well as the data changes and their impact on the consistency. Note that the database schema evolution mainly focuses only on the schema changes. The main reason for this distinction is that a change in a XML document may induce inconsistencies in the same document and in the XML Schema this document is based on. Thus, it is critical to detect in advance whether an update is a valid change that preserves the consistency, i.e. whether a required change will result in an XML document still conforming to the given XML Schema. The similar strategy has to be applied to the ontology evolution since the changes at the ontology instance level may provoke inconsistencies to the conceptual level when the cardinality constraints are invalidated.

Further, for any change in the XML schema, an evolution support system would need to verify that the suggested change leads to (i) a new well-formed XML schema and (ii) corresponding changes are propagated to all old XML documents to conform to the changed XML Schema. Similarly, an ontology evolution system must guarantee the consistency of an ontology that a change is applied to as well as of all artefacts (e.g. dependent ontologies) that reuse it. In both the scenarios the physical distribution of dependent elements (i.e. XML documents or dependent ontologies) is very important. However, the XML evolution considers only the change propagation between XML Schema and its document. Even though an XML Schema may include other schemas, the problem of propagating changes from the included schema to the including schema is not considered.

2.5.3 Ontology evolution vs. maintenance of the knowledge-based systems

The term “maintenance” refers to the process of keeping in working order or good repair some object such as a building or vehicle. In computer systems terminology, this is translated to the process of modifying a program and its documentation after it has been delivered and is in use. It also includes the addition of new features, which the term maintenance in classical sense does not cover.

Maintenance cannot be neglected in the development of conventional software systems, and therefore, it cannot be ignored in knowledge-based systems. The knowledge-based systems have developed from experts systems, which are software systems behaving like an expert in some domain when solving a problem in that domain. The essential characteristic of knowledge-based systems is that the knowledge representation and the means for manipulating that knowledge are separated. Several paradigms for representing knowledge have been suggested such as “if-then” production rules, propositional and predicate logic and structured objects (e.g. originally frames, now objects in object-oriented programming sense). The inference engine then contains methods for manipulating that knowledge. The inference engine is usually written in procedural code, which can be maintained using traditional methods for the software maintenance. Therefore, the maintenance of the knowledge-based systems is focused on the maintenance of its knowledge base. More precisely, knowledge

base maintenance is the process of reflecting over some knowledge-based system in order to handle a new situation [81].

Since ontologies can be considered as enriched knowledge bases (due to richer set of modelling primitives such as hierarchy, features defined for properties etc.), the experiences from the knowledge-based system maintenance (in form of methodological as well as tools support) should be taken into consideration [69], [83]. Therefore, another research area that is relevant as state of the art for the ontology evolution is the maintenance of the knowledge-based systems. An excellent overview of research problems related to the knowledge-based maintenance is given in [18].

Since the maintenance in general can be categorised into four groups [56]: (i) adaptive maintenance; (ii) perfective maintenance; (iii) corrective maintenance and (iv) preventive maintenance, we comment upon each of them.

Adaptive maintenance results from the changes in the environment in which the knowledge-based system operates or from the better understanding of a domain of a domain expert. Both causes of changes are important for the ontology evolution as well. Regarding the first cause, an ontology must be able to respond to the changes in the domain. Moreover, an ontology is usually not developed automatically. Rather, it is extracted from the domain experts by applying methodologies for the ontology development [46]. Since all experts are continually learning, the acquisition of new knowledge is required. Therefore, an ontology that represents this knowledge formally and explicitly, needs to be maintained.

Perfective maintenance of knowledge-based systems results from changes in the users' requirements. For example, it includes the amount and the organisation of the information presented to the users. Since ontologies are developed incrementally and collaboratively, the ontology evolution has to take into account the changes in the users' needs. For example, to gain the better understanding of an ontology concept, it would be better to organise its subconcepts into a deeper concept hierarchy, and, consequently, to focus the user's attention on the important parts.

Corrective maintenance of the knowledge-based systems is maintenance that is not perfective or adaptive. It is applied when the knowledge-based system is not behaving as it should. For example, given a particular set of facts, a wrong conclusion may be drawn or a conclusion not arrived at. These problems arise from the syntactic and semantic errors and errors in identifying the knowledge stored in the knowledge base. The ontology evolution has to be driven by the corrective actions as well. This can be achieved (i) by formalising the ontology consistency, which assures the avoidance of the syntactic errors and (ii) by taking into account the usability of an ontology as described in chapter 6.

Preventive maintenance combines all actions necessary for improvements to avoid future problems. Indeed, the need for the knowledge-based maintenance is not only because of the knowledge base is wrong in content. It may be possible that the knowledge base is wrong in structure. This distinction is very important since the defects in the structure can be identified or at least suggested by analysing the knowledge base alone. We adopted this idea to the ontology evolution, which is described in section 3.4. Further, we extend this approach by including the usage of an ontology in the domain, which it represents.

One knowledge-based system that has undergone extensive maintenance is the XCON (originally referred as DEC R1/XEC) [4]. This system has been built using production rules. The experience of this system (as well as many others) shows that the addition of new rules is not sufficient since a new rule may require the rewriting of many already existing rules. Therefore, there is a need for a mechanism that is able to list automatically the rules (or the frames) that will be affected. This also holds for the ontology evolution since a small change

in one part of an ontology may cause a lot of problems in other parts of this ontology. However, the ontology evolution is much more complex not only because of the richer knowledge model. The most important reason is that an ontology may include other ontologies, some of them distributed over the Web. Therefore, often changes to some ontologies have to be made if included ontology is updated. Such situations complicate the ontology evolution and increase the overall complexity.

In [12] the authors consider the possibilities to enable a domain expert to maintain her own knowledge in a knowledge based system. They constructed a domain ontology and a task model for the knowledge based system to be maintained. Further, they developed a maintenance tool for the existing knowledge based system by separating domain and system concepts. Similarly, we model ontology evolution problem (see chapter 4). By considering the usability issue, we propose an approach for the usage-driven ontology evolution that supports an inexperienced user, who does not need to be an ontology engineer, to develop/modify an ontology. Moreover, we allow an ontology engineer (i) to specify her complex request for a change declaratively without considering how it can be realised and (ii) to control the effects of ontology changes.

In [140] the authors propose the script-based knowledge acquisition tools that help users follow typical modification procedures (i.e. KA scripts). Based on this work we define the dependencies between ontology changes in form of rules that model side effects of a change on the other related entities. However, we go a step further by introducing evolution strategies as a means to "find" the consistent ontology that meets the needs of an ontology engineer. In this way the ontology engineer is able to control the resolution of changes.

2.6 Conclusion

The goal of this thesis is to build a general framework for the ontology evolution. As the basis of this framework, the KAON ontology model (with its consistency definition and its changes) is used. However, the basic ideas are not strongly bound to this ontology model. The main principles can be more or less easily adapted to other ontology models. Since the evolution in general is driven by the set of changes and has to preserve the consistency, in this section we define the ontology consistency and provide the complete taxonomy of the consistency-preserving ontology changes for the KAON ontology model.

3 Ontology Evolution Process

Ontology evolution is the timely adaptation of an ontology to the changes in the business requirements, to trends in the ontology instances and the patterns of the usage of the ontology-based application, as well as the consistent management/propagation of these changes to dependent artefacts. A modification in one part of the ontology may generate many inconsistencies in other parts of the same ontology, in the ontology-based instances as well as in depending ontologies and applications that are based on this ontology [66]. This variety of causes and consequences of the ontology changes makes the ontology evolution a very complex operation that should be considered as both an organisational and a technical process [119]. It requires a careful analysis of the types of the ontology changes [74] that can trigger the ontology evolution as well as the environment in which the whole ontology evolution process is realised [133].

Based on the requirements for the ontology evolution system that is discussed in section 3.1, we identify a possible six-phase evolution process and focus on providing the user with capabilities to control and customise it.

3.1 Requirements Capturing

To develop a system, one has to start from the requirements for that system. Requirements' gathering simply means "figuring out what to make before to make it". Typically, there are requirements that capture the intended behaviour of the system. This behaviour may be expressed as services, tasks or functions the system is required to perform. In development of any system (software, product, etc.), it is useful to distinguish between the baseline capabilities necessary for any system to compete in its domain, and features that differentiate the system from others. These features include the additional capabilities or they differ from the basic capabilities along some quality attribute such as customisation.

Based on our experience[19] in building ontologies and using them in several applications [133], we have formulated the following set of design requirements for an ontology evolution system:

1. *Mandatory requirement*:
    a. *Functional requirement* – Ontology evolution has to (i) enable the handling of the given ontology changes [37] and (ii) ensure the consistency of the underlying ontology and all dependent artefacts [127];

[19] Since everyone's knowledge and experience is needed during the analysis of the requirements for the ontology evolution, the key persons for this analysis are people familiar with the ontology development, who are aware of the problems occurring during this process and whose know-how makes success of the analysis.

2. *Supplementary requirements*:
    a. *Guidance requirement* - Ontology evolution should be supervised allowing the user to manage changes more easily [140];
    b. *Refinement requirement* - Ontology evolution should offer advice to user for continual ontology refinement [39], [95].

The functional requirement is the essential one for any ontology evolution system – after applying a change to a consistent ontology, the ontology should remain in a consistent state. The guidance requirement complements the first one by presenting the ontology engineers with information needed to control changes. It also helps them make appropriate decisions since it enables the assessment of the effectiveness of the change management activity, such as the identification and the overcoming of undesired changes. The last one assists the preparation for a change. It states that potential changes improving the ontology may be discovered semi-automatically from the ontology-based data and through the analysis of the user's behaviour.

Whereas the first requirement is mandatory, the last two requirements help bridge the gap between the functionality offered by an ontology evolution system and the needs of ontology engineers who use this system to modify an ontology. Indeed, the supplementary requirements significantly improve the usability[20] of the ontology evolution system by providing a qualitatively new level of services. On the other hand, the functional requirement defines the extent to which, independent of the usability, an ontology evolution system provides means for resolving a user's request for a change.

The proposed set of the requirements for the ontology evolution does not contain the non-functional requirements[21] such as efficiency, reliability, maintainability, etc. These requirements are relevant for the implementation of a system since they constrain the design of the system. For example, they restrict the freedom of software engineers as they make design decisions because they limit the resources that can be used just as they set bounds on aspects of the software's quality. However, the non-functional requirements are not important for defining the conceptual description of an ontology evolution system since they do not describe services that this system has to provide.

A more careful analysis of the proposed requirements (e.g. the changes have to be captured, analysed, applied and validated by the user) implies the necessity to consider the ontology evolution problem as a composition of several subproblems realised in a determined sequence. This sequence of activities, which resolve ontology changes in a composite way, is called the ontology evolution process. Consequently, the system, i.e. software, which copes up with the ontology evolution problem, has to be process-based, following currently the most popular programming paradigm in the business software development.

In the remainder of this section, we analyse the above-mentioned requirements and derive the ontology evolution process that fulfils them.

[20] The usability in general is an important quality of a system that measures the extent to which a product or process is learnable, enables users to be productive and avoid errors, and also subjectively satisfies them.

[21] Non-functional requirements or system qualities capture required properties of a system, such as performance, security, maintainability, etc.-in other words, how well some behavioral or structural aspect of the system should be accomplished.

3.2 Functional Requirement

The functional requirement states that after applying and resolving changes in an ontology already in a consistent state (see Definition 6), the ontology, its distributed instances and dependent ontologies/applications must remain in (another) consistent state. This requirement encompasses two crucial aspects of the ontology evolution:

- enabling the resolution of changes, and,
- maintaining the system consistency.

The functional requirement may be realised through the following four phases as shown in Figure 9. These phases form a core ontology evolution process since they realise the mandatory (i.e. functional) requirement.

The core ontology evolution process starts with representing a request for a change formally and explicitly as one or more ontology changes (c.f. *Representation* in Figure 9). Then, *the semantics of change phase* prevents inconsistencies by computing additional changes that guarantee the transition of the ontology into another consistent state. In *the change propagation phase* all dependent artefacts (distributed ontology instances, dependent ontologies and applications using the ontology that has to be changed) are updated. During *the change implementation phase* required and derived changes are applied to the ontology in a transactional manner. All phases of the core ontology evolution process are elaborated in the rest of this section.

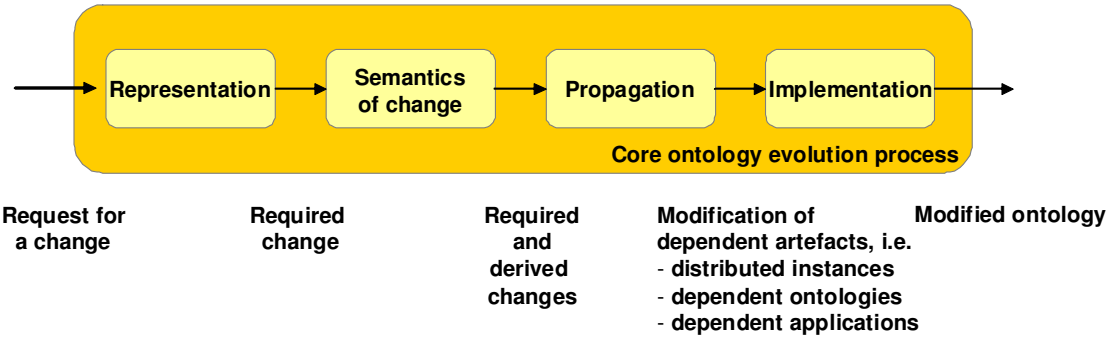


Figure 9. Four elementary phases of the ontology evolution process enabling the resolving changes while keeping the consistency

3.2.1 Change Representation

To resolve changes, they have to be identified and represented in a suitable format ([74], [101]). The ontology changes shown in Table 1 are derived from our ontology model definition described in section 2.2. They specify the fine-grained changes that can be performed in the course of the ontology evolution. They are called elementary changes since they cannot be decomposed into simpler changes.

Definition 16 *An elementary ontology change* is an ontology change that modifies (adds or removes) only *one* entity of the ontology model.

However, this granularity of the ontology changes is not always appropriate. Often, the intent of the changes may be expressed on a higher level. For example, an ontology engineer may need to generate a common superconcept of two concepts. Let's consider the application of this request on the ontology shown in Figure 3. The goal is to "group" the concept "*PhD Student*" and "*MSc Student*" into a common concept "*Working Student*". The ontology engineer may bring the ontology into the desired state through successive application of a list of elementary ontology changes:

1. *AddConcept("Working Student")* – creates a new concept "*Working Student*";
2. *AddSubConcept("Working Student", "Student")* – defines the concept "*Working Student*" as a subconcept of the concept "Student";
3. *RemoveSubConcept("PhD Student", "Student")* – removes the inheritance relation from "*PhD Student*" to its current parent "*Student*";
4. *AddSubConcept ("PhD Student", "Working Student")* – defines the concept "*PhD Student*" as a subconcept of the concept "*Working Student*",
5. *RemoveSubConcept("MSc Student", "Student")* – removes the inheritance relation from "*MSc Student*" to its current parent "*Student*";
6. *AddSubConcept ("MSc Student", "Working Student")* – defines the concept "*MSc Student*" as a subconcept of the concept "*Working Student*".

However, this has significant drawbacks:

- There is an impedance mismatch between the intent of the request and the way the intent is achieved. It is required to create a superconcept of two concepts, but one needs to translate this operation into six separate steps, making the whole process error prone;
- A lot of unnecessary changes may be performed if each change is applied alone. For example, removing the subconcept relation (e.g. between the concept "*PhD Student*" and the concept "*Student*") may introduce changes to property instantiations (i.e. the deletion of all property instances for the properties that are inherited from the parent concept). However, some of them (e.g. the "*hasFirstName*" property) should be reversed when assigning the subconcept-of relation from "*PhD Student*" to "*Working Student*".

To avoid these drawbacks, it should be possible to express changes on a more coarse level, with the intent of change directly visible. We introduce the composite ontology changes representing a group of elementary changes applied together. Whereas an elementary change can be seen as an isolated modification of an ontology, a composite change defines a "context" of the evolution: It results from an analysis of entities that are close to each other [108]. Therefore, to define a set of composite changes the "closeness" of ontology entities should be clarified.

Definition 17 *The neighbourhood* of an ontology entity is defined in the following way:

$$\text{Neighbourhood(e)} = \begin{cases} CN(e), & e \in C \\ PN(e), & e \in P \\ IN(e), & e \in I \end{cases}$$

where *CN* represents the neighbourhood of an ontology concept, *PN* corresponds to the neighbourhood of an ontology property and *IN* stands for the neighbourhood of an ontology instance. They are calculated as:

$$CN(e) = \{ x \in E \mid Hc(x,e) \vee Hc(e,x) \vee e \in domain(x) \vee e \in range(x) \vee x \in instconc(e) \}$$

$$PN(e) = \{ x \in E \mid x \in domain(e) \vee x \in range(e) \vee Hp(x,e) \vee Hp(e,x) \vee (\exists i \in I \quad x \in instprop(e,i) \vee \quad i \in instprop(e,x)) \}$$

$$IN(e) = \{ x \in E \mid e \in instconc(x) \vee (\exists i \in I \quad e \in instprop(x,i) \vee \quad i \in instprop(x,e)) \}$$

The neighbourhood of a concept consists of its subconcept, superconcepts, properties, for which it is specified as a domain or as a range concept, and instances defined for that concept. The neighbourhood of a property contains its domain concepts, range concepts, subproperties, superpropeties, instances it is defined for as well as instances it points to.The neighbourhood of an instance includes its concepts, properties that are instantiated for it as well as properties that point to it.

As can be noticed, we define the neighbourhood only for concepts, properties and instances. Between these ontology entities it is possible to establish several types of relationships as shown in Table 3. By abstracting these relationships between entities, an ontology is considered as a graph. The nodes correspond to the ontology entities and the edges represent all "links" between them. Consequently, the "closeness" between two entities is defined as a length of a path (i.e. a number of edges) between nodes that abstract the entities.

Table 3. Different types of "links" between ontology entities

	Concept	**Property**	**Instance**
Concept	Concept hierarchy	Property Domain/Range	Instantiation
Property	Property Domain/Range	Property hierarchy	Property instantiation
Instance	Instantiation	Property instantiation	-

Two nodes that are linked through just one link are close together and they are called neighbours. Therefore, the neighbourhood of an entity consists of all entities that are directly linked to it by one link of one of the following types: (concept and property) inheritance, property domain, property range, instantiation, property instantiation.

Figure 10 shows the neighbourhood of a concept in general as well as the neighbourhood of the concrete concept "*Person*" from the ontology depicted in Figure 3. As can be seen, the neighbours of a concept are (i) its parents or children (because of the "subConcept" link), (ii) properties whose domain or range is a considered concept (because of the "propertyDomain"/"propertyRange" links) and (iii) its instances (because of the "instanceOf" link). Therefore, the neighbourhood of the concept "*Person*" (see Figure 3) contains the

following entities: the "*Root*", "*Academic Staff*" and "*Student*" concepts due to the concept hierarchy, the "*hasFirstName*" and "*hasLastName*" properties due to the property domain relationship and the property "*includes*" due to the property range relationship.

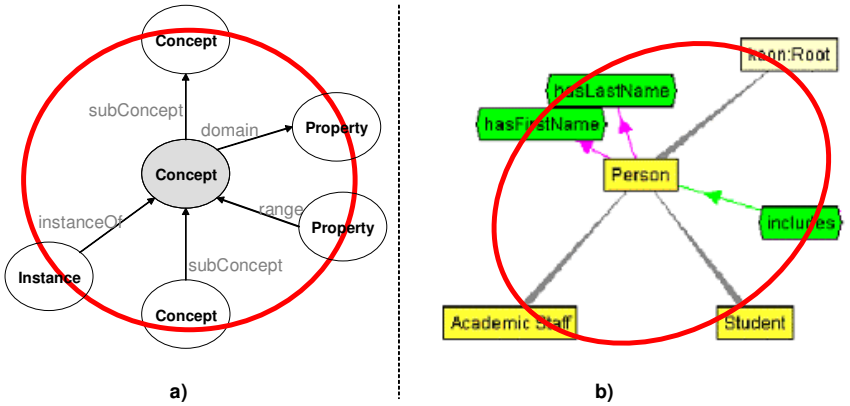


Figure 10. Neighbourhood of a concept. (a) Concept's neighbourhood in general; (b) the neighbourhood of the concept "*Person*" from the ontology shown in Figure 3

To clarify the property neighbourhood and the instance neighbourhood let's consider the ontology shown in Figure 4. The neighbourhood of the property "*includes*" consists of the concept "*Project*" (because of the "propertyDomain" link), the concept "*Person*" (because of the "propertyRange" link), and the instances "*OntoLogging*" and "*SteffenWezler*" (because of the "propertyInstantiation" link). On the other hand, the neighbourhood of the instance "*SteffenWezler*" contains the concept "*BSc Student*" (because of the "instanceOf" link) and the properties "*hasFirstName*", "*hasLastName*" and "*includes*" (because of the "propertyInstantiation" link).

Based on the definition of the neighbourhood of an ontology entity we define the composite changes in the following way:

Definition 18 A *composite change* is an ontology change that modifies (creates, removes or changes) the *neighborhood* of an ontology entity.

Similarly to [108], the definition of composite changes is based on the notion of the "link" between ontology entities. By observing a "neighbourhood" of entities it is possible to discover more complex than elementary ontology changes shown in Table 1. For example, by considering the neighbourhood of the concept "*Person*" (see Figure 10), one composite change would be to extract its properties (i.e. "*hasFirstName*" and "*hasLastName*") into a new concept "*Name*" and to relate it to the original concept.

A part of the composite changes related to the inheritance link between concepts (i.e. to the concept hierarchy) is shown in Table 4. These changes coalesce several entities such as merging or grouping of a set of concepts into one concept, or they produce different entities through e.g. splitting of one concept into other concepts, etc. The graphical representation of these changes is depicted in Figure 11.

In addition to the changes related to the concept hierarchy (see Figure 11), the composite changes also include changes on the nature of the link (e.g. a subconcept relationship is replaced with the property domain/range relationship), extracting or inlining properties (e.g. the replacement of a property with the attributes of a referenced concept), the transformation

of a concept into an instance and vice versa, the movement/merging/grouping/splitting of properties, instances, etc. Note that the set of composite changes is also model-dependent since the definition of the neighbourhood depends on the underlying ontology model. However, the most typical composite changes such as move, group, merge etc., are general enough to be part of any ontology evolution system.

Table 4. Composite ontology changes related to the concept-concept links

Composite Change	Syntax & Semantics
Pull concept up	***PullConceptUp(c)*** Attach a concept *c* to the all parents of all its parents
Pull concept down	***PullConceptDown(c)*** Attach a concept *c* to the children of all its parents excluding itself
Group concepts	***GroupConcepts(c1,c2, newC)*** Create a common superconcept *newC* for the concepts *c1* and *c2* and transfer common properties to it
Split concept	***SplitConcept(c, newC1, newC2)*** Split a concept *c* into two concepts *newC1* and *newC2* and distribute properties and instances among them
Merge concepts	***MergeConcepts(c1, c2, newC)*** Replace concepts *c1* and *c2* with one concept *newC* and aggregate all properties and instances
Concept copy	***CopyConcept (c, newC)*** Duplicate a concept *c* with all its properties and directed instances by creating a new concept *newC* and attaching it to the all parents of *c*
Concept generalisation	***AddConceptGeneralisation(c, newC)*** Add a new concept *newC* between a given concept *c* and all its parents
Inheritance extension	***AddInteriorConcept(c1,c2, newC)*** Add a new concept *newC* and attach it to a concept *c1* as its parent and to a concept *c2* as its child
Concept specialisation	***AddConceptSpecialisation(c, newC)*** Add a new concept *newC* between a given concept *c* and all its children

Composite changes specify coarse-grained changes. They are more powerful since an ontology engineer does not need to go through every step of the sequence of basic changes to achieve the desired effect. For example, for an ontology engineer it may be more useful to know that a concept was moved from one place in the hierarchy to another than to know that it was detached from one concept and attached to another concept. Therefore, composite changes make the ontology evolution much easier, faster and more efficient since they correspond to the one "conceptual" operation that someone wants to apply without understanding the details (i.e. a set of elementary changes) that an evolution system has to perform.

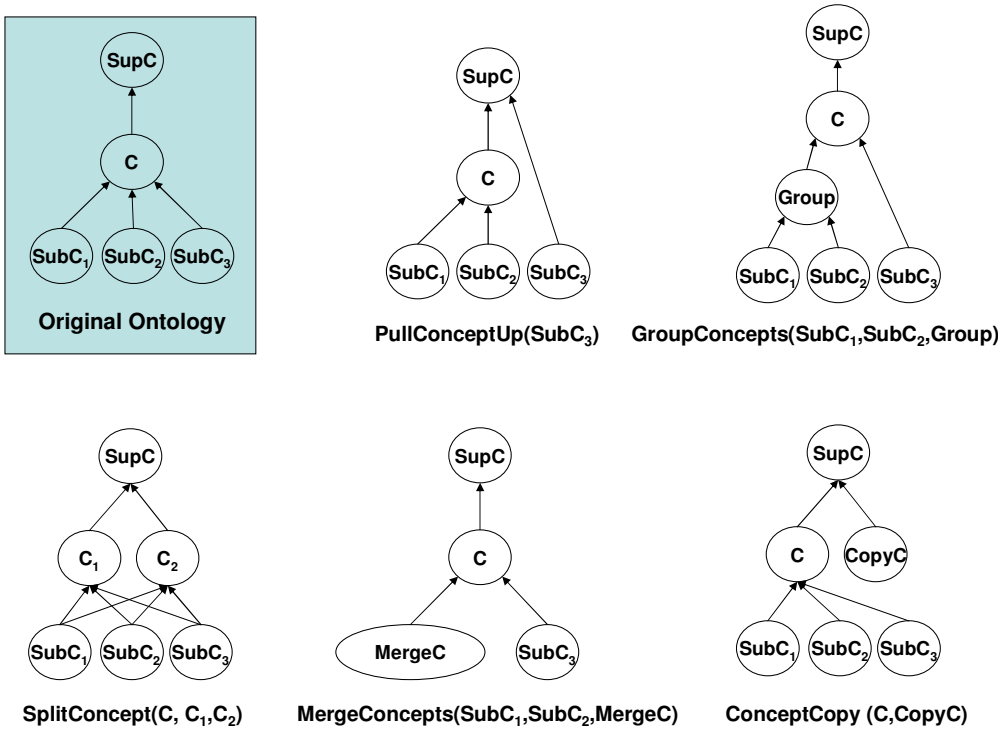


Figure 11. Some composite changes related to a concept hierarchy

Furthermore, composite changes often have more meaningful semantics [55]. For example, the semantics of moving the concept from one parent concept to another is clearly different from (i) the semantics of the removal and the addition of the subconcept relation as well as from (ii) the semantics of the addition and the removal of the subconcept relation. The "move" as a composite change maintains the identifiers of a subconcept relation and preserves some properties and instances. On the other hand, the removal and the subsequent addition or the addition followed by the removal create a new identifier for the subconcept relation. Finally, the removal and the later addition cause the loss of much information (e.g. at the instance level).

The previous example shows that a pure composition of the elementary changes is not powerful enough to express a sufficiently large class of transformations. To achieve the desired update, the glue "logic" (i.e. composite changes) has to be employed.

Note that the set of (elementary and composite) ontology changes does not include all the transformations that might be desired in the future. The set of elementary changes is exhaustive since it is derived from the underlying ontology model. However, it is not feasible to specify the complete set of composite changes. There is always a possibility to define a new composite change by combining existing ontology changes. For example, the grouping of two subconcepts into one common parent may be extended into grouping of n subconcepts (n $\geq$2). Further, this change can be spread out in several ways. It can either be broadened into the grouping of all subconcepts or narrowed down into grouping of subconcepts, which have common properties or instances. Finally, some extensions may be domain-specific (e.g. grouping of subconcepts, which are parents of the concrete instance). Thus, the set of ontology changes covers only a fixed set of well known (i.e. most frequently used) ontology changes.

Therefore, the next level of the abstraction of ontology changes includes complex changes. It encompasses all "real-life" changes not included in the elementary and composite changes. Complex changes are very important as they raise the level of abstraction as well as reusability. They are a prerequisite for an effective evolution system [108]. However, while a set of complex changes may never be complete (there can always be some other combination of changes that would constitute a meaningful single modification in some settings), it is not possible to enumerate them. Rather, we identify some of them that are frequently used. For example, the set of complex changes include changes such as:

- move concept, which is generalisation of the *PullConceptUp* (or *PullConceptDown*) composite changes since it attaches one concept to another arbitrary concept not necessary related to the first concept through the inheritance relationship;
- move a set of sibling concepts to a different location, which move two or more concepts that are siblings in the concept hierarchy to the same new location in the concept hierarchy (i.e. they remain siblings, but under different parent);
- deep concept copy, which recursively apply "shallow" copy (i.e. the *CopyConcept* composite change) to all subconcepts of a considered concept, etc.

Definition 19 A *complex change* is an ontology change that can be decomposed into any combination of at least two elementary and composite ontology changes.

Ontology changes, independently of the level of their complexity (see Figure 12), are the force that triggers off the ontology evolution. A richer set of available ontology changes makes it easier for an ontology engineer to formalise a request. As depicted in Figure 9, the role of *the change representation phase* of the ontology evolution process is to map a request for a change into one or more ontology changes. Obviously, selecting only one change from a predefined set of changes enables us to accomplish the change representation task efficiently. On the contrary, a much greater effort is needed to formalise a request as a sequence of changes since it requires deep knowledge about resolution of ontology changes (more details are given in chapter 4).

Above mentioned changes are represented as instances of an evolution ontology – a special ontology which explicitly represents semantic information about ontology entities, changes in the ontology and mechanisms to discover and resolve changes. This formal, explicit representation of ontology changes makes them machine-understandable, usable by other ontology evolution systems as well as exploitable for supplementary functionality of an ontology evolution system such as learnability. A detailed discussion of this ontology is given in section 3.2.4.

3.2.2 Semantics of Change

Application of an elementary change to an ontology can induce inconsistencies in other parts of the ontology. We distinguish structural and semantic inconsistency. Structural inconsistency arises when the ontology model constraints (see section 2.3) are invalidated (e.g. undefined entities at the ontology or instance level are used). Semantic inconsistency arises when the meaning of an ontology entity is changed due to the changes performed in the ontology [142].

For example, let's consider the case where an ontology engineer wants to delete the subconcept relation between the concept "*Person*" and the concept "*Student*". This change will generate an inconsistency related to the concept "*Student*" since it does not have a parent

concept any more. Further, it will generate inconsistencies of instances of the concept "*Student*" or its subconcepts since they inherit all the properties whose domain or range is the concept "*Person*". For example, after breaking the inheritance relationship between the concept "*Student*" and the concept "*Person*", the concept "*BSc Student*" will no longer be in the domain of properties "*hasFirstName*", "*hasLastName*" and in the range of the property "*includes*". Therefore, instances of the concept "*BSc Student*" may no longer have these properties. As this example illustrates, the removal of a subconcept relation results in structural inconsistency [53].

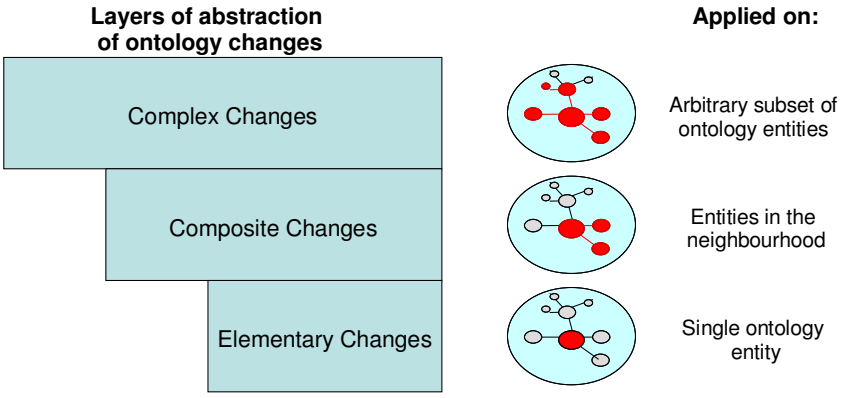


Figure 12. Different layers of abstraction of ontology changes and their impact on the ontology

Resolving that problem is treated as a request for a new change in the ontology, which can induce new problems that cause new changes and so on. Therefore, one change can potentially trigger other changes and so on. If an ontology is large, it may be difficult to comprehend fully the extent and meaning of each induced change. The task of *the semantics of change phase* is to enable the resolution of induced changes in a systematic manner, ensuring consistency of the whole ontology. To help a better understanding of the effects of each change, this phase should contribute maximum transparency providing a detailed insight into each change being performed. Some mechanisms used in this phase are described in chapter 4.

In the course of evolution, the actual meaning of concepts often shifts to represent the structure of the real world better. This causes semantic inconsistency. While some shifts of the concept meaning are performed explicitly, a meaning of a concept can sometimes shift implicitly through changes in other parts of the ontology. For example, consider an ontology describing a relationship between jaguars and persons represented in Figure 13.

In this ontology the meaning of the concept "*Jaguar*" is clear through the existence of the property "*eats*" that links "*Jaguar*" and "*Person*" – it is obvious that the concept of "*Jaguar*" stands for an animal from the feline family. For any reason one may delete the concept "*Person*", which may result in the removal of the property "*eats*" as well. After the change is performed, the semantics of the concept of "*Jaguar*" is not clear any more – is it a *Jaguar* cat or a *Jaguar* car? Note that since an ontology would still be well formed, there is no way to automatically detect the problem. As a result, false conclusions may be drawn from this ontology.

Figure 13. Concept properties define its meaning

These kinds of ambiguities can be eliminated in several ways. The simplest solution is by introducing a superconcept "*Animal*" before the change is performed. However, if the ontology is large, such issues may be easily overlooked because it is very hard to keep the complete ontology structure in mind at once.

This problem can be avoided by using a richer description determining semantic role of ontology entities. By attaching meta-information about e.g. essential properties of a concept [49], deeper knowledge about concept meaning is provided. For example, by specifying that the property "*eats*" is the essential property for the concept "*Jaguar*" (since it defines the meaning of the concept), its removal is not allowed. Consequently, the ontology engineer has to be informed about all changes that will cause this deletion.

Moreover, semantic ambiguities of ontology entities may be resolved through additional documentation, such as who the author of an entity is, what the purpose of introducing an entity is, how frequently an entity may be changed [142] etc. Contrary to meta-information determining the semantic role of ontology entities, "documentation" meta-information cannot be used for formal consistency checking. However, they may help ontology engineer to comprehend how to perform the evolution.

In the rest of this thesis (see chapter 4) we focus on the structural inconsistency due to possibilities to assist ontology engineers in detecting and resolving inconsistencies. On the other hand, resolving semantic inconsistency heavily depends on the precise semantic representation of ontology entities. Therefore, the standard ontology model has to be enriched with the semantic information that exactly characterises the concept's properties and expected ambiguities, including the properties that are prototypical for a concept and that are exceptional or essential as well as the behaviour of a properties over time and the degree of applicability of properties to subconcepts [49], [142]. Since meta-information about ontology entities are recognised to be the necessary and sufficient condition, i.e. the means for obtaining a semantically consistent ontology, the semantic inconsistency problem is not of the interest for further elaboration.

3.2.3 Change Propagation

The task of *the change propagation phase* of the ontology evolution process (see Figure 9) is to bring automatically all dependent artefacts into a consistent state after an ontology update has been performed. As shown in Figure 14, an ontology change might corrupt the dependent ontologies, instances, as well as application programs running against the ontology [66]. Note that other aspects of the change propagation represented through the "mightHaveConsequence" link in Figure 14 are discussed in section 3.4.

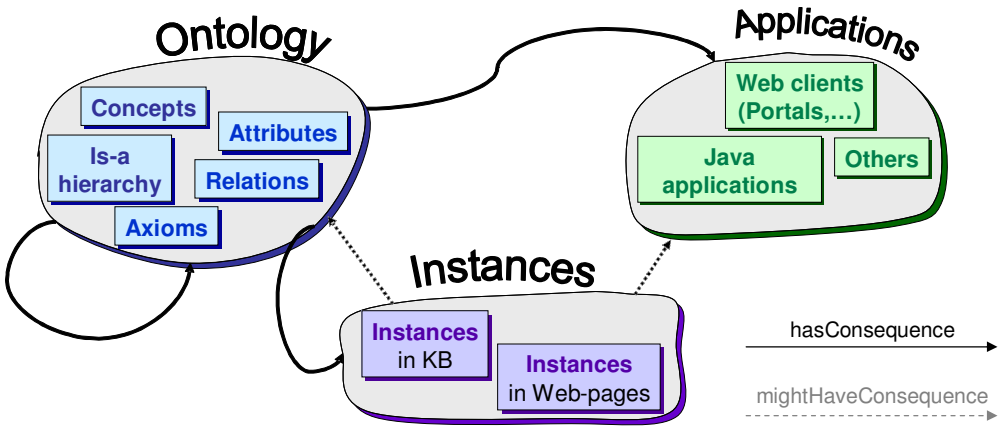


Figure 14. Consequences of an ontology change

Effect of Changes on the Dependent Ontologies

Ontologies often reuse and extend other ontologies. Therefore, an ontology update might also corrupt ontologies depending on the modified ontology (through the inclusion, mapping integration, etc. [72], [107]) and consequently, all the artefacts based on these ontologies. This problem can be solved by recursive applying the ontology evolution process to these ontologies. However, it also requires the methods for the synchronisation between dependent ontologies due to their independency. They are elaborated in section 5.3.

Further, in addition to the structural inconsistency, the semantic inconsistency can also arise when, for example, the dependent ontology already contains a concept that is added in the original ontology. Returning to the example shown in Figure 5, the dependent ontology *PO* (the project ontology) may include the concept "*das Projekt*", although the concept with the same meaning (the "*Project*" concept) is previously defined in the included basic ontology *BO*. Since the resolution of that problem requires establishing equivalencies between the "*das Projekt*" and "*Project*" concepts, it is more related to the meta modelling than to the ontology evolution.

Effect of Changes on the Ontology Instances

When the ontology is modified, ontology instances need to be changed to preserve consistency with the ontology [53]. Two cases may be distinguished:

1. *Metadata evolution* - instances are distributed over the Web, e.g. web pages are annotated with the ontology instances;
2. *Knowledge base evolution* – instances are organised in an instance pool.

The first case (i.e. *the metadata evolution*) requires means for the continuous adaptation of the annotated information to the new semantic terminology and relationships. The resolution of this case can be performed in three steps. Since the instances are on the Web, as for example in the case of MEDLINE (see section 5.4), they have to be collected in a temporary ontology [28]. In order to speed up the whole acquisition process, only the instances that may depend on a change have to be gathered [28]. For example, if the concept "*Student*" is deleted, only

instances of that concept and its subconcepts should be assembled. The output of this step is a list of instances together with the reference to the web documents containing them.

The next step is reduced to the evolution of dependent ontologies (see chapter 5) since the temporary ontology is a dependent ontology consisting of only ontology instances. It must be transformed to confirm to the modified ontology. This step provides an output in the form of a list of modified instances with the reference to the corresponding web resource. Note that this is the only step performed for *the knowledge base evolution*. In that case, the instances already form an instance pool that is treated as a dependent ontology.

In the last step "out-of-date" instances on the Web are replaced with corresponding "up-to-date" instances. Some updates can be done automatically, but for the instances that are "write-protected" the notification has to be sent to the author of the annotation in order to inform her about the changes and to suggest how to correct the instance [127], [130]. Therefore, *the metadata evolution* does not resolve all the problems. However, it provides guidelines, such as suggestions which resources' metadata has to be checked and eventually changed to run again the modified ontology.

Effect of Changes on the Applications

When an ontology is changed, applications based on the changed ontology may not work correctly. An ontology evolution system has to recognise which change in the ontology can affect the functionality of dependent applications and to react correspondingly [55].

Most of the applications are written to be as generic as possible. However, there is a certain number of "hard-coded" elements that should be treated specially in some way [66]. In most of the semantic web portals, there is a set of predefined queries. They are "hard-coded" into the services that are invoked as a response to the specific action. When an entity from a query is removed, the query rewriting process is needed [36]. Since a query is a part of an internal model that may become incompatible with the ontology, the portal (i.e. software) evolution is needed.

Therefore, the first problem is how to find an application that uses the changed ontology since there is a lack of the connection from the high level modelling methods to the lower level implementation methods. An application can be semi-automatic maintained only if there is metadata describing which ontology and/or which ontology entities that application uses. Thus, the annotation of applications is necessary [127]. By connecting applications back to the model, it is possible to build up a fully traceable model from the broad process outline to the software artefacts actually being constructed.

Secondly, in order to avoid overhead, which may heavily increase if the application modification is performed every time an ontology is modified, the categorisation of the ontology changes from the application point of view is required [55]:

- *ontology-extending changes* – a new entity never has an impact on the existing application;
- *ontology-modifying changes* which cover:
    - *compilation-safe changes* – an application does not use a changed entity in its code, and,
    - *compilation-unsafe changes* – an application accesses a changed entity in its code.

Therefore, ontology-extending and compilation-safe changes do not require any application modification. On the contrary, the compilation-unsafe changes demand the application modification and recompilation since the application becomes obsolete and may produce incorrect results.

3.2.4 Change Implementation

The role of *the change implementation phase* of the ontology evolution process is (i) to inform an ontology engineer about all consequences of a change request, (ii) to apply all the (required and derived) changes and (iii) to keep track about performed changes. Subsequently, we describe these functionalities in detail.

Notification of the Consequences of a Change

In order to avoid performing undesired changes, before applying a change to an ontology, a list of all implications (i.e. required and derived changes) to the ontology and dependent artefacts should be generated and presented to an ontology engineer who modifies this ontology [140]. Only if the ontology engineer is informed about all the changes that are going to be performed on a request, can she make strategy decisions posed by the system. The ontology engineer should however have possibilities to make such choices or even to abort the entire ontology evolution process when she realises that it would have undesired consequences for other parts of the ontology, for dependent ontologies, for distributed instances or for existing applications. Consequently, she should be able to comprehend a list of all the changes and approve or cancel them. When the changes are approved, they are performed by successively resolving changes from the list. If changes are cancelled, the ontology should remain intact. This is more elaborated in description of implementation in chapter 7.

Change Application

In order to give an ontology engineer a chance to cancel a change after it has been completely analysed, it is necessary to separate the analysis[22] of the user's request for the change from the final execution of this request within the ontology evolution system. Therefore, the main task of *the change implementation phase* of our ontology evolution process is the application of changes. During this phase all changes (i.e. required and derived changes) are applied to a consistent ontology and result into a new consistent state of this ontology.

Indeed, one of the main advantages of our ontology evolution process is the separation of the phases where ontology evolution requests are analysed (i.e. *the semantics of change* and *the change propagation phases*) from the final execution of the changes (i.e. *the change implementation phase*). This separation was naturally driven by the need for the transaction[23]. Only after a successful commitment of the hypothetical "reasoning" performed by *the semantics of change* and *the change propagation phases*, the changes in effect took place on the ontology itself. Once acknowledged by the ontology engineer for the implementation, all

[22] The analysis of a change covers the semantics of change and the change propagation phases of the ontology evolution process where the change is extended with the additional changes that ensure the consistency of the ontology itself and the dependent artifacts.

[23] A transaction represents a sequence of actions that is treated as a unit for the purposes of satisfying a request. For a transaction to be completed, it has to be accomplished in its entirety.

the changes are considered as an atomic ontology "transaction" (i.e. they act like a transaction), although the changes are executed step by step. Different optimisation methods used for performing this task are discussed in chapter 7.

Change Logging

The last task of *the change implementation phase* is to keep track about the performed changes. Information about changes can be represented in many different ways (e.g. [91], [100]). To communicate about changes, we need a common understanding of a change model and of a log model. Therefore, we introduce the evolution ontology and the evolution log [74]. The evolution ontology is a model of ontology changes enabling better management of these changes. The evolution log tracks the history of applied ontology changes as an order sequence of information (defined through the evolution ontology) about particular change.

Figure 15 explains the dependencies between these ontologies and the ontology that is changing (c.f. the domain ontology). There is a clear distinction between general knowledge about the ontology evolution (i.e. the evolution ontology), the knowledge that is specific to a particular domain (i.e. the domain ontology) and the knowledge about the development/maintenance of a model of that domain (i.e. the evolution log).

Note that only a common understanding of the changes (achieved through the evolution ontology) and of a log (achieved through the evolution log) enables the synchronisation between the evolving domain ontology and the dependent artefacts (e.g. applications based on this ontology) that have to incorporate or adapt to those changes. Further, the evolution log based on the formal model of ontology changes (i.e. on the evolution ontology) enables the recovery from "failure" since it makes possible to undo and redo applied changes as needed.

Subsequently, we describe these two introduced ontologies in detail.

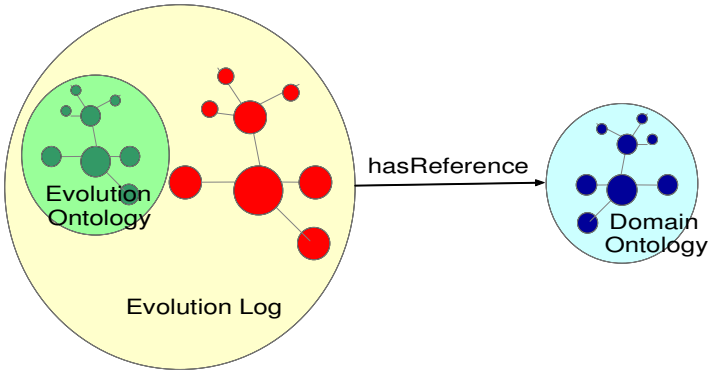


Figure 15. Dependency between domain ontology, evolution ontology and evolution log

Evolution Ontology

In order to have the explicit representation of changes, we need an agreed-upon ontology of changes. Therefore, we develop a special ontology, the so-called evolution ontology that supports, alleviates and automates the ontology evolution process [74], [127]. It is about a meta-ontology that is used as a backbone for creating evolution logs. This ontology is a central part of our ontology evolution process, because different process phases must agree on representing changes. Thus, to develop this common shared model of ontology changes, we incorporate the requirements put by all the phases of the ontology evolution process.

The evolution ontology, shown in Figure 16, models what changes, why, when, by whom and how are performed in an ontology. Therefore, the most important concept is the "*Change*" concept. The structure of the hierarchy of ontology changes reflects the underlying ontology model by including all possible types of changes. For example, elementary changes are decomposed into the changes causing the addition (the concept "*AdditiveChange*") and the changes provoking deletion (the concept "*SubtractiveChange*"). The additive changes are further decomposed into e.g. "*AddConcept*", "*AddProperty*", etc. Indeed, each leaf concept in the hierarchy of the concept "*Change*" represents a specific ontology change (see e.g. Table 1 and Table 4). For the composite changes the property "*includes*" and the set of its subproperties are defined indicating that these changes are realised as a sequence of elementary or composite changes. This sequence is represented in the evolution ontology explicitly, enabling the declarative specification of the semantics of the composite changes. This richer description of the changes, their causes, and consequences provide more scope to resolve possible inconsistency.

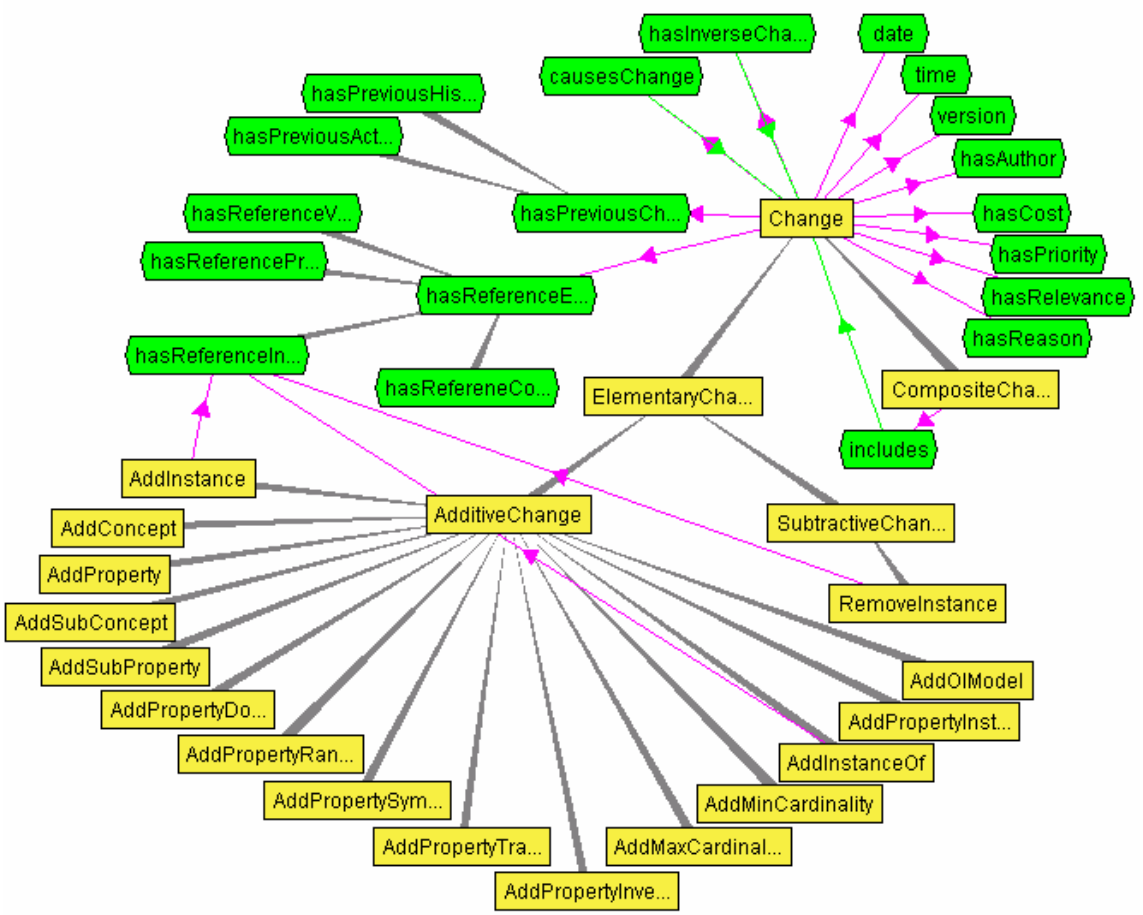


Figure 16. A part of the evolution ontology

Representing the knowledge about ontology changes in a concept hierarchy enables us to specify the common properties of ontology changes in an efficient way since the inheritance mechanism may be exploited [64]. For example, for each change the additional information, such as the date and time of the change, the version number, as well as the identity of the change initiator may be associated. This is modelled through appropriate properties defined

for the concept "*Change*" and can serve as a source for different knowledge discovery methods, e.g. mining about change trends [74].

Moreover, information supporting decision-making, such as cost, relevance, priority, textual description of the reason for a change etc. may also be included [91]. The cost of a change is used to determine its validity especially for the deletion of some ontology entities. The relevance of a change describes whether and how it can fulfil the requirements. For example, the cost of the concept deletion is estimated based on the ontology structure (e.g. the number of the subconcepts and the total number of the instances of these concepts). If the cost of a change application is huge and the relevance of a change is low, then this change should be cancelled. This information might be used to guide ontology engineers through the evolution process [140]. Further, in order to keep a memory of past decisions, it may be important to record the decisions leading to updates [67]. The "*reason*" property is used to explain the motivation for making a change. For instance, an ontology engineer should give an explanation or an example why a certain concept should be introduced or its definition was changed, etc. This information may be used for learning about decision making process as well as about the competencies of ontology engineers.

As described previously, elementary changes may cause new changes in order to keep the ontology consistent. Such dependencies may be represented using the "*causesChange*" property. Groups of changes of one request are maintained in a linked list using the "*hasPreviousChange*" property. These two properties have completely different semantics. The first one is used to model the "cause-effects" dependency between changes whereas the second one models the order in which the changes are requested. However, both of them are necessary for supporting the reversibility as well as the change propagation. The property "*causesChange*" makes it possible to reverse all side effects of the required change. The property "*hasPreviousChange*" allows reconstructing the sequence of required changes and is specialised into two subproperties: "*hasPreviousActualChange*" and "*hasPreviousHistoryChange*". The "*hasPreviousActualChange*" property indicates the current state of the ontology by excluding the effects of the inverse changes (i.e. reversibility). In this way, it specifies only the necessary changes to achieve the resulting ontology. On the contrary, the "*hasPreviousHistoryChange*" property takes into account the sequence of all the changes that have actually taken place. Therefore, it models the actual evolution process in a unique way (since it records all intermediate ontology versions as well). In the case that a log of changes does not contain changes that undo other changes (e.g. *AddConcept("X")* followed by *RemoveConcept("X")*), the "*hasPreviousActualChange*" and "*hasPreviousHistoryChange*" properties for each change instance point to the same target instance.

For properties establishing the relations between changes (i.e. "*causesChange*", "*hasPreviousActualChange*" and "*hasPreviousHistoryChange*") the corresponding inverse properties are defined enabling the usage of implicit knowledge for some purposes.

The above-introduced properties are universal in the sense that they can be recorded for all ontology changes. However, there are additional properties depending on the change type. These properties are used to represent the peculiarities of a particular type of a change, such as its arguments. The change-specific properties are modelled by using a property hierarchy. For example, entities from the ontology being changed are related to the instances of the "*Change*" concept through the "*hasReferenceEntity*" property and its subproperties. The number of properties referencing the domain entities defined for one ontology change and their semantics are specific to that change. Therefore, the property "*hasReferenceEntity*" is specialised into several subproperties indicating the type of entity that is considered. For example, the "*hasReferenceConcept*" property is used to reference the changes related to the concepts in the domain ontology or the "*hasReferenceValue*" designates the changes to the

values of property instances. This information is used for the verification of performed ontology changes by checking the type of the entity that the change is applied to. Note that these properties are shared between several concepts in the hierarchy of the concept "*Change*". For example, the property "*hasReferenceInstance*" is specified for changes "*AddInstance*", "*RemoveInstance*", "*AddInstanceOf*", etc. since all of them operate on ontology instances.

We note that the "*hasReferenceEntity*" property (and its subproperties as well) is not defined as a relation since it is also used to reference the entities from the domain ontology that do not exist anymore (e.g. "*RemoveConcept*"). Therefore, it is an attribute whereas its value is the unique identifier of the entity from the domain ontology that is changed. This is the reason why the evolution log does not include the domain ontology, but rather refers to it as shown in Figure 15.

Note that the hierarchy of the concept "*Change*" invariably depends on the underlying ontology model. Defining the standard set would be realistic when there is a common ontology language. However, the set of properties defined for the concept "*Change*" and its subconcepts are general enough to be considered as a standard for the change representation.

The evolution ontology enables the representation, the analysis, the reasoning about, the realisation and the sharing of ontological changes in a more systematic and consistent way. Benefits of the using the evolution ontology are manifold:

- Changes are represented formally and can be managed formally;
- A history of changes is stored and can be used for the recovery or the additional analysis;
- Based on the formal representation and the history of changes the change propagation problem (see chapter 5) may be approached (e.g. any software application that makes use of an evolving ontology would be able to recognise applied changes and to incorporate them easily);
- Using the same representation model for the domain ontology and ontology changes simplifies storage and allows reuse of system components (e.g. searching for entities from the domain ontology can be reused for the searching for the applied changes since they are entities of the evolution ontology).

Evolution Log

While the structure of the evolution ontology aims at better characterisation and understanding of the changes, an evolution log records an exact sequence of changes that occurred when an ontology engineer updated an ontology. Therefore, it contains instances of subconcepts of the concept "*Change*", which include the elementary as well as the composite ontology changes. Indeed, a request for a change and all its consequences are represented as instances of the corresponding change concepts. Each instance contains a data about a particular change. For example, for all changes a log includes a timestamp, author, version etc. Figure 17 shows a part of a possible log of changes. In this way the level of granularity at which changes are specified is close to a single user-interface operation (cf. *AddConcept("Person")*). However, an evolution log provides a complete and unambiguous change specification at a very fine level of detail since for each change request all effects are represented explicitly (cf. *AddSubConcept("Person", "Root")*).

```
<a:AddConcept rdf:ID="i-1079525222689-178059496"
   a:hasReferenceConcept="file:/C:/BasicOntology#Person“
   ...
   a:version="1"/>
<a:AddSubConcept rdf:ID="i-1079525222689-973090869"
   a:has_referenceSubConcept="file:/C:/BasicOntology#Person"
   a:has_referenceSuperConcept=http://kaon.semanticweb.org/2001/11/kaon-lexical#Root
   ...
   a:version="1">
   <a:hasPreviousActualChange rdf:resource="#i-1079525222689-178059496"/>
   <a:hasPreviousHistoryChange rdf:resource="#i-1079525222689-178059496"/>
</a:AddSubConcept>
```

Figure 17. A part of the evolution log represented in XML/RDF format

Keeping a record of changes is important. Introducing an ontology for representing them gives additional advantages. The advantage of our evolution log is the formal, explicit semantics that is provided through the evolution ontology. This enables easier synchronisation between dependent ontologies. Namely, this problem can be approached by comparing the evolution logs, which is not difficult due to the fact that logs reuse the same evolution ontology. Further, if the formal model for representing changes is used, the developers of applications depending on the ontologies will be well served since they will devote less time and expense to understand and manage changes due to possibility to automate the update [101]. Finally, the meta knowledge provided by the evolution ontology can assist intelligent search in the evolution log for some previously made changes. For example, if a query (e.g. “Find all “*RemoveConcept*” changes”) sends back no result, then the evolution ontology can be used to generalise automatically the query to find nearest partial matches (e.g. “Find all “*SubtractiveChange*” changes”). Indeed, reasoning allows inferring implicitly represented knowledge from the knowledge that is explicitly contained in the evolution ontology.

3.3 Guidance Requirement

The guidance requirement is related to the user’s management of changes. It considers the needs of different groups of users since it ensures that the ontology evolution system is usable for both beginners and experts in modifying an ontology.

As mentioned above, a change in one part of the ontology can have far reaching consequences for other parts of the ontology and dependent artefacts. *The semantics of changes phase* (see section 3.2.2) was introduced into the evolution process to help the ontology engineers comprehend the effect of a change. This can help in reducing the number of accidental ontology changes and can even guide the ontology refinement process. Still, there are numerous circumstances where it may be desired to reverse the effects of the ontology evolution, to name just a few:

- The ontology engineer may fail to understand the actual effect of the change and approve the change that should not be performed;
- It may be desired to change the ontology for experimental purposes;
- When working on an ontology collaboratively, different ontology engineers may have different ideas about how the ontology should be changed.

In order to enable recovering from these situations, we introduce *the change validation phase* in the ontology evolution process (see section 3.5). It enables justification of performed changes and undoing them at user's request. Consequently, the usability of the ontology evolution system is increased.

It is important to note that reversibility means undoing all effects of some change, which may not be the same as simply requesting an inverse change manually. For example, if a concept is deleted from a concept hierarchy, its subconcepts will need to be deleted as well, attached to the root concept, or attached to the parent of the deleted concept. Reversing such a change is not equal to recreating the deleted concept – one needs, also, to revert the concept hierarchy into original state.

The problem of reversibility is typically solved by creating evolution logs (see 3.2.4). An evolution log tracks information about each change in the system, allowing the reconstruction of the sequence of changes leading to current state of the ontology. Therefore, supporting traceability[24] ensures that an ontology engineer can always get out, go back or undo a change.

Moreover, *the change validation phase* helps ontology engineers to find out whether they have built the right ontology, i.e. whether the changed ontology represents a piece of reality and the users' and/or application's requirements correctly. One technique for the validation is the generation of the explanation. Therefore, in contrast to *the semantics of change phase* that concerns formal properties of a model such as consistency and syntactical correctness, *the change validation phase* requires the "behaviour knowledge", because if we do not have some expectation of the appropriate behaviour, we cannot assess if the runtime behaviour is adequate. Indeed, *the change validation phase* provides answers to the following questions: how, why, what, what-if etc. More information about the assessment of an ontology based on the end-users' behaviour is given in chapter 6.

Besides reversibility and explanation, the guidance requirement is also related to the usability of an ontology evolution system. Indeed, *the change validation phase* realising the guidance requirement also has to provide the following set of functionalities:

- ensuring that an ontology engineer always knows what she can and should do next, and what will happen when she does it;
- making sure ontology engineers can move in a step by step manner as they perform their tasks;
- guarantying that the sequences of changes to achieve a request are as simple as possible;
- making sure that everything that appears on the screen should be easily understandable to ontology engineers, organised effectively, without displaying too much information;
- providing good feedback including effective error messages in the case that the request cannot be applied;
- when something goes wrong, explaining the situation in adequate detail and help the user resolve the problem;
- ensuring that ontology engineers always feel in control e.g. by informing them of the progress of the changes and of their location as they navigate.

[24] Traceability is the ability to determine the information that leads to a decision being made.

3.4 Refinement Requirement

Refinement requirement enables the continual improvement of an ontology. The refinement patterns serve to direct the ontology engineers' focus onto a set of issues that might be insightful. It is realised through *the change capturing phase* of the ontology evolution process (see section 3.5)

In the ontology evolution we may distinguish two types of changes: top-down changes and bottom-up changes, whose generation is the task of *the change capturing phase*. *The top-down changes* are *explicit* changes, driven, for example, by ontology engineers who want to adapt the ontology to new requirements or by the end-users who provide the explicit feedback about the usability of ontology entities. These changes cover the business strategy evolution, the modification in the application domain, new user's needs, additional functionality, etc. and they are captured in a variety of ways such as the direct discussion or interviews, by considering customer specifications, by conducting some surveys, etc.

However, the gathering of these changes is a time consuming, error-prone and difficult activity due to many reasons: (i) ontology engineers may have difficulty grasping the knowledge area; (ii) the end-users may have difficulty expressing their needs; (iii) some relevant changes may be overlooked; (iv) some irrelevant changes may be required, etc. Moreover, some changes in the domain are *implicit*, reflected in the behaviour of the system and can be discovered only through the analysis of this behaviour. For example, if nobody is interested in information covered by one ontology concept for a longer period of time, it might indicate that this concept is not necessary and, consequently, it could be removed. These changes mined from different datasets are called *the bottom-up changes*. The application of these changes enables the continual improvement of the ontology. The different sources of both the types of changes are shown in Figure 18.

Note that these types of changes correspond to the two methods for the knowledge acquisition[25]. The *top-down (deductive) changes* are the result of the *knowledge elicitation* techniques that are used to acquire knowledge direct from human experts (domain experts or end-users). On the other hand, the *bottom-up (inductive) changes* match the *machine learning*[26] techniques, which use the different methods to infer patterns from the sets of examples.

One source of the bottom-up changes is the structure of the ontology itself [95]. Indeed, the previously described *change validation phase* results in an ontology which may be in a consistent state, but contains some redundant entities or can be better structured with respect to the domain. For example, multiple users may be working on different parts of an ontology without enough communication. They may be deleting subconcepts of a common concept at different points in time to fulfil their immediate needs. As a result, it may happen that only one subconcept is left. Since classification with only one subclass beats the original purpose of classification, we consider such ontology to have a suboptimal structure. To help users in detecting such situations, we investigated the possibilities of applying the self-adaptive systems principles [59] and proactively make suggestions for *ontology refinements* – changes to the ontology aimed at improving ontology structure, making the ontology easier to understand and cheaper to modify. As known to the authors, none of the existing systems for ontology development and maintenance offer support for (semi-) automatic ontology improvement. However, this support is very important since one method for reducing the cost of the evolution is to automate aspects of the evolutionary cycle when possible [145].

[25] Knowledge acquisition is a subfield of the Artificial Intelligence (AI) concerned with eliciting and representing knowledge of human experts so that it can later be used in some application.

[26] Machine learning provides techniques for extracting knowledge (e.g. concepts, rules) from data.

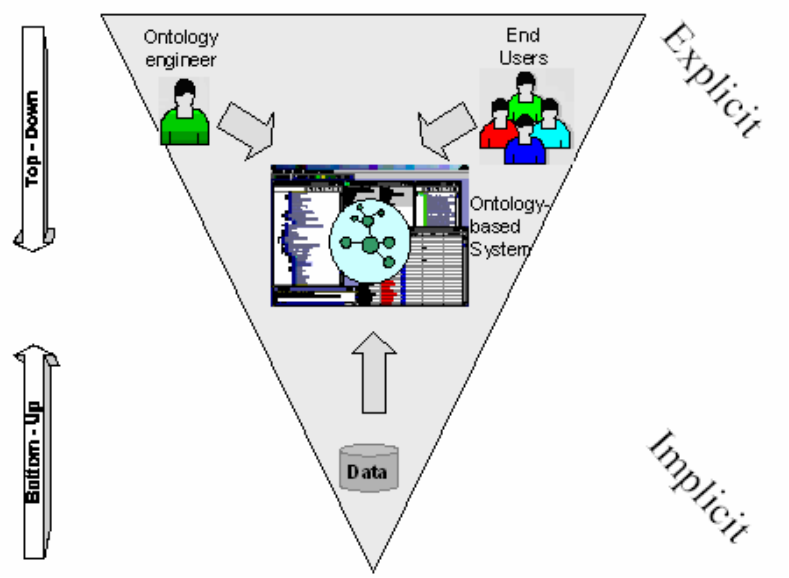


Figure 18. Two types of ontology changes

Based on heuristics knowledge and/or data mining algorithms, suggestions for changes that refine the ontology structure may be induced by the analysis of the following data sources: (i) the ontology structure itself, (ii) the ontology instances or (iii) the information describing patterns of ontology usage. In the rest of this section we further elaborate on these variants.

3.4.1 Structure-driven Change Discovery

Structure-driven change discovery is based on the knowledge of ontology engineers that they use in the decision making during the ontology evolution. It exploits a set of heuristics to improve an ontology based on the analysis of its structure. We found the following heuristics:

- If all subconcepts have the same property, the property may be moved to the parent concept;
- A concept with a single subconcept should be merged with its subconcept;
- If there are more than a dozen subconcepts for a concept, then an additional layer in the concept hierarchy may be necessary;
- All the siblings in the concept hierarchy must be at the same level of generality;
- A concept without properties is a candidate for deletion;
- If a direct parent of a concept can be achieved through a non-direct path, then the direct link should be deleted.

The proposed heuristics model the expertise and experience of the ontology engineers in order to give advice on strategic matters such as the coherence of an ontology. Indeed, the structure-driven change discovery stems from the idea to apply refactoring[27] [145]. Refactoring interpreted from the point of view of the ontology evolution enables the evolution of an ontology on an if-needed basis. In such a way it reduces unnecessary complexity and

[27] Refactoring is a technique to restructure code in a disciplined way.

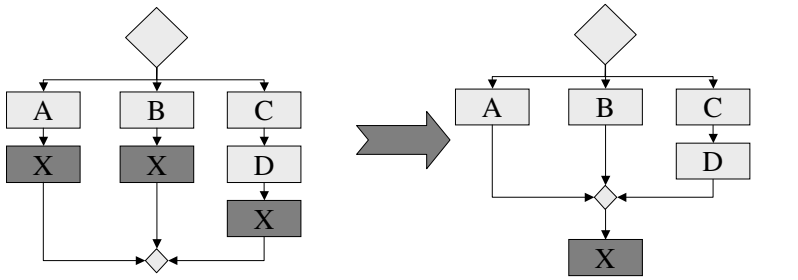
inefficiency. For example, the first heuristic is the consequence of the *"Consolidate Duplicate Conditional Fragments"*[28] refactoring method [38], which states:

"*When the same fragment of code is in all branches of a conditional expression, then it has to be moved outside of the expression*".

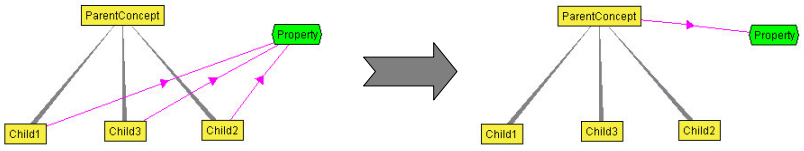
If we consider a fragment of a code as a property and a branch of a conditional expression as a subconcept, then the previous refactoring method can be translated into the request to move a property outside the subconcepts, i.e. into a common parent concept. In other words, when defining a domain or range for a property, then the most general concept or concepts should be used. Indeed, we interpret this pattern in the following way:

- If all subconcepts of a concept have the same property, this property should be moved to the superconcept;
- If a concept and its single superconcept have the same property, this property should be moved to the superconcept;
- If most subconcepts of a concept have the same property, one has to consider moving it to the superconcept.

The dependency between refactoring methods and the ontology evolution is shown in Figure 19.



a) Refactoring



b) Structure-driven Change Discovery

Figure 19. The interpretation of the refactoring method from the ontology evolution point of view

3.4.2 Data-driven Change Discovery

The data-driven change discovery considers the ontology instances in order to refine the ontology (including its instances as well).

[28] http://www.refactoring.com/catalog/index.html

Note that this can not be realised in the ontology evolution systems based on the description logic (DL) ontology languages such as OWL since DL systems naturally adhere to the open world assumption. This assumes that the present instances are just an explicitly known subset of valid instances, and more valid instances may be inferred by reasoning [20]. Thus, if an assertion implies a deduced fact that is consistent with all known assertions and instances, then the fact is assumed to be true even if it is not present in the set of instances. Otherwise, an insertion in DL is always treated as the insertion of incomplete information. On the other hand, KAON as other frame-based ontology languages assumes the closed world. By the closed world assumption, the absence of information is interpreted as negative information. Therefore, it is assumed that the "complete information" about instances is available.

The difference between ontology-languages regarding the open or closed-world assumption provokes significant consequences on the data-driven change discovery, which takes into account ontology instances with the goal to improve the ontology. Namely, the lack of information in the DL makes it useless for the instance-based learning since it makes no sense to conclude something from incomplete information.

The data-driven change discovery[29] explores different techniques such as data mining (DM), formal concept analysis (FCA) or even with the heuristic knowledge on the ontology instances in order to support the task of maintaining ontologies. In the following text, we point out possible approaches.

DM is defined as the process of extracting hidden knowledge from large volumes of raw data [153]. If ontology instances are considered as raw data, then DM methods may be used to analyse them from different perspectives and summarise them into useful information. For example, clustering methods that group data items according to logical relationships may be used to improve a concept hierarchy. In [76] the authors propose an approach for clustering ontology-based metadata based on the definition of a set of similarity measures for comparing ontology-based metadata. Hierarchical clustering algorithms are preferable for the concept-based learning since they produce hierarchies of cluster. Therefore, the matching between these clusters and the existing concept hierarchy may result in a recommendation for the improvement of the concept hierarchy. Other data-mining methods may be applied [73], [77]. For example, the application of the association rules on ontology instances may result into new properties, the useful rules may be extracted by applying Inductive Logic Programming (ILP) methods etc.

Moreover, Formal Concept Analysis (FCA) [41] may be also used for adjusting the concept hierarchy based on the context of the instance pool. FCA is mainly used for the analysis of data, i.e. for investigating and processing explicitly given information. Such data are structured into units that are formal abstractions of concepts of human thought, allowing meaningful and comprehensible interpretation. These concepts are organised in the (mathematic) lattice structure, which guaranties the minimality of the generated structure (e.g. for each two concepts there is exactly one infinum and one supremum). FCA can be used for learning hierarchies from a set of instances in a step by step manner. It means that not only the most upper concept, but the whole hierarchy tree (i.e. a lattice) can be generated for a set of instances. The root of such a lattice is the most general concept (parent) of the given

[29] Ontology learning [73] exploits a lot of existing resources, like text, thesauri, dictionaries, databases and so on to develop an ontology in a semi-automatic way. It combines techniques of several research areas, e.g. from machine learning, information retrieval or agents and applies them to discover the semantics in the data and to make them explicit. Therefore, it starts from scratch.
On the contrary, the data-driven change discovery assumes that an ontology, that needs to be improved, already exists. This information is used to guide the learning process as background knowledge. Moreover, for the data-driven change discovery it is not enough to induce knowledge. It is also important how to interpret this knowledge with respect to the existing ontology. More information is given in chapter 6.

instances. In that way using FCA methods, one can learn the whole hierarchy and not only the most upper concept like by using the most common subsumer operator in DL. However, the structure of instances FCA is able to deal with is very limited, since FCA methods are not suitable for instances with instantiated relations.

Besides the application of data-mining methods and FCA on ontology instances, some recommendations for the ontology refinement may be generated by applying heuristic knowledge. The following heuristics are identified:

- A concept with no (direct and indirect) instances may probably be deleted;
- If no instance of a concept C uses any of the properties defined for C, but only properties inherited from the parent concept, then the concept C is not necessary;
- A concept with many instances is a candidate for being split into subconcepts and its instances distributed among newly generated concepts;
- If all instances of a concept X are also instances of some other concept Y (which may have additional instances as well), then the concept X should be a subconcept of the concept Y.

Finally, the data-driven change discovery might be used as a verification tool since it helps in finding "weak" parts in ontology instances. Since we assume that instances must be consistent with the underlying ontology, the "quality" of instances [131] can be assessed through the existence of redundancy, inaccurate or incomplete information in an instance pool (see section 5.4.3). To note that assessment is performed on the instance pool level and the ontology structure is the basis for all measures. The assessment can help an ontology engineer refine and improve the ontology instances. Therefore, the analysis of the instance pool results into a set of recommendations for the improvement of the ontology instances. This approach is applied to the MEDLINE system that is elaborated in section 5.4.3.

3.4.3 Usage-driven Change Discovery

One of the most difficult problems in ontology development is its evaluation since the ontology development is subjective. What does it mean for an ontology to be correct (objectively)? The best test is the application for which the ontology was designed, the behaviour of the end-users of that application. *The usage-driven change discovery* tries to support the ontology engineers in adapting an ontology with respect to the end-users' needs. For example, by tracking when the concept has last been retrieved by a query, it may be possible to discover that some concepts are out of date and should be deleted or updated. Chapter 6 presents our approach for the usage-driven change discovery based on the usage-specific heuristic knowledge. We model information about the usage of ontology entities and process heuristic knowledge in order to generate expert advice.

The broad scope of the supported changes (each of them related to one or more heuristics) indicates that the change discovery might have a significant impact when applied to evolving ontologies. This claim is validated with the real application (see chapter 6) in which many hand-coded changes between two ontology versions are automated. Additionally, the set of heuristics defined to support the discovery of changes guide the learnability, i.e. the speed with which an inexperienced ontology engineer can become proficient with the ontology evolution system. Further, the acceptability, defined as the extent to which ontology engineers like a system, is also increased since they can easily find "weak spots" in the ontology and they are provided with the solution for these problems.

3.5 Process

As ontologies grow in size, the complexity of change management increases, thus requiring a well-structured ontology evolution process. A complete ontology evolution process derived from our discussion on ontology evolution requirements (see section 3.1) is presented in Figure 20. It has a cyclic structure since validation of realised changes may (automatically) induce new changes in order to obtain the model consistency or to satisfy users' expectations. The functional requirement for the ontology consistency results in the core component that consists of four phases: *the change representation*, *the semantics of change*, *the change propagation* and *the change implementation phases*. The second requirement for user guidance results in *the change validation phase* and the third requirement for continual ontology refinement results in *the change capturing phase*.

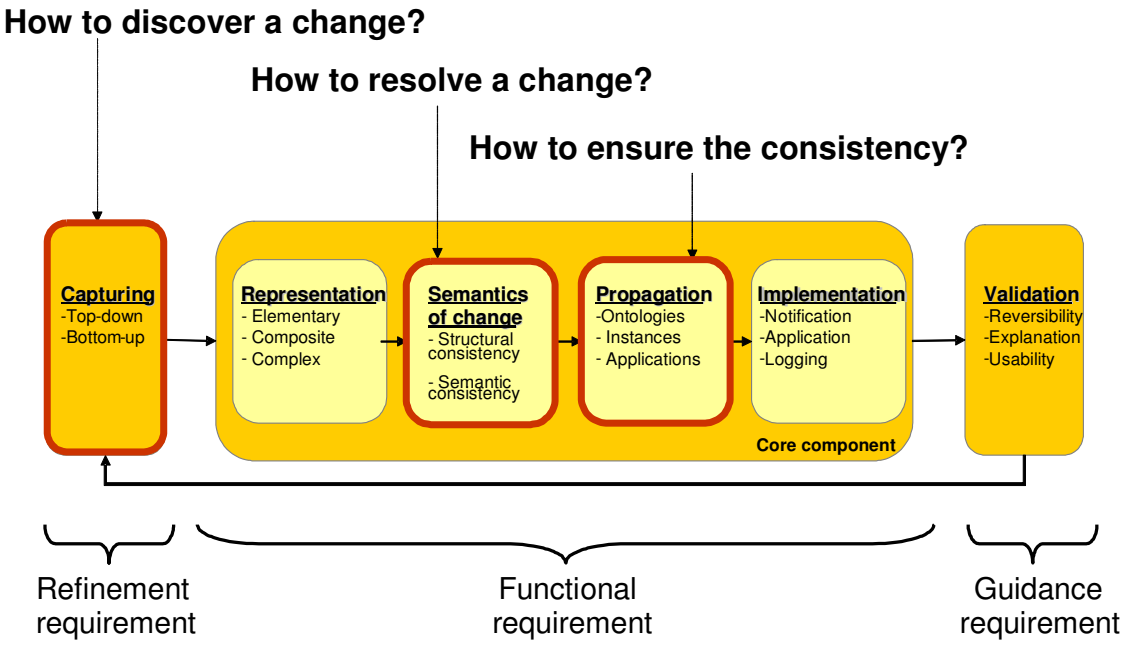


Figure 20. Ontology evolution process

In the rest of this thesis we focus only on the three phases:

- *the semantics of change phase* that helps to resolve a change (see chapter 4);
- *the change propagation phase* that ensures the consistency of the dependent artifacts (see chapter 5);
- *the change capturing phase* that helps to discover changes (see chapter 6).

The whole process (including the phases that are not further elaborated) is automated and the implementation details are given in chapter 7. This section also contains a short evaluation study that compares our approach and its realisation with the existing systems for the ontology development and evolution.

3.6 Related Work

In the last decade there has been very active research in the area of ontology engineering. The majority of research studies in this area are focused on construction issues. However, coping with the changes and providing maintenance facilities require a different approach. We cannot say that there exist commonly agreed methodologies and guidelines for the ontology evolution. Thus, there are a very few approaches investigating the problems of inducing changes in ontologies.

The discussion about the related work is split into three parts: (i) requirements for the ontology evolution support, (ii) ontology changes and (iii) the ontology evolution process.

3.6.1 Requirements

In [86] the author presents the guiding principles laid down in the fields of conceptual graphs, description logic, general frame systems, object-oriented modelling, knowledge acquisition, etc. for building consistent and principled ontologies. The goal is to alleviate their creation, the usage and the maintenance in the distributed environments. The author analyses the requirements for the tool environments that enforce consistency. These requirements include naming convention, support for multiple ontologies, versioning, etc. They enable us to identify problems, but not to resolve them. Therefore, the approach is not complete and provides a rather informal basis for ontology management. Our work supplements this approach because we have developed and realised the whole methodology for ontology management. It covers the whole ontology lifecycle. Moreover, it incorporates the necessary means such as keeping the consistency as well as the means that increase the usability of the ontology evolution system by guiding the ontology engineers through the evolution and making suggestions for the continual ontology improvement.

In [109] the authors define two main problems in conceptual schema evolution: the applicability problem (does the underlying system support the desired evolution and how to realise it?) and the conformity problem (are the evolved schema, data and application conformed to the required evolution?). In our approach these two problems are captured by the functional requirement for an ontology evolution system. We go step further by taking into account the usability (the guidance requirement) and learnability (the refinement requirement) issues.

In [47] the authors propose some design criteria and a set of principles that have shown to be useful for the development of ontologies. For example, they propose the minimisation of the semantic difference between sibling concepts. To improve ontology understanding, it would be advisable for definitions that are children of the same parent to be defined according to the same template. The same approach can be applied not only to the ontology development, but also to the ontology evolution as well. We follow these design criteria and extend them for example by taking into account the usage of an ontology. Further, we formalise them, which results in a system, which is able to make recommendations for continual ontology improvement.

More philosophical arguments concerning the need for ontology revision are made by Foo [39]. Based on this research, our approach for the handling of ontology evolution in dynamic environments relies heavily on the usage of meta-primitives, also represented in the form of ontologies.

[142] presents an extended ontology knowledge model that represents semantic information about concepts explicitly. This enriched semantic is not used for supporting evolution

problems, but for describing what is known by agents in a multi-agent system. However, it could complement our approach for resolving the semantic inconsistency as mentioned in section 3.2.2.

3.6.2 Changes

Regarding ontology changes, we discuss (i) the set of available changes and (ii) the representation of these changes.

Set of Changes

Taxonomy of ontology changes can be found in [63]. They deal with the creation, deletion or modification of the "building blocks" of the OWL ontologies. Since there are many differences between our ontology model and the OWL, the corresponding sets of changes are diverse. Moreover, our set of elementary changes does not encompass the "modify" changes due to the fact that they can be realised as the creation followed by the deletion.

A formal method for tracking changes in the RDF repository is proposed in [98]. The RDF statements are pieces of knowledge they operate on. The authors argue that during the ontology evolution, the RDF statements can be only deleted or added, but not changed. Our approach also does not distinguish "modify" changes. The number of changes supported in our approach is significantly higher than in [98] since our ontology language is an extension of the RDF (S) ontology language and therefore richer than the RDF (S). Further, the higher levels of abstraction of ontology changes such as composite and complex ontology changes are not considered at all in that approach.

Elementary changes are mentioned in different works. The first systematic analysis of the changes in the database systems is given in [3]. These changes represent the typical schema modification allowed in most systems today. Similar enumeration can be found in [154]. We have taken these approaches as foundation for our work and adopted them to the ontology evolution.

A new categorisation of the different modifications in object oriented databases is proposed in [108]. The authors defined three categories of changes: primitive, composite and complex changes. We have followed that approach and adapted it to the ontologies. Therefore, we have quite a different set of elementary changes. Further, they are much more composite changes due to the increased complexity of the ontology model and an interwoven ontology structure. Finally, although the authors discussed the need for the complex changes, they provided neither a catalogue of these changes nor a means to form and resolve them. On the contrary, we allow an ontology engineer to specify an arbitrary complex request for a change as a composition of elementary or composite ontology changes and propose an approach to resolving it.

In [93] the authors identify a set of common complex ontology changes that mainly affect a concept hierarchy. However, this set of changes is not defined in a clear manner. For example, the semantic of the "RemoveSubtree" change, which is a complex change, is equal to the elementary change "RemoveConcept" since in most of the ontology evolution systems the request for the concept removal causes the removal of all its subconcepts. In our approach, the differences between elementary, composite and complex changes originate from their definition. Elementary changes are determined by the ontology language itself. Composite changes take into the consideration entities in the neighbourhood. Finally, complex changes

are changes consisting of several elementary and/or composite changes that, taken together, constitute a single modification.

Change Representation

Information about change can be represented in many different ways. The approaches encompass various tables and lists that store different kinds of data and text (e.g. mixtures of structured and unstructured data). However, formal models for changes are lacking.

Based on our work on the evolution ontology, in [63] the authors define an ontology for representing ontology changes for the OWL knowledge. Although this ontology is similar to our evolution ontology, it contains much more elementary changes due to expressivity of the OWL ontology language. However, it has several disadvantages. Firstly, this ontology is not minimal since it contains "*modify*" changes, which specify that an old value is replaced by a new value. Even though the authors are aware that these changes can be formed by combing a "*delete*" and an "*add*" change, they are included in the ontology. Secondly, the properties defined for the concept "*Change*" do not allow us to derive the history of changes in the unique way. Therefore, the reversibility is not supported. Next, there is no information about cause and consequences of changes, which makes change propagation impossible. Finally, a log contains a list of specific operations with references to the concepts or properties that they operate on. In the case that a concept or a property is already deleted, this log would be inconsistent. Therefore, references to the domain entities must be captured in a syntactic way through their identifiers, not through themselves.

In [64] the authors propose a method for finding complex ontology changes. It is based on a set of rules and heuristics to generate a complex change from a set of elementary changes. On the contrary, our approach provides information about changes at the level at which they occurred. This information is stored in the evolution log. Therefore, there is no need to discover these changes. On the other hand, our approach makes recommendations for some changes that lead to further ontology improvement.

In [101] the authors developed a principled approach for management of changes. It consists of a set of changes (with their semantics) and a change-documentation model that may be appropriate for controlled terminologies in health case. The authors propose the CONCORDIA concept model to cope with the changes. The main aspects of CONCORDIA are that all concepts have a permanent unique identifier. The concepts are given a *retired* status instead of being physically deleted. Moreover, special links are maintained to track the retired parents and children of each concept. In our approach, all the changes are physically applied to ontology. However, it is always possible to revert to the previous state due to capturing changes in an evolution log. Regarding the change representation, the CONCORDIA approach requires to specify, for each performed change, the concept unique identifier and the current concept name. Although the concept unique identifier is a meaningless alphanumeric string, it identifies a concept unambiguously. Therefore, the current concept name is superfluous.

Change representation can be also considered from the management point of view as described in [91]. Before a change is implemented in a system, it is necessary to understand the change. The better a change is understood, the better it can be implemented and maintained. Many questions need to be answered. What problem is being solved? How does the change solve it? What are the side effects of making this change? What are the implications of not making this change? Who is affected? How are they affected? What is the reliability of this change? What is the best way to make the change? What is the scope of the change? Who is experienced in this area, with this change? That is, who can help if an

unforeseen problem occurs during or after the change? All changes should be well thought out beforehand. This process can be greatly facilitated by having a change control form which brings these questions/issues, as well as others into consideration. Once a change has been *thought out* it is tested on a system. We follow this work and incorporate the main points as attributes defined for the concept "*Change*" in the evolution ontology. However, we extend the change representation with the many aspects needed for undoing and propagating changes as well as for learning from changes.

The advantage of our approach is that the change characteristics are formally and explicitly represented. However, we do not take into account all of the characteristics (e.g. implications of not making a change, reliability of changes, who has recently applied this change) since it would require an additional effort for ontology engineers to specify them without possibility to exploit them automatically.

3.6.3 Process

The methodology for the ontology development is given in [137]. Since the ontology development is necessarily an iterative and a dynamic process, the ontology evolution is unavoidable. Therefore, the ontology development methodology includes the ontology evolution as well. In this thesis we focus only upon this phase by discussing the requirements to realise it and by introducing a process capable to cope with the changes in a more systematic way.

In [109] the authors define three major steps (request specification, identification of changes and implementation) describing the evolution process whose actor is the schema manager. This process does not fulfil all the requirements for an evolution system. The change capturing phase and the change validation phase are not covered at all. Regarding the core evolution process dealing with the consistency of a schema and its dependent artefacts, it does not treat the semantics of change problem and requires writing the transformations to update data if they do not have to be lost. Therefore, the proposed process is only a subset of our ontology evolution process. However, regarding the implementation of changes it allows to realise the evolution by view, by version or by the immediate update whereas our evolution process considers only the application.

In [106] the authors describe the activities that compose an ontology integration process and present a methodology that provides support and guidance to perform those activities. Ontology integration assumes that an ontology is not built from scratch, rather the integration is the process of building an ontology in one subject reusing one or more ontologies in different subjects ontology. Therefore, it is far more complex than the development of a single ontology. However, since reused ontologies are subject to continual change, the integration methodology has to take into account the consequences of the changed ontologies for the dependent ontology. This activity is completely covered by our ontology evolution process due to the fact that the ontologies are distributed over the Semantic Web.

A component-based framework for ontology evolution is given in [64]. It is based on the different representations of ontology changes. The approach proposes a framework that integrates these representations. It covers the following tasks: (i) data transformation; (ii) ontology update; (iii) consistent reasoning; (iv) verification and approval; and (v) data access. Our system supports the first four tasks but not the last one since it is related to the ontology versioning that was not our goal. In contrast, we provide a lot of means for the user-driven evolution, allowing the ontology engineers to specify strategies for updating when changes in an ontology occur. The users' needs are not taken into account in [64].

Methontology [35] is a methodology for building ontologies either from scratch, reusing other ontologies as they are, or by a process of re-engineering them. The framework consists of: identification of the ontology development process where the main activities are identified (evaluation, configuration, management, conceptualisation, integration implementation, etc.); a lifecycle based on evolving prototypes; and the methodology itself, which specifies the steps to be taken to perform each activity, the techniques used, the products to be output and how they are to be evaluated. Even though Methontology already mentions evolving prototypes, none of these (and similar others) methodologies responds to the requirements for distributed, loosely controlled and dynamic ontology engineering. On the contrary, the evolution of dependent and distributed ontologies in a systematic way is one of main advantages of our methodology.

In the business community, the change management assumes the process, tools and techniques to effectively manage people and the associated human resource issues that surface when implementing business changes [91]. Based on Prosci's[30] research of the most effective and common applied change, most change management processes contain the following three phases: (1) phase 1 – preparation, assessment and strategy development; (ii) phase 2 – detailed planning and change management implementation; (iii) phase 3 – data gathering, corrective action and recognition. This process is very similar to our ontology evolution process. The preparation for change phase corresponds to our change representation phase; the managing change phase should cover the semantics of change, the change propagation and the change implementation; and the reinforcing change is a simplified version of the change validation and capturing phases. However, whereas this process is focused on the organisational aspect, our approach is "engineering" centric and problem solving in nature.

3.7 Conclusion

In this section we presented a novel approach for dealing with ontology changes. The approach is based on a six-phase evolution process, which systematically analyses the causes and the consequences of the changes and ensures the consistency of the ontology and depending artefacts after resolving these changes. In order to enable the user to obtain the ontology most suitable to her needs, we specifically focus on the possibilities to customise the ontology evolution process. We identify several means to do that:

- by allowing an ontology engineer to represent an arbitrary request for a change in the easiest manner;
- by enabling an ontology engineer to control the way of resolving a request;
- by suggesting an ontology engineer to generate a change, implied by the analysis of the structure of the ontology, ontology instances or users' behaviours in the underlying ontology-based applications.

Ontology changes are dedicated to the ontology evolution regardless of the way the evolution is done. Regarding the set of supported changes, we define three levels of abstractions. In addition to elementary ontology changes that are an integral part of each ontology evolution system, we propose composite ontology changes as more complex modifications that are very relevant, realistic and important for the ontology evolution. Moreover, we explain the importance of the composite changes even though they can not be controlled a priori. Finally,

[30] http://www.change-management.com

we introduce the evolution ontology and the evolution log enabling the formal, explicit representation of ontology changes.

4 Semantics of Change

The semantics of change is one phase in the ontology evolution process that enables the resolution of ontology changes in a systematic manner by ensuring the consistency of the whole ontology. This chapter gives an overview of the problems occurring in this phase and provides two ways for resolving them.

4.1 Problem Definition

Changes are force that drives the evolution. They can be applied to an ontology in a consistent state, and after all the changes are performed, the ontology and dependent artefacts must pass into another consistent state.

However, the application of a single ontology change might not always leave an ontology in a consistent state. It means that the checking ontology consistency prior the execution of a change (see section 2.4.2) does not resolve all problems. Even though the preconditions of an ontology change are satisfied, after applying a change, an ontology may be in an inconsistent state. For example, the precondition for a concept removal is that a concept is defined in an ontology. However, deleting a concept will cause subconcepts, some properties and instances to be inconsistent since the concept is referenced somewhere whereas it does not exist anymore. Returning to the example shown in Figure 3, the removal of the concept "*Person*" causes the inconsistencies since the invariants related to the concept hierarchy (e.g. *IC_4: Concept-Closure Invariant*) and the user's defined constraint (e.g. *UC_1: Domain/Range Property User-defined Constraint*) are violated. Indeed, the parent concept for the concepts "*Academic Staff*" and "*Student*" and the child for the concept "*Root*" are not defined anymore. Moreover, the domain concept for the properties "*hasFirstName*" and "*hasLastName*" as well as the range concept for the property "*includes*" do not exist to any further extent. These irregularities are depicted in Figure 21.

Note that if an ontology engineer makes only the required change, i.e. the *RemoveConcept("Person")* change, the ontology will be left in an inconsistent state that will render it unusable. The searching for the ontology entities may result not just in the low precision – missing some relevant answers, but more important in the incorrectness – the delivery of wrong answers. That is the case with the query "Give me all concepts whose parent concept is a domain of the "*includes*" property" for the situation presented in Figure 21.

The role of an ontology evolution system is much more than finding these problems (i.e. inconsistencies) in an ontology and alerting an ontology engineer about them. This is pretty much the kind of support provided by conventional compilers. However, helping ontology engineers notice the inconsistencies only partially addresses the issue. Ideally, an ontology

evolution system should be able to support ontology engineers in resolving the problems at least by making suggestions how to do that.

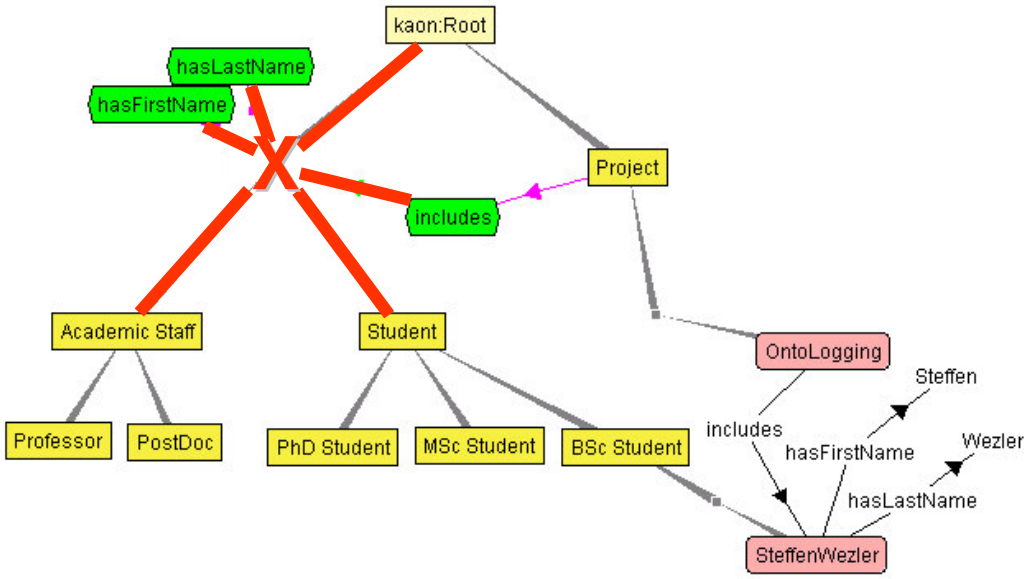


Figure 21. Inconsistent ontology since undefined entity "*Person*" is used

Therefore, when updating an ontology, it is not enough just to consider the entities figuring in the request for a change, because the other entities in the same ontology may also be affected by the updates. Since it is not sufficient to change only a part of the ontology that is related to the request for a change while keeping all the other entities intact, we introduce *the semantics of change phase* in the ontology evolution process. The task of this phase is to enable the resolution of changes in a systematic manner by ensuring the consistency of the whole ontology. This can be done by completing required changes with additional changes, which guarantee the transfer of the initial consistent ontology into another consistent state. Indeed, the updated ontology is not defined directly by applying a requested change. Instead, it is indirectly characterised as an ontology that satisfies a user's requirement for a change and it is at the same time a consistent ontology.

The role of the semantics of change phase is presented in Figure 22. It captures the knowledge about ontology changes and supports ontology engineers in completing the modification they already started. This modification may require several additional changes (cf. Ch^1,…, Ch^{n-1} in Figure 22) to various entities of the ontology, which need to be carefully co-ordinated to prevent ontology engineers from leaving the ontology in an inconsistent state.

In order to formalise the semantics of change phase we introduce the following definition:

- the function $consistency(O, M) = \begin{cases} true, & \text{if an ontology } O \text{ satisfies the set of ontology consistency constraints } M \text{ (see the definition of the ontology consistency model given in section 2.3)} \\ false, & \text{otherwise} \end{cases}$

Definition 20 Given an ontology O and a request for a change Ch, *the semantics of change phase* of an ontology evolution process is defined as:

$$SemanticsOfChange(O,Ch)= (Ch^1,..., Ch^i, Ch^{i+1},...,Ch^{n-1})$$

where:

- O is a given consistent ontology, i.e. $consistency(O, M)=true$;
- Ch is a requested change that can be applied to the ontology O, i.e. $preconditions^{31}(O,Ch)=true$;
- $O^1=Ch(O)$ is an ontology representing the result of the application of the required change Ch to the ontology O, i.e. $postconditions(O^1,Ch)=true$;
- Ch^i ,$1 \leq i \leq n-1$, is a derived change that satisfies the following set of conditions:
  - $O^{i+1} = Ch^i \circ O^i = Ch^i(O^i)$, which implies that $preconditions(O^i, Ch^i)=true$ and $postconditions(O^{i+1}, Ch^i)=true$;
  - $consistency(O^i, M)=false$, $1 \leq i \leq n-1$ and $consistency(O^n, M)=true$.

Consequently, the result of applying the request for the change Ch to the ontology O is the ontology O' defined as:

$$O' = O^n = Ch^{n-1} \circ ... \circ Ch^{i+1} \circ Ch^i \circ ... \circ Ch^1 \circ Ch \circ O =$$

$$Ch^{n-1}(...Ch^{i+1}(Ch^i(...Ch^1(Ch\ (O)))))$$

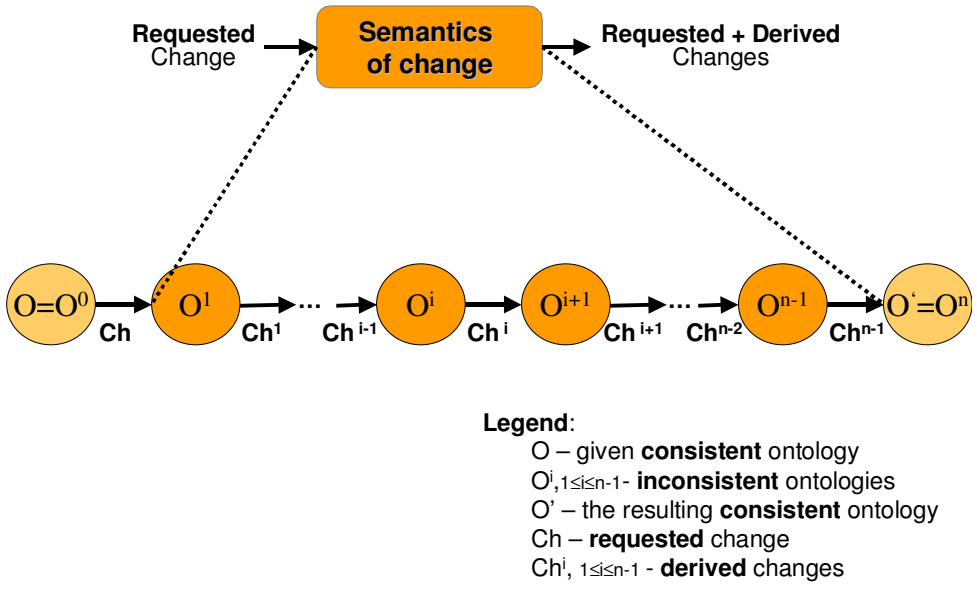


Figure 22. The role of the semantics of change

We note that the Definition 20 is an extension of Definition 14, due to the modification not just to the entities in the request for a change, but also to the entire set of ontology entities. To eliminate the drawbacks of the solution that only applies a change to the ontology O, the

[31] See section 2.4.2 for the definition of the preconditions and postconditions functions.

semantics of change is introduced to ensure the consistency of resulting ontology *O'*. It is done by extending a required change *Ch* with the additional set of changes ($Ch^i, 1 \leq i \leq n-1$) that guaranties the consistency of the ontology *O'*. It is also important to note that the set of generated changes is minimal since the removal of any change from this set would result in an inconsistent final ontology *O'*.

The challenge of the semantics of change problem is how to find the additional changes that preserve the ontology consistency. Since a new change has to be induced when the result of applying a previous change is a inconsistent ontology, it is required to bridge the gap between the postconditions, that represent the result of an ontology change, and the consistency constraints that define the ontology consistency (see section 2.3). It is up to an ontology engineer or an ontology evolution system how to perform this mapping.

Returning to the example shown in Figure 21, the semantics of change might generate a set of changes[32] depicted in Figure 23. The request for the removal of the concept "*Person*" affects other entities in the ontology: the concept "*Student*" and the concept "*Academic Staff*" since they were children of the concept "*Person*", the "*includes*" property whose range was that concept, properties "*hasFirstName*" and "*hasLastName*", whose domain was that concept, and the concept "*Root*", which was the parent of the concept "*Person*". As a result, the additional changes, denoted as 1-6 in Figure 23, are generated. Furthermore, some of these additional changes originate, in turn, the need for additional changes. For example, the change for the deletion of the concept "*Student*" causes the deletion of the concepts "*PhD Student*", "*BSc Student*" and "*MSc Student*" as well as the removal of the inheritance relationship between the concept "*Student*" and the concept "*Person*" (see a-d in Figure 23). These changes trigger new changes and so on.

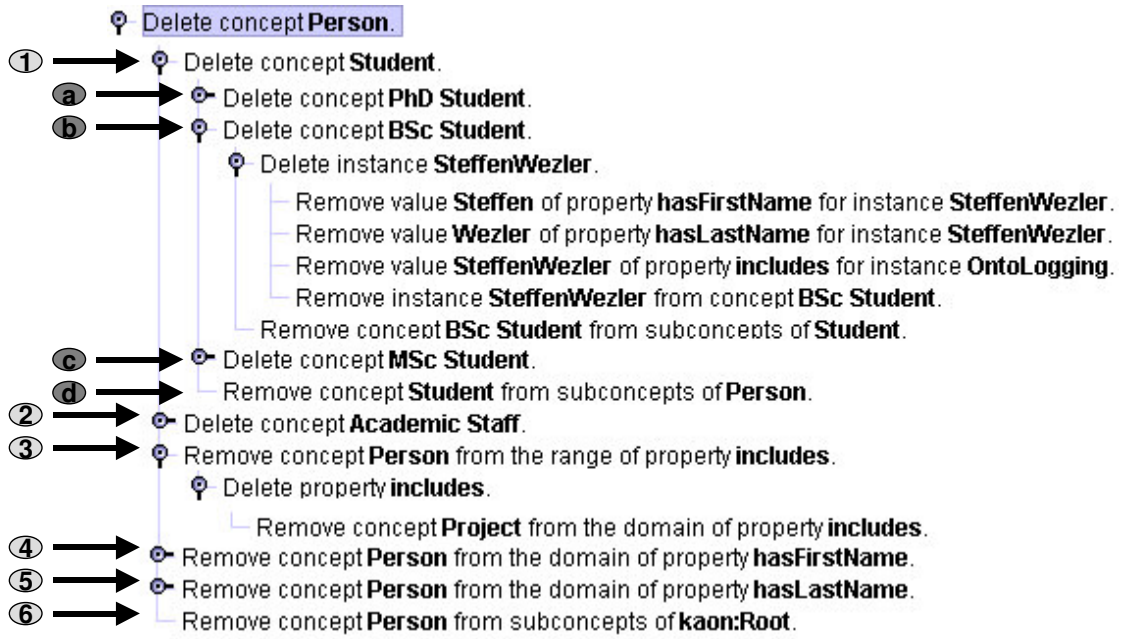


Figure 23. The result of the semantics of change for the example shown in Figure 21

The experiences from the knowledge-based systems modification [140] shows that it is hard for a knowledge engineer to track down and keep in mind all the changes that are pending.

[32] To aid the understanding of why some changes are induced, related changes are grouped and organised in a tree-like form.

Since an ontology can be much more complex that a knowledge based system (e.g. due to increasing complexity of the ontology model and physical distributed of an ontology), it is even harder for an ontology engineer to follow up on all dependencies between ontology entities. For example, it is difficult to recognise problems that are results of the inference process (e.g. a concept hierarchy) because they are not directly observable to an ontology engineer. Thus, the request for a change can almost always end up with an inconsistent ontology. Since an ontology engineer can easily overlook some part of the overall modification, it cannot be expected that the generation of additional changes needed to keep the consistency can be done manually.

In the rest of this section we discuss methods for realising this task automatically. In particular, two main approaches were adopted from the database community and followed to ensure the consistency in pursuing this semantics of change problem [37]:

1. *Procedural approach* - this approach is based on the constraints, which define the consistency of a schema, and definite rules, which must be followed to maintain constraints satisfied after each change;
2. *Declarative approach* - this approach is based on the sound and complete set of axioms (provided with an inference mechanism) that formalises the dynamics of the evolution.

We follow both approaches for the schema evolution and adapt them to the ontology evolution problem. The adaptation is not just the consequence of the differences between an ontology model and a schema model. The most important extension lies in the incorporating means for guiding the selection of the most meaningful (from the ontology engineer point of view) way for generating additional changes. In this way we introduce completely new approaches (both the procedural and the declarative) for the semantics of change that focus on providing the ontology engineers with capabilities to control and customise the ontology evolution process.

The rest of this section is devoted to both approaches for the semantics of change problem. Firstly, in section 4.2 we discuss our procedural approach for the semantics of change that builds the ways for generating additional changes into the code. Secondly, in section 4.3 we present the declarative approach that removes the *how's* (i.e. the control flow and sequencing) from the solution. Finally, in section 4.4 we give an analysis of the advantages and disadvantages of the proposed approaches.

4.2 Procedural Approach for the Semantics of Change

The precise meaning or semantics must be associated with each ontology change, so that we know how to preserve the ontology consistency after applying a change. The procedural approach for the semantics of change phase of the ontology evolution process is realised by a procedural mechanism that incorporates the semantics of ontology changes. It is capable of providing answer to a wide class of users' requests for a change, i.e. to all the requests that can be specified with one of the foreseen changes. Such an approach is given in this section. We explore the complexity of the semantics of change problem and introduce the concept of an evolution strategy encapsulating the policy for the evolution with respect to the requirements of ontology engineers [129]. We go a step further by introducing the so-called advanced evolution strategies that support ontology engineers in resolving problems by suggesting an evolution strategy that may be most appropriate in a particular situation.

4.2.1 Motivating Example

As mentioned, the essential role of *the semantics of change phase* in the ontology evolution process is to figure out which elementary changes need to be performed for one change request, e.g. deletion of a concept. If this were left to an ontology engineer, the evolution process would be too error-prone and time consuming – it is unrealistic to expect that humans will be able to comprehend entire ontology and interdependencies in it. This requirement is especially hard to fulfil if the rationale behind domain conceptualisation is ambiguous or if the ontology engineer does not have the experience.

However, there are many ways to achieve consistency after a change request. For example, when a concept from the middle of the hierarchy is being deleted, all subconcepts may either be deleted or reconnected to other concepts [11]. If subconcepts are preserved, then properties of the deleted concept may be propagated, its instances distributed, etc. Different ways for resolving the request for the removal of the concept "*Student*" by considering only the concept hierarchy are shown in Figure 24. The first solution (cf. b in Figure 24) does not contain any subconcepts of the concept "*Student*". In the second solution (cf. c Figure 24) all subconcepts are retained and connected to the concept "*Person*", which is the parent concept of the concept "*Student*". The last solution (cf. d Figure 24) also preserves all subconcepts while reconnecting them to the root of the concept hierarchy.

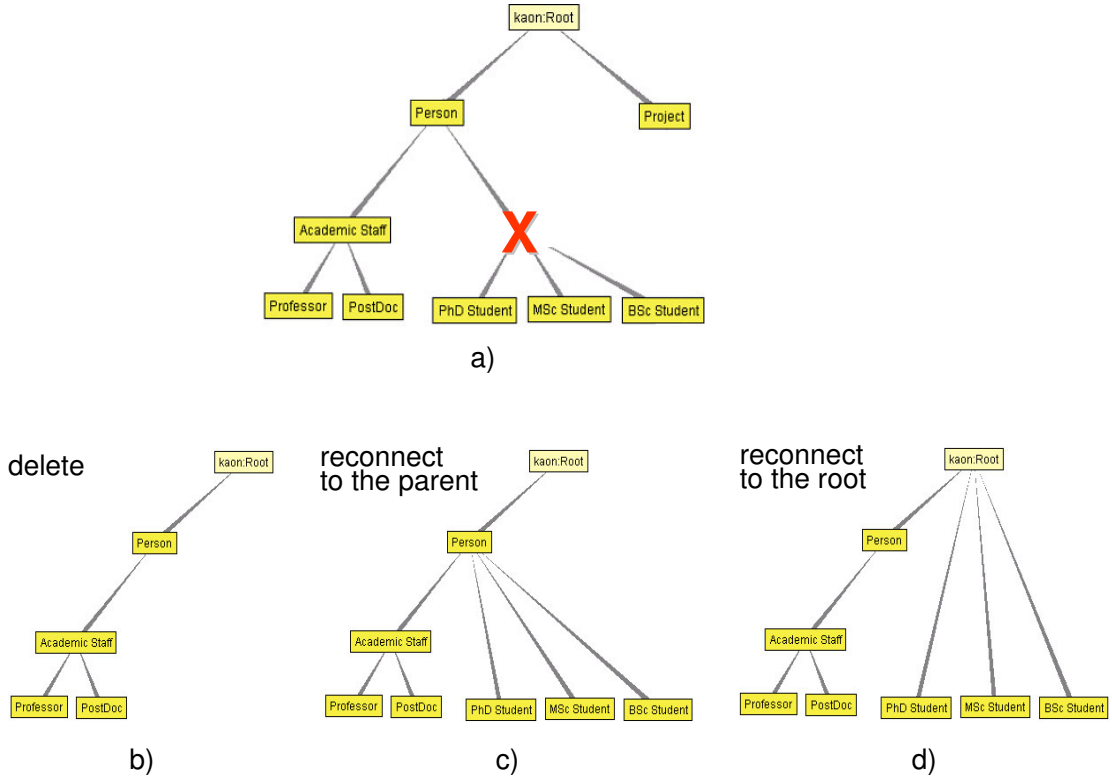


Figure 24. Different ways of resolving a change. (a) The given ontology after applying only the removal of the concept "Student"; (b) The updated ontology where all subconcepts are deleted; (c) The updated ontology where all subconcepts are reconnected to the parent concept; (d) The updated ontology where all subconcepts are reconnected to the root concept.

Each of these solutions is appropriate for some situation but not for all of them. For example, the first solution is useful for keeping an ontology as minimal as possible since it removes everything related to an entity that occurs in a request. The second solution is more suitable for preserving as many as possible entities from the existing ontology, which will help an ontology engineer to comprehend fully the effects of a request for a change. The last one keeps only the rough information about (sub)concepts that may later inform an ontology engineer about the decisions made in the past. We note that all existing systems for the ontology development provide only one possibility for realising a change and this is usually the simplest one. For example, in these systems the deletion of a concept always causes the deletion of all its subconcepts.

The previous example shows that for some changes in an ontology, it is possible to generate different sets of additional changes, leading to different final consistent states of the ontology. Thus, the role of *the semantics of change phase* of an ontology evolution system should not be only to help ontology engineers resolve all side effects of changes in order to complete the modification that they started. Ideally, it should also accomplish a request for a change in a way that is most suitable for an ontology engineer. Since only the ontology engineer has enough knowledge to decide which solution is appropriate for a request she wants to perform, she should be able to direct this process. To do so, we introduce the notion of evolution strategies (see section 4.2.3), allowing an ontology engineer to customise the process of the generation of additional changes according to her needs. In that way an ontology engineer has the possibility to transfer an ontology in the desired consistent state.

4.2.2 Conceptual Description of the Procedural Approach

Since our approach for the ontology evolution enables ontology engineers to control the semantics of changes phase of the ontology evolution process in order to tailor an ontology to suit their needs, it requires a complex model behind it. The conceptual architecture of the procedural approach is illustrated in Figure 25. An ontology engineer makes a request for a change. This request is formalised in the change representation phase of the ontology evolution process (see section 3.2.1) and it consists of one or more changes that are supported by the ontology evolution system. Therefore, a request (cf. 1 in Figure 25) is represented as a sequence of ontology changes.

This sequence is passed to the semantics of change phase shown in Figure 25. Its role is:

i. to prevent illegal changes i.e. changes that would cause inconsistencies;

ii. to ensure the preservation of the consistency in the case that the request can be applied.

This is done by processing one by one change from the given sequence until there are no changes that can be employed.

Whereas the prohibition of illegal changes is settled by checking the preconditions[33] of an ontology change (cf. 2 in Figure 25), the change generation module (cf. 3 in Figure 25) is responsible for keeping consistency. Since there are many ways to maintain the consistency, an evolution strategy (cf. 5 in Figure 25) is imposed to be in charge of directing how to do that. This process is repeated until there is no change that should be handled. Finally, all the changes that are dealt with are applied to the ontology (cf. 4 in Figure 25). This is done by adopting the postconditions of each change (see Definition 12) since they specify the direct

[33] Preconditions are applicability conditions, that is to say, the conditions under which ontology changes are semantically correct (see Definition 11).

impact of the change.

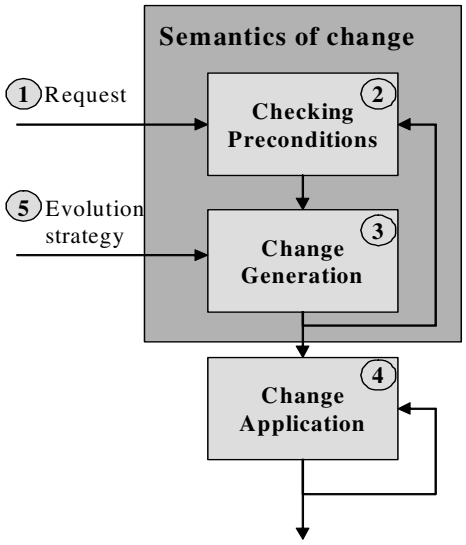


Figure 25. The conceptual architecture of the procedural approach

For example, the user's request for the removal of the instanceOf relationship between the instance "*SteffenWezler*" and the concept "*BSc Student*" from the ontology shown in Figure 4, would be processed in the following way: The input into the semantics of change phase, i.e. the request, would be the *RemoveInstanceOf("SteffenWezler", "BSc Student")* change. This instanceOf relationship is still an entity of the given ontology, and therefore the preconditions of the removal are fulfilled (see section 2.4.2). On the other hand, there is no other concept that is specified for the instance "*SteffenWezler*". Therefore, the instance "*SteffenWezler*" would be an orphaned instance. Since it does not comply with the IC_{10} invariant (see section 2.3), the change generation has to induce new changes. The resolution of the orphaned instance "*SteffenWezler*" can be performed in two ways: by generating either the *RemoveInstance("SteffenWezler")* change or the *AddInstanceOf("SteffenWezler", "Student*") change. Therefore, an evolution strategy is necessary for controlling how to abolish the arisen inconsistency. As a result the initial list of changes that has contained only the required change *RemoveInstanceOf("SteffenWezler", "BSc Student*") would be extended with one of the proposed changes. This extension depends on the needs of the ontology engineer who modifies the ontology. Anyhow, the additional change has to be handled in the same manner. Thus, the process iterates for the generated changes until all the changes in the list of changes are processed. If there are no pending changes and the ontology is still inconsistent, the whole evolution process is aborted. Otherwise, the process terminates by successively applying all the changes from the list of changes (i.e. required and induced changes) to the ontology. This is done in the change application module.

The main challenge of this approach is how to incorporate preferences of an ontology engineer during the generation of the additional changes. Our approach amounts to facing classical problems of schema change [10]: specify the semantics of ontology changes (i.e. preconditions and postconditions) and accordingly, specify rules to preserve the consistency of the resulting ontology. We extend this approach by specifying the multiple set of rules (i.e. the evolution strategies) for ensuring the consistency. Further, we define the so-called meta-rules (i.e. the advanced evolution strategies) for controlling the set of consistency enforcing rules that have to be used. This is elaborated in the next subsections. We start with the introduction of rules (section 4.2.3) that specify different side effects of a change on other

related entities. Then in section 4.2.4 we explain the meta-rules that prioritise and arbitrate among different possible solutions.

4.2.3 Evolution Strategies

As mentioned in the motivating example (see section 4.2.1), one ontology change can be resolved in several ways. It means that the different set of additional changes may be generated. Each of these sets of changes leads to the different final consistent ontology. It would be too restrictive to force ontology engineers to resolve a change in only one way. Therefore, a mechanism is required for ontology engineers to manage changes resulting not in an arbitrary consistent state, but in a consistent state fulfilling some constraints (e.g. minimal number of changes).

This can be done by using evolution strategies. Evolution strategies are developed as a method to "find" the consistent ontology that meets the needs of an ontology engineer. Evolution strategies try to capture the diversity of evolution policies. All the existing ontology evolution systems apply their own, single evolution policy. This straightforward approach is inconvenient, as it does not offer ontology engineers any way of adapting an ontology to suit individual applications. Therefore, we go a step further by providing a flexible choice of entire strategy sets as well as of low-level decision concerning a single ontology change.

In order to generate additional changes, the ontology change definition has to be enriched in the following way:

Definition 21 *An ontology change* $Ch_i \in CH$[34], $1 \leq i \leq |CH|$, is a 5-tuple:

$$Ch_i := (name, args, preconditions, postconditions, rules),$$

where the meaning of the first four parameters is given in Definition 13, and rules specify the side effects of a change on the other related entities.

To define the rules for each change, we started by finding out the cause-effect relationship between the ontology changes. This kind of dependency between the ontology changes forms the so-called *change dependency graph*.

Definition 22 *A change dependency graph* is a directed graph defined as:

$$CDG := (CH, E)$$

where:

- $CH = \{Ch_i\}$, $1 \leq i \leq |CH|$, is a set of nodes and each node represents an ontology change Ch_i;
- $E = \{E_k\}$, $1 \leq k \leq |E|$, is a set of labelled edges and each edge represents the cause-effect dependency between ontology changes (i.e. nodes). An edge is defined in the following way:

$$E_k = (Ch_i, Condition_j, Ch_l), \qquad Ch_i, Ch_l \in CH,\ 1 \leq i,l \leq |CH|,\ i \neq l.$$

$Condition_j$ is a prerequisite for the edge existence. It states **when** a change Ch_i may cause a change Ch_l. It is represented as a logical formula that contains only ontology entities.

[34] *CH* is a set of elementary ontology changes.

Therefore, $E_k = (Ch_i, Condition_j, Ch_l)$ can be read:

$$IF\ Ch_i\ THEN\ Ch_l$$

$$WHEN\ Condition_j$$

For example, one has to interpret the edge:

(RemoveConcept(x), (((x,b)$\in H_C \wedge$ x$\equiv$a) $\vee$ ((a,x)$\in H_C \wedge$ x$\equiv$b)), RemoveSubConcept(a,b))

as: the change *RemoveConcept* will trigger the change *RemoveSubConcept*, i.e. the rule:

"IF RemoveConcept(x) THEN RemoveSubConcept(a,b)"

can be applied if the following condition is satisfied:

$$((x,b) \in H_C \wedge x \equiv a) \vee ((a,x) \in H_C \wedge x \equiv b).$$

The applicability of a condition depends on the content of an ontology and the user's request. For example, for the request *RemoveConcept("Student")*, the dependency between the change *RemoveConcept* and the change *RemoveSubConcept* would be taken into account, since the concept "*Student*" in the ontology shown in Figure 24 has subconcepts. However, the request for the removal of the concept "*BSc Student*" would not provoke the generation of changes related to concept specializations since this concept is a leaf concept. In this way the change dependency graph can be considered as a schema for generating additional changes.

Note that the change dependency graph is a directed graph. For the previous example, the edge in the opposite direction:

(RemoveSubConcept(a,b), ((x,b)$\in H_C \wedge$ x$\equiv$a), RemoveConcept(x))

means: the rule "IF RemoveSubConcept(a,b) THEN RemoveConcept(x)") is applicable if:

$$(x,b) \in H_C \wedge x \equiv a.$$

The approach is based on a common technique for the maintenance of the knowledge-based systems [81], which states that dependencies between knowledge have to be represented explicitly. However, while in these systems the dependency graph consists of knowledge elements (e.g. rules in the expert systems), in an ontology evolution system the nodes of this graph are ontology changes.

It is worth mentioning that the size of the graph is fixed since the number of changes is predefined. Nevertheless, the change dependency graph has a very complex, interwoven structure[35]. In order to improve the understanding, we represent this graph as a sparsely populated matrix that is shown in Table 5. We call it *the dependency matrix*. Note that it represents a simplified view of the change dependency graph since the conditions are not shown.

The rows and the columns of this matrix signify ontology changes. If an element of the matrix $Dependency[Ch_k][Ch_j]$ has the value *0*, it means that a change Ch_k that is assigned to the row k can never induce a change Ch_j denoting a column j. Otherwise, a change Ch_k could generate a change Ch_j when the conditions are fulfilled. The corresponding element of the matrix $Dependency[Ch_k][Ch_j]$ contains these conditions[36]. Note that different changes may produce the same effect (i.e. Ch_j) and this effect is followed up independently of its cause (i.e. the original change Ch_k).

[35] The richer the set of ontology changes, the more difficult it becomes to give a precise characterization of the dependency between changes.

[36] Due to the limit of the page size we use the symbol "x" as the replacement for all the conditions.

Here we give the explanation of the content of Table 5 by assuming the following set of consistency constraints: $M=IC \cup SC \cup UC$, $IC=\{IC_i\}$, $1 \leq i \leq 16$, $SC=\{SC_j\}$, $1 \leq j \leq 2$, $UC=\{UC_1\}$. The *AddConcept* change causes the *AddSubConcept* change due to *IC4* consistency constraint. The *RemoveConcept* change causes the removal of all "links" pointing to the corresponding concept or from the concept. Consequently, the changes *RemoveSubConcept*, *RemovePropertyDomain*, *RemovePropertyRange*, *RemoveInstanceOf* can be triggered. The *RemoveSubConcept* change causes either the *RemoveConcept* change or the *RemovePropertyInstance* change depending on the number of the parent concepts that are defined for the subconcept. Since a property with a domain/range concept is considered as consistent[37], the *AddProperty* change causes the *AddPropertyDomain* and *AddPropertyRange* changes. Similarly to the *RemoveConcept* change, the *RemoveProperty* change requires the removal of all its "links", i.e. it can trigger the following set of changes: *RemoveSubProperty*, *RemovePropertyDomain*, *RemovePropertyRange*, *RemovePropertySymmetric*, *RemovePropertyTransitive*, *RemovePropertyInverse*, and *RemovePropertyInstance*. The *RemovePropertyDomain* and *RemoveProperyRange* changes may trigger the *RemoveProperty* change due to the UC_1 consistency constraint and the *RemovePropertyInstance* change due to the IC_{12} consistency constraint. Moreover, the *RemovePropertyDomain* change may cause the *RemoveMinCardinality* and the *RemoveMaxCardinality* changes, since cardinality constraints must be defined for the concept-property pairs. Since three property characteristics (symmetric, transitive and inverse of) can never be specified for attributes, the *RemovePropertyRange* change may activate the *RemovePropertySymmetric*, *RemovePropertyTransitive* and *RemovePropertyInverse* changes. The *AddMaxCardinality* change may cause the *RemovePropertyInstance* change, when the number of the corresponding instances exceeds the max-cardinality value. On the other hand, the *AddMinCardinality* change may provoke the *AddPropertyInstance* change in order to create the required number of property instances. The *AddInstance* change always causes the *AddInstanceOf* change due to IC_{10} consistency constraint. The *RemoveInstance* change causes the removal of all entities related to the instance, i.e. the *RemoveInstanceOf* and the *RemovePropertyInstance* changes. Since an instance cannot exist without being attached to a concept, the *RemoveInstanceOf* change may cause the *RemoveInstance* change. Moreover, it may trigger the *RemovePropertyInstance* change, when the instance has additional concepts. In case that the addition (removal) of a property instance corrupts the SC_2 (SC_1) cardinality constant, the *AddPropertyInstance* (*RemovePropertyInstance*) change triggers the *RemoveMaxCardinality* (*RemoveMinCardinality*) change. All other changes (e.g. *AddSubConcept* change) do not initiate additional changes.

Therefore, each row of the dependency matrix represents a set of rules that are defined for a change denoting a row. Consequently, an ontology change is a 5-tuple:

$$(name, args, preconditions, postconditions, Dependency[k]),$$

where k is a row in the dependency matrix that is assigned to the considered change.

Definition 23 The *change generation*[38] is defined as:

$$ChangeGeneration: CH \rightarrow 2^{CH},$$

where each $ChangeGeneration(Ch_k)=\{Ch_{k1},\ldots,Ch_{ki},\ldots,Ch_{kn}\}$ consists of the THEN part of those rules, defined for a particular change Ch_k, that can be applied. We note that the applicability of a rule is determined by the conditions (see Definition 22).

[37] Note that the set of consistency constraints M includes the user-defined consistency constraint UC_1.
[38] See *Change Generation* module shown in Figure 25

This solution will satisfy the needs of some ontology engineers, but not all of them. For an ontology evolution system, which claims to be "intelligent", a flexible choice from the set of potential solutions would be a powerful means for adapting an ontology to the specific requirements. To the best of our knowledge, no existing ontology evolution system offers such a strategy choice. The main reasons are the complexity of the mechanisms and interdependencies which have to be taken into account already for realising one single strategy (see Table 5), not to talk about the coexistence of different strategies in the same ontology evolution system.

To allow an ontology engineer to modify an ontology according to her preferences, the following set of notions is introduced:

Definition 24 *RP* is a set of the *resolution points*, $RP=\{RP_i\}$, $i=1,..,9$ (see Table 6), where each resolution point represents one dilemma that might occur during the change resolution.

Definition 25 *Elementary evolution strategy EES* a set of possible *ways for resolving resolution point*, $EES=\{EES_j\}$, $j=1,\ldots,23$ (see Table 6).

For example, one resolution point is how to handle orphaned concepts, i.e. concepts which do not have parent concepts anymore. One evolution strategy is to reconnect the orphaned concepts to the root concept.

Definition 26 *Resolution way RW* is a set of the possible elementary evolution strategies for resolving one particular resolution point:

$$RW: RP \rightarrow 2^{EES}.$$

For example, in the case of the previously mentioned resolution point about handling orphaned subconcepts of a concept C, the possible options (i.e. elementary evolution strategies) are:

- orphaned subconcepts of C may be deleted as well;
- orphaned subconcepts of C may be connected to the parent concept(s) of C;
- orphaned subconcepts of C may be connected to the root concept of the hierarchy.

Therefore, for the resolution point RP1 three elementary evolution strategies (EES1, EES2 and EES3) are defined: deleted, reconnected to the parent concept and reconnected to the root concept. They form a resolution way. The different impact of these elementary evolution strategies on the final ontology is illustrated in Figure 24. Similarly, the elementary strategies for all other resolution points may be elicited.

The resolution points that may occur during the change resolution and the set of elementary evolution strategies associated to each resolution point are shown in Table 6. Some resolution points can be specific to a particular change; some of them can be used during the resolution of multiple changes. Indeed, they represent the knowledge about the ontology evolution, i.e. knowledge acquired during the analysis of an ontology evolution problem. By using knowledge that is specific for the ontology evolution problem, an ontology evolution system can provide a very strong support in modifying ontologies.

Table 5. The cause and effect relationship between ontology changes organised as the Dependency matrix. The value x of an element, i.e. Dependency[i][j]=x, indicates that the resolution of a change related to the row i might induce a change related to the column j.

	AddConcept	RemoveConcept	AddSubConcept	RemoveSubConcept	AddProperty	RemoveProperty	AddSubProperty	RemoveSubProperty	AddPropertyDomain	RemovePropertyDomain	AddPropertyRange	RemovePropertyRange	AddPropertySymmetric	RemovePropertySymmetr	AddPropertyTransitive	RemovePropertyTransitive	AddPropertyInverse	RemovePropertyInverse	AddMaxCardinality	RemoveMaxCardinality	AddMinCardinality	RemoveMinCardinality	AddInstance	RemoveInstance	AddInstanceOf	RemoveInstanceOf	AddPropertyInstance	RemovePropertyInstance
AddConcept	0	0	x	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
RemoveConcept	0	0	0	x	0	0	0	0	0	x	0	x	0	0	0	0	0	0	0	0	0	0	0	0	0	x	0	0
AddSubConcept	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
RemoveSubConcept	0	X	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	x
AddProperty	0	0	0	0	0	0	0	0	x	0	x	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
RemoveProperty	0	0	0	0	0	0	0	x	0	x	0	x	0	x	0	x	0	x	0	0	0	0	0	0	0	0	0	x
AddSubProperty	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
RemoveSubProperty	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
AddPropertyDomain	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
RemovePropertyDomain	0	0	0	0	0	x	0	0	0	0	0	0	0	0	0	0	0	0	0	x	0	x	0	0	0	0	0	x
AddPropertyRange	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
RemovePropertyRange	0	0	0	0	0	x	0	0	0	0	0	0	0	x	0	x	0	x	0	0	0	0	0	0	0	0	0	x
AddPropertySymmetric	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
RemovePropertySymmetric	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
AddPropertyTransitive	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	AddConcept	RemoveConcept	AddSubConcept	RemoveSubConcept	AddProperty	RemoveProperty	AddSubProperty	RemoveSubProperty	AddPropertyDomain	RemovePropertyDomain	AddPropertyRange	RemovePropertyRange	AddPropertySymmetric	RemovePropertySymmetr	AddPropertyTransitive	RemovePropertyTransitive	AddPropertyInverse	RemovePropertyInverse	AddMaxCardinality	RemoveMaxCardinality	AddMinCardinality	RemoveMinCardinality	AddInstance	RemoveInstance	AddInstanceOf	RemoveInstanceOf	AddPropertyInstance	RemovePropertyInstance
RemovePropertyTransitive	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
AddPropertyInverse	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
RemovePropertyInverse	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
AddMaxCardinality	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	x
RemoveMaxCardinality	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
AddMinCardinality	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	x	0
RemoveMinCardinality	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
AddInstance	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	x	0	0	0
RemoveInstance	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	x	0	x
AddInstanceOf	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
RemoveInstanceOf	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	x	0	0	0	x
AddPropertyInstance	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	x	0	0	0	0	0	0	0	0
RemovePropertyInstance	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	x	0	0	0	0	0	0

Table 6. Resolution points and their elementary evolution strategies

Resolution point		**Elementary Evolution Strategy**	
Number	**Description**	**Number**	**Description**
RP1	Determines how to handle orphaned concepts	EES1	Orphaned concepts are deleted
		EES2	Orphaned concepts are reconnected to their parents
		EES3	Orphaned concepts are reconnected to the root concept
RP2	Determines how to handle orphaned properties	EES4	Orphaned properties are deleted
		EES5	Orphaned properties are reconnected to parent
		EES6	Orphaned properties are left alone
RP3	Determines how to propagate properties to the concept whose parents are changed	EES7	Don't propagate any properties of superconcepts
		EES8	Propagate all properties (including the inherited properties) to the concept whose parent changes
		EES9	Propagate only properties of the parent concept to the concept whose parent changes
RP4	Determines how to handle instances whose concept is deleted	EES10	Instances are deleted
		EES11	Only unique instances[39] are deleted
		EES12	Instances are reconnected to the parents
RP5	Determines whether a domain (range) of a property can contain a concept that is at the same time a subconcept of some other domain(range) concept	EES13	Property domain (range) cannot contain a concept whose superconcept is also a domain(range) concept
		EES14	Property domain (range) can contain any concept
RP6	Determines what constitutes a valid domain of some property	EES15	Property can exist without a domain concept
		EES16	Property cannot exist without a domain concept

[39] Unique instance is an instance attached to only one concept.

RP7	Determines what constitutes a valid range of some property	EES17	Property can exist without a range concept
		EES18	Property cannot exist without a range concept
RP8	Determines the allowed shape of the concept hierarchy	EES19	There is no constraint for a concept hierarchy shape
		EES20	Remove multiple paths to a superconcept in the hierarchy
		EES21	Do not allow multiple paths to a superconcept in the hierarchy
RP9	Determines how to handle property instances whose property is deleted	EES22	Property instances should be removed
		EES23	Property instances should be defined for the parent properties

In order to clarify the meaning of the resolution points and their elementary evolution strategies, here we discuss the resolution point RP3 in more details. This resolution point is considered when the subconcept relationship between two concepts should be deleted. Such an example is shown in Figure 26. To resolve the removal of the subconcept relationship between the concept "*Child*" and the concept "*Parent*" by assuming that the concept "*Child*" has to be preserved, there is a need to decide what to do with the properties which are inherited through this subconcept relationship. This set of properties includes the properties whose domain is the concept "*Parent*" such as "*pdC1*",…, "*pdCn*", the properties whose range is the concept "*Parent*" such as "*prC1*",…, "*prCm*" as well as the properties that the concept "*Parent*" inherits from all its parent concepts (cf. the concept "*ParentOfParent*" in Figure 26).

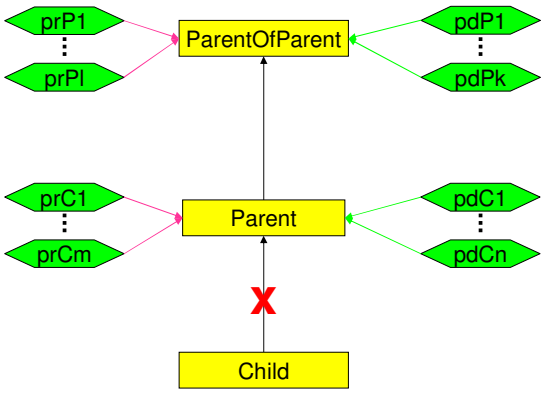


Figure 26. An example of the resolution point PR3

There are three alternatives as represented by the elementary evolution strategies *EES7*, *EES8* and *EES9*. The simplest solution achieved by the choice of the *EES7* is the avoidance of the property propagation. The elementary evolution strategy *EES8* allows the propagation of all properties (including the inherited properties) to the concept whose parent changes. By

selecting this elementary evolution strategy the concept "*Child*" will be the domain concept for the properties "*pdC1*",…, "*pdCn*" inherited from the concept "*Parent*" as well as for the properties "*pdP1*",…, "*pdPk*" inherited from the concept "*ParentOfParent*". Similarly, the concept "*Child*" will be the range concept for all properties inherited directly from the concept "*Parent*" (i.e. "*prC1*",…, "*prCm*") or indirectly from the concept "*ParentOfParent*" (i.e. "*prP1*",…, "*prPl*"). The third solution represented by the elementary evolution strategy EES9 is to propagate only properties of the direct parent concept. By adopting this elementary evolution strategy the concept "*Child*" will be attached to the properties "*pdC1*",…, "*pdCn*" as a domain concept and to the properties "*prC1*",…, "*prCm*" as a range concept.

Definition 27 *An evolution strategy ES* is defined as:

$$ES=\{(x,y) \mid x\in RP \wedge y\in RW(x)\}$$

under the following conditions:

Condition1: $\forall(x,y)\in ES \neg\exists z\in RW(x)\setminus\{y\} \ (x,z)\in ES$

Condition2: $|ES| = |RP|$

An evolution strategy unambiguously defines the way in which elementary changes will be resolved. It is a set of ordered pairs, where each pair consists of a resolution point and an elementary evolution strategy defined for that resolution point. The *condition1* requires that for each resolution point one and only one elementary evolution strategy is specified whereas the *condition2* requires that all resolution points are taken into account.

Thus, to resolve a change, the evolution process needs to determine answers at many *resolution points* – branch points during change resolution where taking a different path will produce different results. Each possible answer at each resolution point is an *elementary evolution strategy*. Common policy consisting of a set of elementary evolution strategies, each giving an answer for one resolution point, is an *evolution strategy*. It is used to customise the ontology evolution process. We call this process – the *user-driven ontology evolution process*, where the user is an ontology engineer who is able to specify an evolution strategy in order to tailor the ontology evolution to suit her needs. The relationship between all the previously introduced notions is shown in Figure 27. Resolution points and their elementary evolution strategies cover all possible ways that an ontology engineer can follow in changing an ontology, where an evolution strategy represents exactly one way to do that.

Typically a particular evolution strategy is chosen by an ontology engineer at the start of the ontology evolution process. Another option is that an ontology engineer sets up a particular evolution strategy during the request resolution, when a strategic decision has to be taken. This requires that the ontology engineer should be informed about the situation and asked for a decision. In both cases, the decision of the ontology engineer is reflected in the additional changes.

To derive a set of resolution points within an evolution strategy, we started by considering types of changes that may be applied to an ontology. Next we analysed what consequences each change can have on the ontology with respect to its definition (see section 2.3) and dependencies between ontology entities. We isolated changes that can provoke syntax inconsistencies and, consequently, could not be applied (for example, *AddSubConcept* change is not allowed if it causes an inheritance hierarchy cycles). Further, we identified that some changes could generate the need for subsequent changes, some of them offering different ways of resolution. For each particular resolution way we defined an elementary evolution strategy. For each elementary change we defined a procedure containing resolution points

encountered during change resolution. Each resolution point represents a branching point, and each elementary evolution strategy represents one possible branch. The choice of exactly one elementary evolution strategy for each possible resolution point forms an evolution strategy.

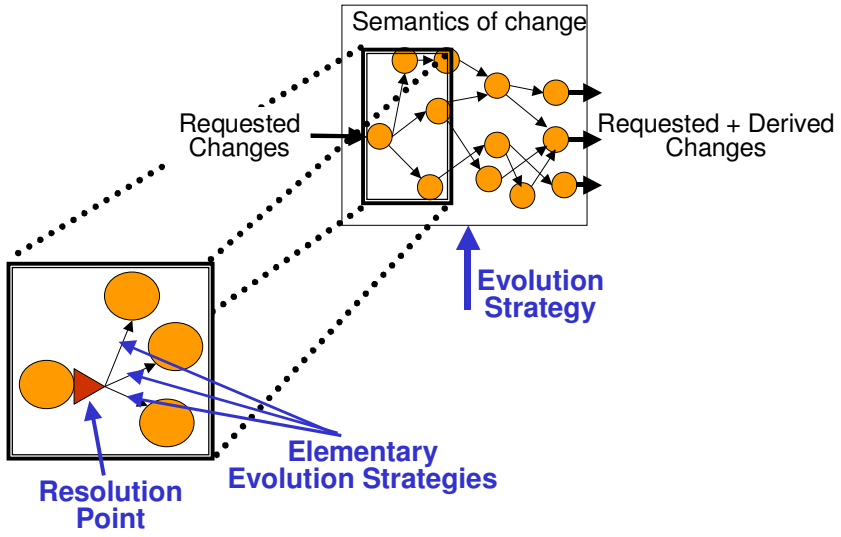


Figure 27. The role of the evolution strategy in the semantics of change

The introduction of evolution strategies entails the modification of the previously given definition of the change generation module (cf. 3 in Figure 25) that represents the key module of semantics of change phase of the ontology evolution process. This adjustment is done in the following way:

Definition 28 The change generation driven by the evolution strategy *ESChangeGeneration* is defined as:

$$ESChangeGeneration: CH \times ES \rightarrow 2^{CH},$$

where the second parameter of the *ESChangeGeneration* function is an evolution strategy that supervises the way of generating additional changes.

The previous definition is the extension of Definition 23. Only in this way does the evolution of an ontology comply with the needs of an ontology engineer.

Since evolution strategies offer more flexibility during the change resolution, they make the change generation process very complicated. Instead of two dimensional cause-effect relationships between ontology changes as illustrated in Table 5, dependencies between ontology changes form now a cube, where the newly introduced third dimension is an evolution strategy. This is shown in Figure 28. Each slice corresponds to the one way for the change generation. Moreover, there is one slice that is identical to the strategy for the change generation that is shown in Table 5.

For example, in the dependency table shown in Table 5 the *AddSubConcept* change does not cause any additional changes. However, if the evolution strategy for the resolution point RP8 that determines the allowed shape of a concept hierarchy uses the elementary evolution strategy EES20, which removes multiple paths to a superconcept in the hierarchy, then the *AddSubConcept* change might cause the *RemoveSubConcept* to change. Therefore, there is a

dependency between these two changes, which results in the creation of a new slice in the cause-effect cube. This slice represents the new version of the change dependency table.

The number of slices is equal to: $\prod_{\forall RPi \in RP} |RW(RPi)| = 3888$. This number represents the total number of possibilities for resolving ontology changes. Its high value explains why the user-driven ontology evolution is very complex. The complexity is one of the reasons why an evolution strategy is defined at the ontology level (i.e. it holds for all ontology entities) but not at the entity level (i.e. for one particular entity). If it were a case, then the number of possible combinations would increase drastically. The second reason is that an ontology engineer would have to answer to the same questions as many times as ontology entities are considered during the request resolution which can be considered as a tedious, troublesome and time-consuming effort.

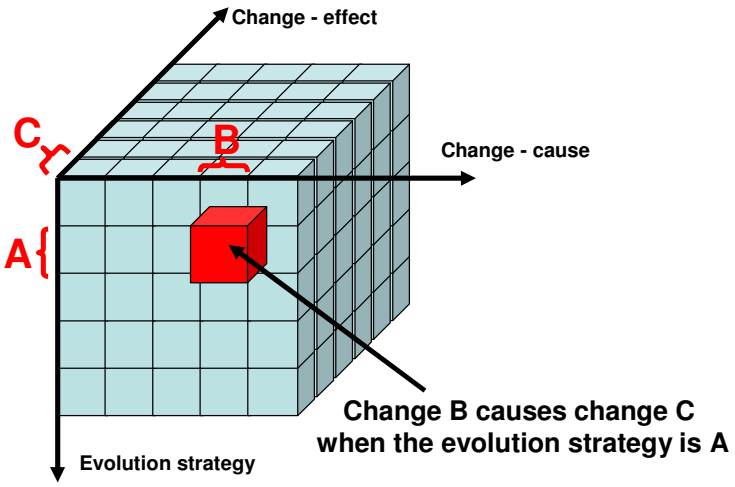


Figure 28. The cause-effect dependencies between ontology changes as a consequence of introducing evolution strategies

Evolution Strategy Example

Let us explain our approach through an example of deleting a concept C embedded in a complex concept hierarchy. The neighbourhood of a concept (see Figure 10) may be corrupted after a concept removal. According to Definition 17, the neighbours of a concept are (i) its parents or children (due to the ”subConcept” link), (ii) properties whose domain or range is a considered concept (due to the “propertyDomain”/”propertyRange” links) and (iii) its instances (due to the “instanceOf” link). An ontology engineer has to decide how to resolve these “links”. For some of them there are different resolution ways and therefore they are related to one or more resolution points. A part of the algorithm for the concept deletion with the corresponding resolution points and available elementary evolution strategies is given in Figure 29 and Figure 30.

To keep an ontology in a consistent state after a concept removal, the following resolution points should be observed:

- RP1: what to do with the subconcepts of C (cf. 6 in Figure 29);
- RP3: what to do with all properties whose domain is C (cf. 17 in Figure 29);

- RP3: what to do with the properties whose range is C (cf. 18 in Figure 29);
- RP4: what to do with instances of C (cf. 22 in Figure 29);

Note that for the *RemoveConcept* change the first resolution point always has to be taken into consideration. The usage of the other resolution points is conditional and depends on the elementary evolution strategy chosen for the resolution point RP1. For example, the resolution point RP3 defining what to do with the properties that subconcepts inherit from their parents is significant only if the subconcepts are preserved. Otherwise, there is no resolution path that takes it into consideration and therefore it is irrelevant for the particular change resolution.

Further, some resolution points are used indirectly. For example, the *RemoveConcept* change may trigger additional changes at the instance level. The resolution of these changes (cf. 58 in Figure 30) requires the consideration of the RP4 resolution point. Moreover, although one evolution strategy is defined by specifying elementary evolution strategies for all resolution points, usually only a subset of resolution points is considered. This subset is determined by the content of the evolving ontology and the request for a change.

Finally, the pseudo code of the *RemoveConcept* change shown in Figure 29 and Figure 30 illustrates that for the practical realisation of the semantics of change problem it is not sufficient to know just the dependencies between changes since they cannot be resolved in parallel. Indeed, the procedural approach controls the order in which additional changes are generated. For each ontology change all dependent changes (defined through the change dependency graph) have to be structured to a sequence of pending changes. In order to make the job of an ontology engineer easier, it is preferable to place earlier in a procedure those changes that could discard the need for other dependent changes. In this way the emergence of many unnecessary changes is avoided.

In the resolution of the concept removal shown in Figure 29 and Figure 30, dependent changes are ordered so that a problem related to the concept hierarchy is resolved primarily since the resolution of other problems depends on this decision. However, if the property propagation is resolved before the resolution of the concept hierarchy, then some concepts become temporary domain/range concepts. At the end of the resolution process these properties do not exist any more, and, consequently, there is no concept that is a domain of them. Therefore, the *AddPropertyDomain* change is generated needlessly since its effect is destroyed through the generation of its inverse change *RemovePropertyDomain*. In order not to burden an ontology engineer with temporary changes, the order of generation of additional changes has to be carefully determined. The dependencies between the generated changes and the order in which change dependency graph is traversed are shown in Figure 31.

This example shows that the dependency graph is an “unclean” and ambiguous description of dependencies between ontology changes, and, therefore, the procedural approach for the semantics of change strives to patch, interpret and clarify (in a code) many parts of the dependency graph [81].

Algorithm Semantics of Change for the Concept Removal

```
VISIT(RemoveConcept event)
Require: ℰ𝒮 = evolution strategy
 1: 𝒞 - concept that has to be deleted and is contained in event
 2: /*Checking preconditions*/
 3: ASSERTINOIMODEL(𝒞)
 4: ASSERTROOTEDNESS(𝒞)
 5: /*Change generation*/
 6: if ℰ𝒮 for the resolution point "What to do with the orphaned concepts" has the
    value ORPHANED-CONCEPTS-DELETE then
 7:    /*Remove subconcepts*/
 8:    for all subconcept of 𝒞 do
 9:       if subconcept has only one parent concept then
10:          VISIT(RemoveConcept(subconcept))
11:       else
12:          VISIT(RemoveSubConcept(𝒞, subconcept))
13:       end if
14:    end for
15: else
16:    /*Reconnect subconcepts*/
17:    RESOLVEPROPERTYDOMAIN(𝒞)
18:    RESOLVEPROPERTYRANGE(𝒞)
19:    RECONNECTCHILDREN(𝒞)
20:    DETACHSUPERCONCEPTS(𝒞)
21: end if
22: PREPAREREMOVELEAFCONCEPT(𝒞)

RESOLVEPROPERTYDOMAIN(Concept 𝒞)
Require: ℰ𝒮 = evolution strategy
23: /*Performs propagation of properties to subconcepts*/
24: if ℰ𝒮 for the resolution point "What to do with the properties" does not have the
    value PROPERTY-PROPAGATION-NO then
25:    if ℰ𝒮 for the resolution point "What to do with the properties" has the value
       "PROPERTY-PROPAGATION-ALL" then
26:       propertiesToPropagate = properties whose domain concept is 𝒞 or its parent
          concepts
27:    else
28:       propertiesToPropagate = properties whose domain is 𝒞
29:    end if
30:    for all property in propertiesToPropagate do
31:       for all subconcept of 𝒞 do
32:          VISIT(AddPropertyDomain(property, subconcept))
33:       end for
34:    end for
35:    /*Remove properties of the concept*/
36:    for all property whose domain is 𝒞 do
37:       VISIT(RemovePropertyDomain(property, 𝒞))
38:    end for
39: end if
```

Figure 29. The semantics of change for the *RemoveConcept* change (first part)

```
RESOLVEPROPERTYRANGE(Concept C)
40: /*Resolve properties whose range is a concept that is being deleted - similar to the
    property resolution when the domain does not exist anymore*/

RECONNECTCHILDREN(Concept C)
Require: ES = evolution strategy
Require: ROOT = the root concept in the ontology
41: /*Perform reconnections of children of a given concept according to evolution pa-
    rameters*/
42: for all subconcept of C do
43:    VISIT(RemoveSubConcept(C,
44:    if ES for the resolution point "What to do with the orphaned concepts" has the
       value ORPHANED-CONCEPTS-RECONNECT-TO-ROOT then
45:       VISIT(AddSubConcept(ROOT, subconcept))
46:    else
47:       for all superconcept of C do
48:          VISIT(AddSubConcept(superconcept, subconcept))
49:       end for
50:    end if
51: end for

DETACHSUPERCONCEPTS(Concept C)
52: /*Detach superconcepts of given concept*/
53: for all superconcept of C do
54:    VISIT(RemoveSubConcept(superconcept, C))
55: end for

PREPAREREMOVELEAFCONCEPT(Concept C)
56: /*Remove a concept from the ontology, provided that the concept doesn't have
    subconcepts*/
57: for all instance of C do
58:    VISIT(RemoveInstanceOf(C, instance))
59: end for
60: for all property whose domain is C do
61:    VISIT(RemovePropertyDomain(property, C))
62: end for
63: for all property whose range is C do
64:    VISIT(RemovePropertyRange(property, C))
65: end for
66: for all superconcept of C do
67:    VISIT(RemoveSubConcept(superconcept, C))
68: end for
```

Figure 30. The semantics of change for the *RemoveConcept* change (second part)

4.2.4 Advanced Evolution Strategies

In real business the choice of how a change (e.g. the deletion of a concept) should be resolved may be based on characteristics of the final state of an ontology (e.g. the depth of hierarchy should be as small as possible) or on characteristics of the process for resolving changes itself (e.g. the minimal cost of changes should be incurred). In order to enable such customisation of the ontology evolution process, an ontology engineer may choose an *advanced evolution strategy*. It represents a mechanism to prioritise and arbitrate among different evolution

strategies that are available in a particular situation, relieving the ontology engineer of choosing elementary evolution strategies individually.

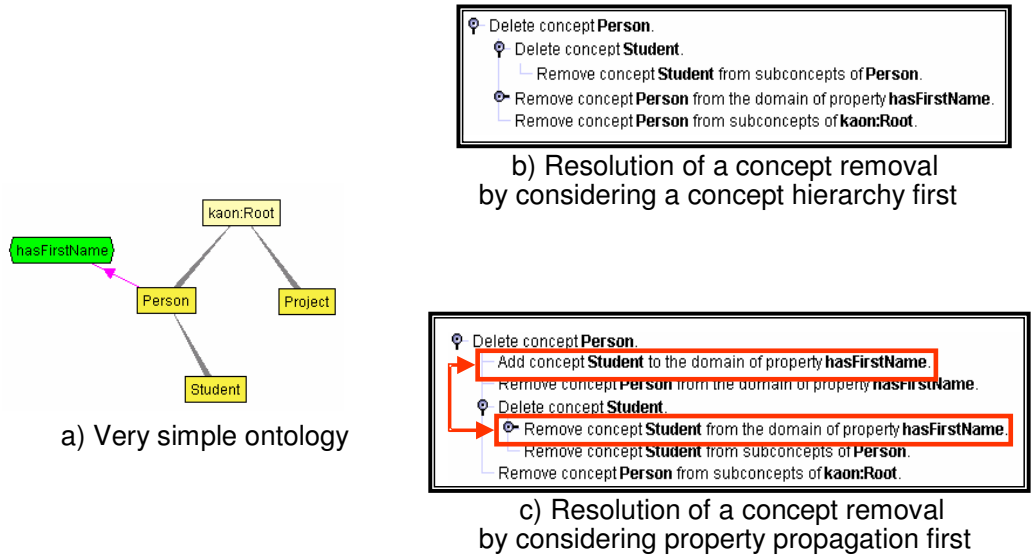


Figure 31. Impact of the order of resolving problems on the generated changes

Indeed, the advanced evolution strategies try to address the usability issue. Ontology engineers have to be able to direct the semantics of change phase of the ontology evolution process, but rather at the right level of generality in the control provided. A control such as "orphaned concepts have to be deleted" is like to be less useful to an ontology engineer than "keep the ontology as minimal as possible". This does not mean that evolution strategies need to be described without selecting elementary evolution strategies for some resolution points. In fact, evolution strategies can be more understandable when they are described in an abstract way. This leads to the advanced evolution strategies.

An advanced evolution strategy automatically combines available elementary evolution strategies to satisfy the user's criteria. We have identified the following set of advanced evolution strategies that reduces the burden involved in modifying ontologies:

- *structure-driven strategy* – resolves changes according to criteria based on the structure of the resulting ontology, e.g. the number of levels in a concept hierarchy. This strategy follows the requirements of the real-word ontology-based applications, such as MEDLINE (see section 5.4). MEDLINE requires a weekly update, usually involving only supplementary concept records. However, the concept hierarchy is updated annually. This kind of changes is performed by keeping the hierarchy minimal, because it alleviates, according to the authors of MEDLINE, the understanding of the conceptualisation;
- *process-driven strategy* – resolves changes according to the process of changes itself, for example optimised per cost[40] of the process or per a number of steps involved[41]. Determining what has to be changed and how to change it requires a deep

[40] Cost may be defined by the number of the removals.

[41] The application of the process-driven evolution with the minimal number of the changes for the ontology shown in Figure 24a) results in the solution (c).

understanding of how the ontology entities interact with one another. We cannot expect that an ontology engineer spends time explaining the reasons for all performed changes and their ordering. One strategy enabling that an ontology engineer can easily follow and understand sequences of the changes is to perform the minimal number of the updates;

- *frequency-driven strategy* – applies the most often used or last recently used evolution strategy.

The structure-driven advanced evolution strategy tries to minimise a concept hierarchy. Setting up appropriate elementary evolution strategies for the resolution points that have impact on the concept hierarchy does this. There are only two resolution points that influence the hierarchy: the resolution point RP1 that determines how to handle orphaned concepts and the resolution point RP8 that determines the allowed shape of the concept hierarchy. The minimal hierarchy is achieved by choosing the elementary evolution strategy EES1 that causes the deletion of the orphaned concepts for the resolution point RP1 and by opting for the elementary strategy EES20 that removes multiple paths to a super-concept in the hierarchy for the resolution point RP8. The choice of the elementary evolution strategies for all other resolution points is not important for achieving the goal – the minimal concept hierarchy.

The process-driven advanced evolution strategy as well as the structure-driven advanced evolution strategy attempt to reduce the number of the combinations for selecting an evolution strategy. However, they are based on quite different heuristics. The process-driven advanced evolution strategy tries to speed up the ontology evolution process by avoiding the subtractive i.e. capacity reducing ontology changes (see Definition 9) since they principally provoke the additional changes. It means that when there is a possibility to resolve an inconsistency problem by generating something in the ontology, this solution is always accepted. For example, for the process-driven advanced evolution strategy the elementary evolution strategy EES1 that causes the deletion of the orphaned concepts for the resolution point RP1 will never be selected. The reason is that there are options for resolving the same problem (i.e. the resolution point RP1) without causing the deletion (i.e. EES2 and EES3). Moreover, the resolution point RP3 deciding what to do with properties whose parent changes, will always be resolved through the choice of the EES8 since in this way the probability for the deletion of the property instances is decreased.

This advanced strategy is convenient for improving the understanding of the ontology evolution process because the number of additional changes that are needed to preserve the consistency is significantly reduced. Moreover, the final ontology is more similar to the source ontology [75] which makes easier for an ontology engineer to comprehend all effect of a required change.

The frequency-driven advanced evolution strategy tries to incorporate the needs of an ontology engineer in the process of defining an evolution strategy. This strategy is based on the assumption that the way an ontology engineer makes a change sheds some light on how she may go about making other changes. Consequently, it is preferable to set up an evolution strategy that is “similar” to the evolution strategies that guided the generation of additional changes in the past.

In fact, the frequency-driven advanced evolution strategy approximates the choice of an evolution strategy by keeping track on when an evolution strategy was last used. If an evolution strategy has recently been used then it is likely that it will be used again in the near future. Conversely, an evolution strategy that has not been used for some time is unlikely to be used again in the future. There are two variants of the frequency-driven evolution strategy:

- *most often used frequency-driven advanced evolution* strategy - it requires the analyses of the history of the used evolution strategies in order to rank them according to their usage. Indeed, an evolution strategy is created by selecting the most used elementary evolution strategies for each resolution point;
- *last recently used frequency-driven advanced evolution strategy* – it sets up an evolution strategy that is identical to the evolution strategy used during the previous evolution of the same ontology.

The frequency-driven advanced evolution strategy requires the tracking of the used evolution strategies. The new evolution strategies are inferred by an analysis of the history of the evolution strategies taken in the past. Past histories are low-level syntactic knowledge that captures the context of the ontology modification [82].

4.2.5 Complexity

A standardised measure of the complexity of an algorithm is the number of operations it takes to solve a problem in the worst case. Considering the complexity of the procedural approach for the semantics of change, the number of the operation determining the complexity is the number of additional changes that are generated during the resolution of a given change. It depends on a given change. The worst case (from the complexity point of view) is a concept removal, which, in its turn, causes the removal of all its subconcepts due to the number of inconsistencies that may arise. We do not consider the *RemoveOI-model* change since it causes inconsistency between ontologies, which is elaborated in more details in chapter 5.

A part of the change dependency graph[42] related to the *RemoveConcept* change is shown in Figure 32. There is only one change (the *RemoveSubConcept* change) that can trigger the concept removal. Moreover, there are four consequences of this change, namely the *RemoveSubConcept* change, the *RemovePropertyDomain* change, the *RemovePropertyRange* change and the *RemoveInstanceOf* change. These consequences form four paths in the change dependency graph that has to be traversed in order to determine the complexity of the approach.

The number of the operation that determines the complexity depends on the "size" of the problem data. Removal of a concept that has 1 million instances is quite different from the removal of a concept with 10 instances. Therefore, we express the number of the additional changes (expected in the worst-case scenario, i.e. during the concept removal) as a function of some characteristic numbers that suffice to capture the volume of the work to be done.

To find out these numbers we analysed all paths[43] in the graph from the node that corresponds to the concept removal to the nodes that do not have any output edges, which implies that the changes assigned to these nodes do not provoke any inconsistency. As shown in Figure 32, there are several leaf nodes: *RemoveSubProperty*, *RemoveMinCardinality*, *RemoveMaxCardinality*, *RemovePropertySymmetric*, *RemovePropertyTransitive*, *RemovePropertyInverse*. We assign the numbers (measures) to each edge on each path between the "concept removal" node and leaf nodes representing changes that do not cause additional changes.

[42] Note that this graph is a simplified view of the all dependencies since it does not include nodes representing all changes (e.g. *AddConcept* change). Nodes stand for ontology changes and the edges correspond to the cause-effect relationships between changes. We do not show the label of edges representing conditions for the change invocation.

[43] A path is a sequence of nodes traversed by following the edges between them.

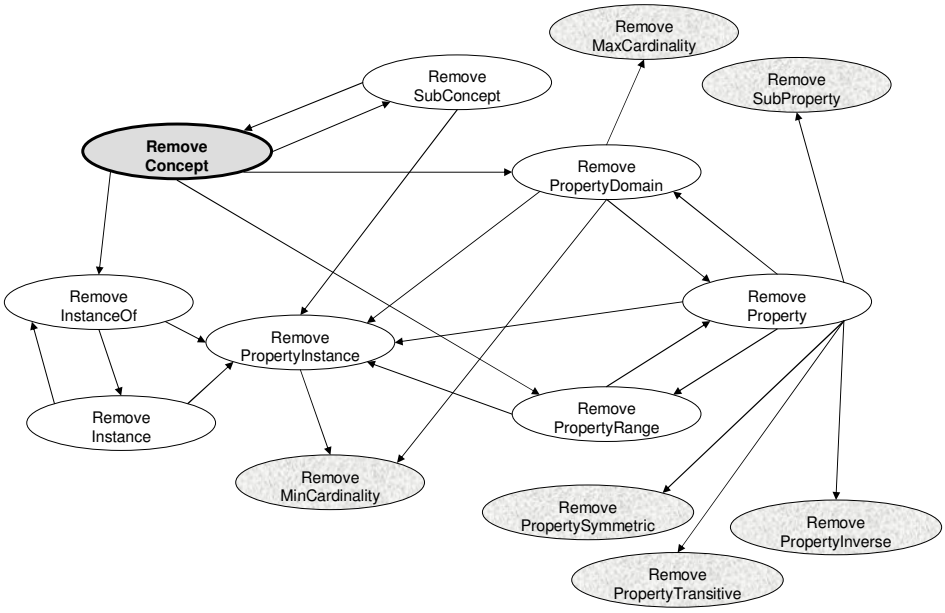


Figure 32. A part of the dependency graph that shows causes and consequences of a concept removal

There are paths that include the same node two or more times. All these cycles require a special attention, because of the complexity they introduce. The most complicated path is through the node related to the removal of the inheritance relationship. The *RemoveConcept* change causes the *RemoveSubConcept* change, which further causes the *RemoveConcept* change. However, the next call of the *RemoveConcept* change is not applied to the same concept but, instead, to its subconcepts. This cycle in a graph is modelled through the "shape" of a concept hierarchy. The number of the layers and the number of the concepts at one layer determine the shape.

Other cycles exist between the *RemoveProperty* change and the changes *RemovePropertyDomain* and *RemovePropertyRange*, respectively. On the contrary to the previous cycle, that provokes a recursion, these cycles have the completely different interpretation. It means that either the *RemoveProperty* change triggers the *RemovePropertyDomain*/*RemovePropertyRange* change or vice versa. However, once activated, the *RemovePropertyDomain*/*RemovePropertyRange* change, as a consequence of the *RemoveProperty* change, will not cause the *RemoveProperty* change again since the property does not exists anymore. Similarly, if the *RemoveProperty* change is a consequence of the *RemovePropertyDomain*/*RemovePropertyRange* change, it does not generate its inducement. The similar strategy can be applied to the cycle between the *RemoveInstanceOf* change and the *RemoveInstance* change.

Therefore, to estimate the complexity of the *RemoveConcept* change the following numbers are used:

- *m* - average depth of a concept hierarchy starting from the considering concept;
- *n* - average number of subconcepts;

- *d* – average number of single-domain[44] properties per concept;
- *r* – average number of single-range[45] properties per concept;
- i - average number of instances per concept[46];
- *pii* – average number of property instances per instance;
- *ppi* – average number of property instances per property;
- *sp* – average number of subproperty per property.

A property can be defined as being transitive/symmetric/inverse or not. Therefore, all outgoing edges from the node that corresponds to the *RemoveProperty* change are associated with the value 1. Similarly, for a pair property-domain either one or none minimum/maximum cardinality constraints can be specified. Therefore, the value 1 is assigned to all input edges to the nodes representing the *RemoveMinCardinality* and *RemoveMaxCardinality* changes.

We note that some edges are not explicitly taken into account since they are covered through the longer path that represents the worse case. For example, the fact that the *RemovePropertyDomain* can cause the *RemovePropertyInstance* change is comprised through the node that represents the *RemoveProperty* change.

By taking into account these numbers, it can be calculated that the *RemoveConcept* change resolves the semantics of change problem in at most:

$$(1 + n^2 + n^3 + ... + n^{m-1}) * (i * pii + (d + r) * (ppi + sp + 3) + d)$$

operations. It means that the *RemoveConcept* change generates approximately this number of additional changes. We are primarily interested in the rate of the growth of this function as the defined numbers increase. Since we want to distinguish between mild and "exploding" growth rates, the lower order numbers are discarded, i.e. $(1 + n^2 + n^3 + ... + n^{m-1}) \approx n^m$. Moreover, the number of singe domain/range properties related to one concept is low in general, so that so the *(d+r)* factor is not important. The same holds for the subproperties defined for a property. Consequently, their impact (i.e. the factor *sp*) can be neglected. Further, the number of property instances per property *ppi* is subsumed by the number of property instances per instance *pii*. Therefore, it does not determine the rate of growth. Finally, the number of property instances for an instance *pii* is usually of a lower order that the number of instances.

The end result is that the complexity of the *RemoveConcept* change can be sufficiently described by the function $g(RemoveConcept) \approx n^m * i$, which means that the resolution of this change takes time $O(n^m * i)$.

Therefore, the total number of additional changes for the concept removal is $O(n^m * i)$. Ontologies usually contain maximum 10 layers (i.e. $m \leq 10$) and not more that 10 subconcepts for one concept (i.e. $n \leq 10$). On the other hand, the number of instances can be very large (e.g. 1.000.000). However, the complexity depends on the number of instances in a linear way.

The calculation does not include the choice of an evolution strategy due to several reasons. Firstly, we analysed the worst case, which is covered through one evolution strategy.

[44] A single-domain property is a property that has only one domain concept. If more than one concept is a domain of a property, then the removal of a concept does not trigger the removal of a property.

[45] The similar explanation as for the single-domain property.

[46] It covers the path of the *RemoveConcept* change to the *RemoveInstance* change (through the *RemoveInstanceOf* change) since an instance cannot exist without being attached to a concept.

Secondly, an evolution strategy is selected before the evolution process is started. Finally, and more importantly, the choice of other evolution strategies can only further restrict the number of generated changes.

4.3 Declarative Approach for the Semantics of Change

The effectiveness of an ontology-based application heavily depends on the possibility of the underlying ontology to model the given domain. Due to the changes in the application's domain or in the user's requirements an ontology evolves over time. Since an ontology is a complex, interwoven structure, an ontology change can be resolved in several ways (see section 4.2). For example, after a concept removal, its subconcepts can be preserved or deleted as well. If they are preserved, they have to be reconnected either to the parent concepts of the deleted concept or to the root concept. Further, it may happen that a user wants to retain some subconcepts and to remove the rest of them. Thus, for a more complex change the number of possible solutions increases dramatically. Each solution meets the requirements of some of ontology engineers, but not of all of them. Since the needs of an ontology engineer cannot be anticipated, it is also difficult to determine exactly which way of resolving a request should be built into a system. Thus, for an ontology evolution system to be useful, effective and efficient, it has to address two issues [125]:

1. how an ontology engineer can specify her request for a change;
2. how this request can be realised.

The first problem requires a method for expressing a need of an ontology engineer in an easier, more exact and declarative manner. It is in contrast to all existing approaches, where the ontology engineer can only select a change from a predefined set of ontology changes, which does not cover all the needs of ontology engineers.

Regarding the second problem, most of the existing approaches provide only one way of resolving a change; this is usually the simplest one. Even though evolution strategies offer more flexibility by enabling an ontology engineer to control and customise the way of resolving a change, they do not cover all possibilities. To achieve the completeness of the system, the solution is not to specify in advance the possible ways of resolving a change, but to enable the system to calculate on its own all the ways that satisfy the needs of ontology engineers.

In the rest of this section we present an approach for the ontology evolution that addresses two key issues: the specification of a change request and the resolution of a change. It considers the ontology evolution as a reconfiguration-design problem solving task [152] and adapts the graph searching method to the ontology evolution problem. By modelling the ontology evolution as reconfiguration-design problem solving task, the problem is reduced to a graph search where the nodes are evolving ontologies and the edges represent the changes that transform the source node into a target one. The search is guided by the constraints provided partially by an ontology engineer and partially by a set of rules defining the ontology consistency. In this way we allow an ontology engineer to specify an arbitrary request declaratively and ensure its resolving. One of the benefits of this approach is that it allows the ontology engineer to specify declaratively what she wants to do and not how to do that.

The rest of this subsection is organised as follows: In section 4.3.1 we give a short example explaining the requirements for an ontology evolution system. The conceptual architecture of the declarative approach for the semantics of change is given in section 4.3.2. In section 4.3.3 we model the semantics of change as reconfiguration-design problem solving task. The possibilities for the request formalisation are discussed in 4.3.4. The adaptation of a graph

search to the ontology evolution is elaborated in section 4.3.5. Finally, we discuss the complexity of the proposed approach in section 4.3.6.

4.3.1 Motivating Example

To illustrate the process of resolving a user's request for an ontology change we use as an example a very simple ontology shown in Figure 24(a). Let's assume that the information about students is not needed any more, but the information regarding PhD students and MSc students has to be retained. In most of the existing approaches this need can be realised through successive applications of a sequence of ontology changes:

- ***AddSubConcept****("PhD Student", "Person")* - defines the concept "*PhD Student*" as a subconcept of the concept "*Person*";
- ***AddSubConcept****("MSc Student", "Person")* - defines the concept "*MSc Student*" as subconcept of the concept "*Person*";
- ***RemoveConcept****(Student)* – removes the concept "*Student*". (1)

Identification of the corresponding sequence of changes that realise a request is complex, because it supposes the correctness of the sequence. Other orders of these changes may not satisfy the request of an ontology engineer, e.g. the removal of the concept "*Student*" in the first step will cause the removal of the concepts "*PhD Student*" and "*MSc Student*" as well, which does not meet the user's request. Moreover, the ontology engineer has to understand the semantic of changes. For example, she has to be able to understand that the removal of the concept "*Student*" causes the removal of the subconcept "*BSc Student*" as well.

However, we cannot expect that an ontology engineer spends time finding, grouping and ordering the ontology changes to perform the desired update. In order to do that, the ontology engineer should be aware of the way of resolving a change, she should find out the right changes, foresee and solve the intermediate conflicts that might appear and order changes in a right way [10]. This activity is time consuming and error prone. The need of the ontology engineer is especially hard to fulfil if an ontology is large (e.g. thousand concepts) or complex (e.g. multiple concept hierarchy).

Further, it is not sufficient that the sequence of changes should carry out exactly the desired modification. This sequence has to be created in a way that its resolution has acceptable performance considering time and memory. Therefore, it must not contain the changes whose side-effects will be subsumed or destroyed by effects of other changes.

A more user-oriented approach can be obtained by introducing evolution strategies that are described in the previous section. For example, one strategy for the concept removal may be to reconnect the subconcepts to the parent concept. By selecting this evolution strategy at the start of the ontology evolution process, the ontology engineer can realise the previously mentioned request using only two changes independently of their order:

- ***RemoveConcept****("Student")* – remove the concept "*Student*";
- ***RemoveConcept****("BSc Student")* – remove the concept "*BSc Student*". (2)

Although the needed sequence of changes in the KAON system (see section 7.2) might be significantly shorter than in other systems (e.g. for a more complex request and/or a large ontology), the disadvantage is that the ontology engineer has to specify an evolution strategy. However, determining the right evolution strategy that meets the need of the ontology engineer requires a deep understanding of what an evolution strategy is. Advanced evolution strategies (see section 4.2.4) are introduced to gain the better understanding.

Two crucial problems arise from the previous discussion:

1. How to specify a request for a change without having deeper knowledge about the ontology and the evolution process? The ontology engineers should be able to specify their complex goals without considering how they can be realised;
2. How to ensure that an arbitrary request for an ontology change can be satisfied in a very large space of resolving changes? The system has to be flexible enough to cope with complex, unpredictable needs of ontology engineers. However, since it is impossible to predict with certainty what future requirements will emerge (e.g. a set of changes with several conditions), it is difficult to determine exactly what flexibility (i.e. changes) to build into a system. Further, it is necessary to relieve the ontology engineers of selecting and ordering changes. Finally, it would be useful to help them choose evolution strategies.

Our goal was to develop an ontology evolution system, which fulfils these two requirements. Considering the first requirement, the request of an ontology engineer for changes from the previous example should be specified declaratively like:

RemoveConcept*("Student") and*

not ***RemoveConcept****("PhD Student") and not* ***RemoveConcept****("MSc Student")* (3)

which is identical to the given need of the ontology engineer. This is in contrast to the previous two specifications ((1) and (2)), which do not specify the needs of the ontology engineer (what has to be done), but the realisation of it (how it has to be done). Note that this specification does not contain anything related to the concept "*BSc Student*" since this was not a part of the request.

Secondly, the more complex a change is, the more difficult it becomes to predict all ways of resolution to be desired in the future. For example, none of the existing system considers that the merging of two source concepts into one single merged concept can be performed as a union, an intersection or a difference of the properties of the two input concepts [15]. Thus, the only solution is not to specify how to resolve a change, but to let the system alone to find all possibilities to perform a change[47] and to order them appropriately. It is analogous to the problem of reconfiguring a system under the given set of constraints, in which a possible solution is found using straightforward graph searching [110]. Such an approach for the ontology evolution is elaborated in the rest of this section.

4.3.2 Conceptual Description of the Declarative Approach

Since our approach for the ontology evolution enables the specification of complex requests for the changes and in the same time gives all possible ways to resolve such a request, it requires a complex model behind it. In order to make the presentation of the approach more understandable, in this section we give the conceptual description of our declarative approach for the semantics of change whereas the particular modules are presented in next sections in more details. The conceptual architecture is shown in Figure 33.

An ontology engineer expresses her request for a change in a declarative manner, as a collection of ontology changes that are supported by a system (e.g. *RemoveConcept("Student") and not RemoveConcept("PhD Student") and not RemoveConcept("MSc Student")*) (cf. 1 in Figure 33). In the request formalisation module (cf. 2) this request is split into two sets of changes. The first one contains changes that must

[47] For example, the concept "*BSc Student*" can or cannot be a part of the resulting ontology.

be performed (cf. 3) whereas the second one consists of changes that must not be performed (cf. 4). Returning to the previous example, the request is represented internally as:

- *posCH={RemoveConcept("Student")}*
- *negCH ={RemoveConcept("PhD Student"),RemoveConcept("MSc Student")}*

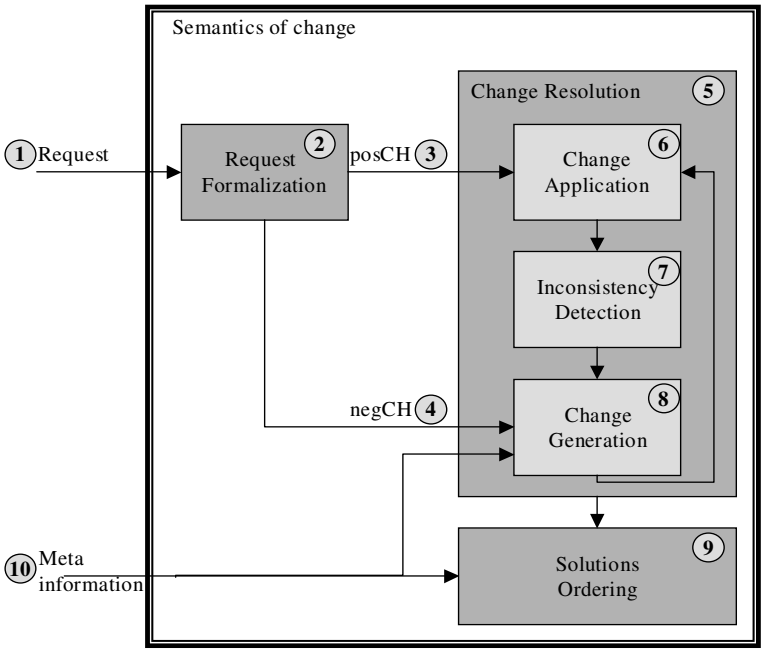


Figure 33. The conceptual architecture of the declarative approach for the semantics of change

Both the sets are inputs for the change resolution module (cf. 5). This module has to apply all changes (see section 2.4.1) from the set *posCH* and to ensure the consistency (see section 2.3) of the resulting ontology. Since the application of a single change (cf. 6) can imply inconsistencies, they have to be found (cf. 7) and resolved (cf. 8). The resolution of each inconsistency is treated as a generation of all possible changes that eliminate this problem. It is realised as searching in the change resolution space (see section 4.3.5) using the forward searching method. Since the searching space can be huge, we introduce several heuristics to constrain it (see section 4.3.5). The changes from the set *negCH* are also considered as a constraint (cf. 4). The process is repeated until ontology consistency is achieved.

As a result, the ontology engineer obtains all possible consistent successive states of the initial ontology satisfying her request. In the solutions ordering module (cf. 9) these resulting ontologies are ranked according to some meta information (cf. 10) given by the ontology engineer (e.g. the minimal number of changes which have to be done) as described in section 4.3.5.

4.3.3 Semantics of Change as Reconfiguration-design Problem Solving Task

Problem solving methods are methods that can be employed to solve a problem of a particular type (such as design, diagnosis, monitoring, etc.) [152]. They refine the generic inference engines to allow a more direct control of the reasoning process and provide abstract model of

reasoning process. Since the control knowledge is specified independently of the application domain, the reuse of this strategically knowledge is enabled for different domains and applications. In addition, problem-solving methods are abstracted from specific representation formalisms, in contrast to general inference engines that rely on a specific representation of the knowledge. As such, problem solving methods are special types of software architectures: software architectures for describing the reasoning part of knowledge-based systems [14], [52], [111].

Configuration can be informally defined as a special case of design activity, with the key feature that the artefact being designed is assembled from a fixed set of predefined components that can only be connected in predefined ways. Selecting and arranging combinations of parts, which satisfy given specifications, is the core of a configuration task. The specification of configuration tasks, in general, involves two distinct phases, the description of the domain knowledge and the specification of the desired product. The domain knowledge describes the objects of the application and the relations among them. The specifications for an actual product describe the requirements that must be satisfied by the product and, possibly, optimising criteria that should be used to guide the search for a solution. The solution has to produce a list of selected components and, as important, the structure and topology of the product.

More formally, a configuration task can be defined as [143]:

Definition 29 Given: (A) a fixed, pre-defined set of components, where each component is described by a set of properties, ports connecting it to other components, constraints at each port that describe the components that can be connected at that port, and other structural constraints; (B) some description of the desired configuration; (C) some possible criteria for making optimal selections, a configuration task has (1) to build one or more configuration that satisfy all the requirements, where a configuration is a set of components and a description of the connection between the components in the set, or (2) to detect inconsistency in the requirements.

Therefore, the configuration design is the task of searching for an assembly of predefined components as a solution, which satisfies a set of requirements and obeys a set of constraints. The reconfiguration-design [135] can, but need not start from a complete requirement specification. It is quite possible that the only information that exists apart from the system that is to be reconfigured is the information about the desired properties, which the system does not provide, i.e. why the reconfiguration is supposed to happen in the first place. Therefore, the reconfiguration-design task can be defined as:

Definition 30 Given that the user‘s requirements have changed, the goal of the reconfiguration-design task is to compute a new configuration, where most parts of the existing configuration can be preserved. A reconfiguration-design problem solving task (*RDPS*) is defined as:

$$\overline{GS} = RDPS(M, E, S)$$

where M is a model of a system, E is an existing system (that satisfies the constraints of the model M), S is a set of required constraints and $\overline{GS} = \{GSi\}$ is a set of possible solutions for the reconfiguration task, each of them containing a set of actions that have to be applied to the system E in order to satisfy the constraints S.

Since the ontology engineers' requirements for a change are applied to an existing ontology resulting in modification of this ontology, the semantics of change problem can be treated as a reconfiguration-design problem solving task in which a model of the system is defined through an ontology consistency model, the existing system is the given ontology and the user's requirements, expressed as ontology changes, correspond to the required constraints.

Definition 31 The declarative approach for the semantics of change phase *DeclarativeSemanticsOfChange* is defined as:

$$\overline{GCH} = DeclarativeSemanticsOfChange(M, O, UserRequest),$$ where:

- M is the ontology consistency model defined through the set of constraints (see section 2.3) that each ontology has to satisfy;
- O is a concrete ontology that has to be changed;
- *UserRequest* is a user's request for a change that is defined in section 4.3.4;
- $\overline{GCH} = \{GCH_i\}, 1 \leq i \leq m, GCH_i \subseteq CH$, where $\overline{GCH}$ is a set of the possible solutions for the evolution problem, m is the number of solutions and each solution GCH_i is represented as a set of changes.

Since each result of the semantics of change is a set of changes GCH_i needed to be applied in order to keep the consistency (M*)* of the ontology (O), the modified ontology can be obtained as:

$$MO = GCH_i \circ O = GCH_i(O)$$

where MO is a modified ontology and ° is an operator performing the application of changes from GCH_i on the ontology O.

4.3.4 Request Formalisation

Our approach requires an extension in the request specification. Contrary to all the existing ontology evolution systems, where an ontology engineer specifies her request as a collection of ontology changes that have to be performed, we express a request in the following way:

Definition 32 The user's request for a change *UserRequest* is defined as:

$$UserRequest = (posExtCH, negExtCH),$$

where:

- *posExtCH* is a set of "*extended*" ontology changes that must be performed;
- *negExtCH* is a set of "*extended*" ontology changes that must not be performed.

An "*extended*" ontology change is an elementary or composite ontology change in which one argument or more arguments can be the symbol "*". The "*" is a wild card for all valid interpretations of an ontology entity on this position in that ontology change. "*Extended*" changes make the specification of the ontology engineer's request more compact since an "extended" change represents a set of ontology changes.

Returning to the example from Figure 24, the "extended" ontology change *RemoveSubConcept("*","Student")* represents the following set of changes: *RemoveSubConcept("PhD Student", "Student")*, *RemoveSubConcept("MSc Student", "Student")* and *RemoveSubConcept("BSc Student", "Student")* since the concept "*Student*" has three subconcepts: "*PhD Student*", "*MSc Student*" and "*BSc Student*".

To interpret an extended ontology change in terms of elementary or composite ontology changes we define the following function:

- the function *substitution:* $ECH \rightarrow 2^{CH}$, where *ECH* is a set of extended ontology changes and *CH* is a set of ontology changes.

Consequently, the user's request for a change can be represented as:

$$UserRequest = (posCH, negCH),$$

where:

- $posCH = \bigcup_{\forall c \in posExtCH} substitution(c)$, is a set of ontology changes that must be performed;

- $negCH = \bigcup_{\forall c \in negExtCH} substitution(c)$, is a set of ontology changes that must not be performed.

As can be noticed, the ontology engineer's request contains two sets of conditions[48] that specify what has to be done during the resolution. However, these two sets of changes have different semantics. The *posCH* differs from the *negCH* in that the changes from the *posCH* must be satisfied, while *negCH* must not be violated.

By taking into account extended ontology changes, the role of the request formalisation module shown in Figure 33 has to be extended. It has to decompose a request into a set of ontology changes. The optimisation of a sequence is not required since the declarative approach can cope with any arbitrary request. However, in order to speed up the resolution process, some assistance is included e.g. through the ranking of changes based on the number of inconsistencies they may produce.

By specifying a request for a change with a set of conditions that have to be fulfilled and without specifying the way to resolve it, we consider the semantics of change phase of the ontology evolution process as a reconfiguration-design problem solving task explained in the next section.

4.3.5 Change Resolution

For resolving a reconfiguration-design problem problem-independent search methods are used [152]. In order to solve a problem, one needs to explicate the underlying search space and to apply a search algorithm within that space. Since this problem is not tractable in general, heuristic knowledge that guides the searching process to the desired solutions has to be exploited.

Since the semantics of change phase of the ontology evolution process is considered as a reconfiguration problem (see section 4.3.3), we reuse the forward searching method [110].

[48] The request for a change cannot be satisfied in the case when these two sets are contradictory.

We configure this method by defining an evolution graph that is used for searching and by using heuristics that are necessary for an effective search.

Evolution Graph

Definition 33 An evolution graph is defined as:

$$EG := (NO, EC, O, User\,Request)$$

where:

- $NO = \{O_i\}, 1 \le i \le n$[49], is a set of nodes and each node represents an ontology O_i;
- $EC = \{ECk\}, 1 \le k \le m$[50], is a set of labelled edges, which are defined as:

  $$EC_k=(O_i,Ch_j,O_l),\ O_l=Ch_j(O_i),$$
  $$Ch_j \in CH,\ O_i,O_l \in NO,\ 1 \le i,l \le n,\ i \ne l$$

  which means that an edge between two nodes O_i and O_l exists, only if there is a single ontology change Ch_j which transforms the ontology O_i into the ontology O_l;
- $O \in NO$ representing the given ontology is a start node of a graph;
- *UserRequest* is a user's request for a change (see section 4.3.4).

By defining an evolution graph we represent the semantics of change problem as a state-space graph, in which a node represents a state of a world (i.e. an evolving ontology), and an edge represents changing from one state of the world to another (i.e. an edge is a change that transforms a source ontology into a target one).

In order to explain how the evolution graph is used to solve semantics of change problem, we introduce the following definitions:

- a path $PATH(O_i,O_l)$ from the node O_i to the node O_l is an alternating sequence of nodes $\{O_j\} \subseteq NO$ and edges $\{EC_j\} \subseteq EC$ with

  $$PATH=O_0,EC_1,O_1,\ EC_2,...,EC_p,O_p$$

  such that $O_0=O_i$ and $O_p=O_l$, where $EC_j=(O_{j-1},\ Ch_j,\ O_j)$ joins nodes O_{j-1} and O_j, $j=1,p$;
- a set of all possible paths between two nodes[51] O_i and O_l denoted as $PathNodes(O_i,O_l)$;
- a set of all possible paths $PathAll = \bigcup_{\forall Oi,Ol \in NO, 1 \le i,l \le |NO|} PathNodes(O_i,O_l)$;
- the function $changesOnPath: PathAll \rightarrow 2^{CH}$, that returns the set of changes that are on the path;
- the set of goal nodes *GOAL* as:

  $$GOAL = \{GOAL_k \ / GOAL_k \in NO\},$$

  where a node O_l is a goal node (i.e. $GOAL_k=O_l$) if:

[49] n is the total number of nodes in an evolution graph.
[50] m is the total number of edges in an evolution graph.
[51] There might be many paths between the two nodes.

$$consistency^{52}(O_l,M)=true \wedge \forall p \in PathNodes(O,O_l)$$
$$(posCH \subseteq changesOnPath(p) \wedge$$
$$\neg(\exists Ch_j \in negCH \wedge Ch_j \in changesOnPath(p)))$$

where O is a given ontology, *posCH* and *negCH* are elements of a users' request for a change *UserRequest(posCH, negCH)*. *posCH* contains changes that must be done and *negCH* contains all changes that must not be done.

Since the semantics of change problem can be transformed into the problem of finding all paths from the start node O to the goal nodes *GOAL* in an evolution graph, the result of the semantics of change is specified as:

$$\overline{GCH} = \{ changesOnPath(p) \mid \forall p \in PathNodes(O,GOAL_k) \wedge \forall GOAL_k \in GOAL \}$$

Therefore, the solution to the semantics of change problem consists of a set of ontology changes between a start node and a goal node.

In general, to carry out a search in a graph, the following aspects have to be considered [110]:

1. a method for generating the potential solutions;
2. a definition of a potential solution;
3. a way of checking whether a potential solution is a solution.

We adapt these general aspects of searching to the evolution graph since it defines the search space. They are resolved in the following way:

1. The generation of a potential solution is realised as a node expansion. A node is expanded with an ontology change only if the preconditions of the change are satisfied. Therefore, the preconditions of an ontology change describe the conditions that make this change relevant to the situation;
2. A potential solution is obtained by applying an ontology change. Therefore, a next node satisfies the postconditions of a change that is represented as an input edge into this node;
3. A potential solution is a solution if:
    - the corresponding ontology is a consistent ontology (i.e. it satisfies the set of ontology constraints) and
    - all the user's defined conditions (e.g. changes that must be done) are satisfied.

Therefore, the peculiarities of the ontology evolution problem are built into the definition of edges and in the definition of the goal nodes. Since the preconditions of a change Ch_j indicate the initial state prior to the execution of a change and the postconditions indicate the state after a change is applied, then an edge $EC_k=(O_i,Ch_j,O_l)$ exists if the ontology O_i satisfies the preconditions of the Ch_j and the ontology O_l satisfies the postconditions of the Ch_j. This is shown in Figure 34 by assuming that a change Ch_j is a request for the removal of the concept "*Student*" (i.e. *RemoveConcept("Student")*).

On the other hand, a leaf node of an evolution graph is a goal node $GOAL_k$, if the ontology $GOAL_k$ is a consistent ontology (i.e. if it satisfies the set of ontology constraints M), and all the user's defined conditions (i.e. changes that must or must not be done) are satisfied. Since it holds:

[52] The consistency function is defined in section 4.1.

$$GOAL_k = Ch_{kp} \circ Ch_{kp-1} \circ \ldots \circ Ch_{k1} \circ O = Ch_{kp}(Ch_{kp-1}(\ldots(Ch_{k1}(O))\ldots)),$$

where O is the given ontology and $p = |GCH_k|$, the k[th] solution is defined as:

$$GCHk = \{ CHki \}, \text{ i=1,...p,}$$

and $GOAL_k$ would be the resulting ontology.

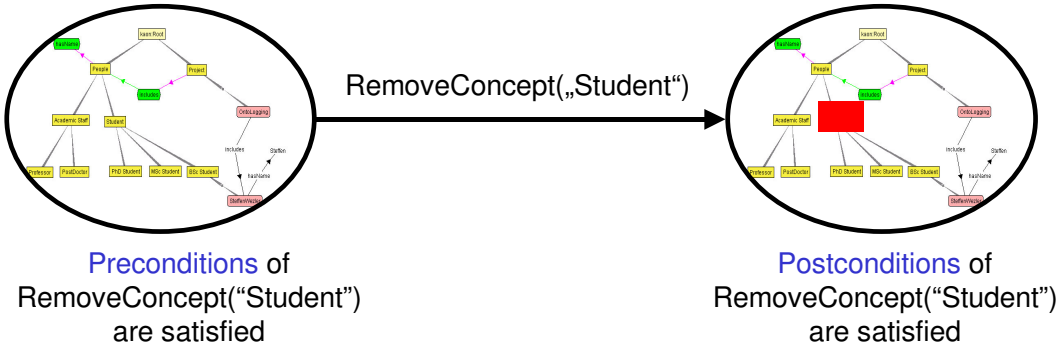


Figure 34. A part of an ontology evolution graph indicating specificities of its edges

Heuristics for the Search Improvement

The configuration/reconfiguration task can be considered as a search problem using the above mentioned types of inputs and outputs (see Definition 29). This search space is restricted to several steps using the various types of inputs [143]. A set of possible components and possible connections between these components are fixed and given beforehand. This restricts the search space to a possible re-configuration space. The constraints restrict the possible re-configuration space to the valid re-configuration space. The user-requirements restrict this valid re-configuration space to the suitable re-configuration space.

In this section we adapt this strategy to the ontology evolution problem and introduce several heuristics to restrict or divide this space further (see Figure 35). Heuristics are principles used in making decisions when all possibilities cannot be fully explored. They are used in order to avoid expensive reasoning about the impact of changes. Therefore, they guide the search process to the goal. In this way, the main disadvantage of the approach, i.e. using a search space that is exponential in the path length is significantly softened.

An evolution graph consists of the nodes that are generated when all possible changes are iteratively applied to the current node that represents a current state of an ontology. The issue is how to manage the expansion and the search through the search space so that the solution can be found efficiently. Indeed, a node is expanded with an edge only if this change satisfies the heuristics.

Let’s define the set $PCH = \{ Chj \,|\, preconditions(Oi, Chj) = true \}$ as a set of possible changes that might be applied to the ontology O_i, i.e. the ontology O_i satisfies the preconditions of all changes $Ch_j(args_j, prec_j, post_j)$. We define the set of changes that are reasonable to be applied

to O_i as $ACH = \{ Ch_j \} \subseteq PCH$, where each change $Ch_j(args_j,prec_j,post_j)$[53] has to fulfil the following set of constraints:

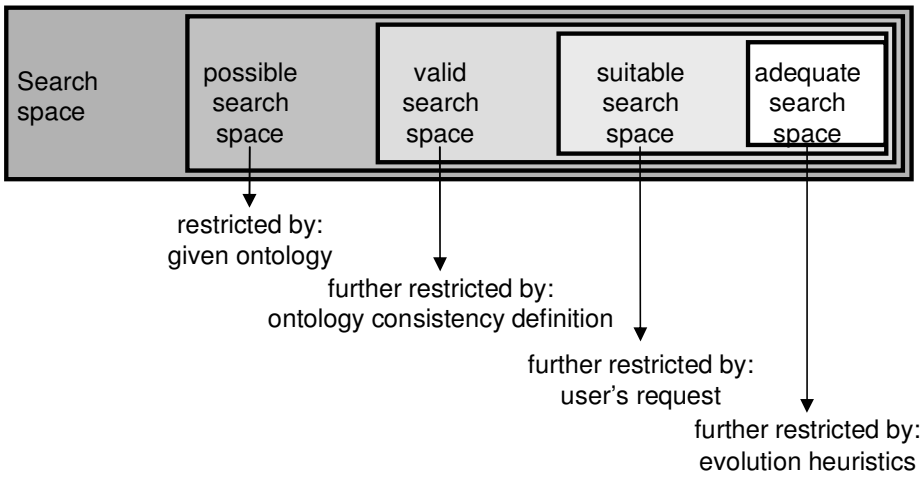


Figure 35. Ontology evolution as a search problem

- **Locality heuristics**:

$$\forall p \in PathNodes(O,O_i)$$
$$(createSet(args_j) \cap$$
$$\{x / x \in createSet(args_k) \wedge Ch_k(args_k,prec_k,post_k) \in changesOnPath(p) \} \neq \emptyset)$$

where *createSet* is a function that transform a tuple into a set. By applying this function to the arguments $args_k$ of an ontology change $Ch_k(args_k,prec_k,post_k)$, the set containing elements of the argument is obtained (e.g. *createSet(("Student", " Person")) = { "Student", "Person" }*). Further, $args_j$ are arguments of the change $Ch_j(args_j,prec_j,post_j)$ that is considered and $args_k$ represents arguments of each change $Ch_k(args_k,prec_k,post_k)$ on a path p from the start node O to the node O_i.

This heuristics is based on the assumption that a change should be applied only if at least one of its arguments is already visited during the resolving of a change request. Without the usage of this heuristics many undesired solutions would be generated such as the removal of all entities except for the root concept for any of the requests of ontology engineers since the precondition for a removal is that an entity exists in an ontology.

Returning to the example from Figure 24, after the deletion of the concept "*Student*", invariants IC_4 and IC_5 (see section 2.3) are invalidated. As a consequence of that, several problems arise:

- undefined child concept for the concept "*Person*";
- undefined parent concept for the concepts *"PhD Student", "MSc Student" and "BSC Student*".

Only changes "related" to these problems should be generated. These problems can be resolved by the removal of the subconcept relationship between the concept "*Student*" and all

[53] Note that the parameters of the change $Ch_j(args_j,prec_j,post_j)$ are used in all formal definitions of the heuristics.

its parents and children. Thus, the application of the following set of changes is needed (see Figure 36):

RemoveSubConcept("Student", " Person"), *RemoveSubConcept("PhD Student", "Student")*, *RemoveSubConcept("MSc Student", " Student")*, *RemoveSubConcept("BSc Student", "Student")*.

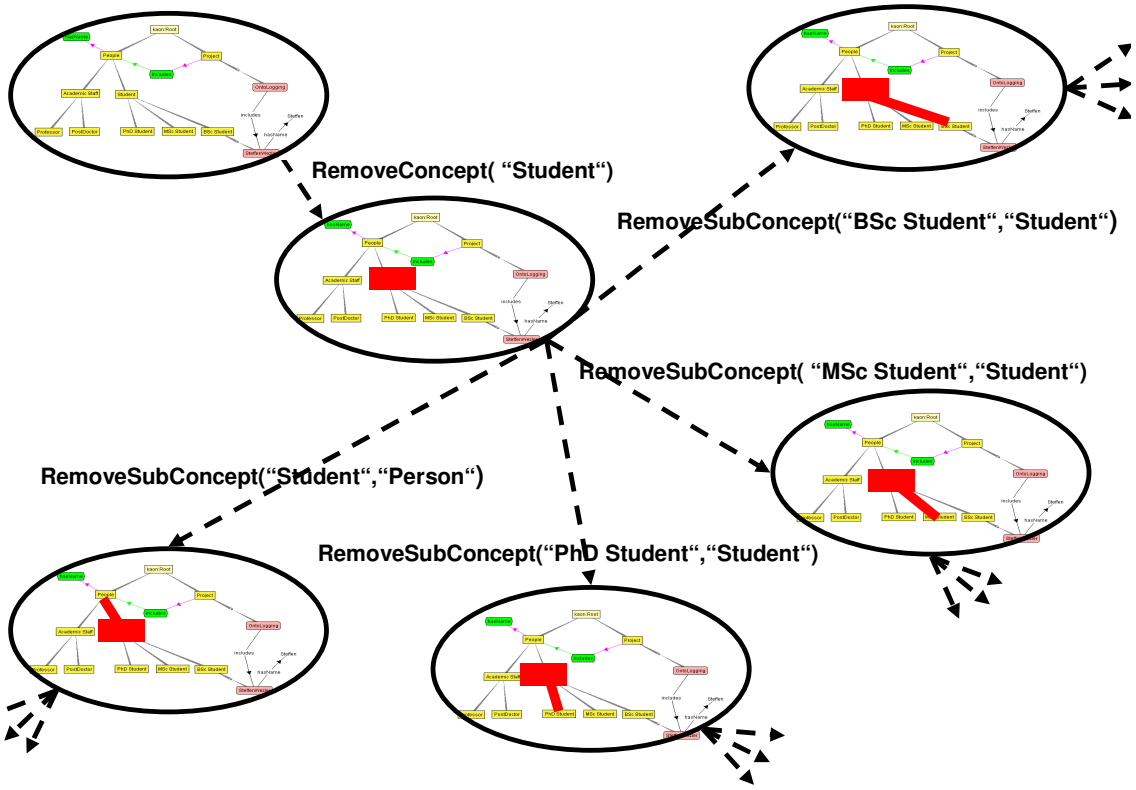


Figure 36. A part of the evolution graph for the removal of the concept "Student" for the ontology shown in Figure 24

The common characteristic of all induced changes is that they contain the concept "*Student*" as one of arguments as required by the locality heuristics. There are many other changes whose preconditions are satisfied such as *RemoveConcept("Project")*, *RemoveSubConcept("Professor", "Academic Staff")*, *AddConcept("Industrial Project")*, etc. However, they cannot be applied since the intersection of the arguments of these changes and the arguments of the already applied change *RemoveConcept("Student")* is an empty set.

- **Cycle detection heuristics**:

$$\forall p \in PathNodes(O,O_i)$$

$$\neg(\exists Ch_k(args_k,prec_k,post_k) \in changesOnPath\ (p) \wedge$$

$$inverseChange(Ch_j(args_j,prec_j,post_j),\ Ch_k(args_k,prec_k,post_k)) = true\ )$$

where the function $inverseChange:CH \times CH \rightarrow \{true, false\}$ returns true, if its arguments (i.e. $Ch_j(args_j,prec_j,post_j)$ and $Ch_k(args_k,prec_k,post_k)$) are inverse changes (see Definition 15), otherwise it returns false.

This heuristic enables the avoidance of the expanding any branch that would result in the generation of the previous version of the same ontology. Since it prevents that the paths in an

evolution graph should contain inverse changes (see Definition 15), it relieves us from worrying about interminable searching, i.e. solutions that comprise the infinite set of changes. For example, none of the paths can contain *RemoveConcept("Student")* and *AddConcept("Student")* changes since they are inverse changes. After the removal of the concept "*Student*" the preconditions for the addition of the concept "*Student*" are satisfied and vice versa.

- **User-defined heuristics**:

$Ch_j(args_j, prec_j, post_j) \notin negCH$,

where *negCH* is an element of a users' request for a change *UserRequest(posCH, negCH)*.

This heuristics is used to quickly inform the search about the direction to the goal. It is focused on the more complex needs of an ontology engineer specified with *negCH* - a set of changes that must not be performed (see section 4.3.4). Regarding the example depicted in Figure 24 it means that there is no edge that corresponds to the *RemoveConcept("PhD Student")* or *RemoveConcept("MSc Student")* changes.

Beside heuristics used for constraining the search space, we have introduced several heuristics to define ordering upon possible solutions. Let's consider two solutions GCH_i and GCH_j of the given semantics of the change problem. We define the partial ordering relation $\leq$ between $GCH_i=\{Ch_{ik}\}$ and $GCH_j=\{Ch_{jl}\}$ in one of the following ways:

1. $GCHi \leq GCHj \leftrightarrow |GCHi| < |GCHj|$;
2. $GCHi \leq GCHj \leftrightarrow$

   $$|\{ Chik | Chik \in GCHi \wedge Chik \in RmCH \}| < |\{ Chjl | Chjl \in GCHj \wedge Chjl \in RmCH \}|$$

   where $RmCH \subset CH$ is a set of changes causing the removal such as *RemoveConcept*, *RemoveSubConcept*, *RemoveProperty*, etc.;
3. $GCHi \leq GCHj \leftrightarrow \sum_k cost(Chik) < \sum_l cost(Chjl)$, where the *cost* function is defined as $cost : CH \rightarrow R^+$, CH is a set of ontology changes, R^+ is a set of positive real numbers;
4. $GCHi \leq GCHj \leftrightarrow |Ei| < |Ej|$, where $E_i=C_i \cup P_i \cup I_i$ ($E_j=C_j \cup P_j \cup I_j$) is a set of entities of the ontology O_i (O_j), $O_i=GCH_i(O)$ and $O_j=GCH_j(O)$, respectively.

The previously defined heuristics enable the ordering of solutions according to the (1) number of needed changes, (2) number of removals, (3) cost of changes, or (4) size of the resulting ontology. Consequently, they facilitate the choice of the solution that is more appropriate to the need of an ontology engineer.

Indeed, through the definition of the ranking heuristics, our approach associates a qualitative assessment with each solution indicating the complexity of the resolution way. An ontology engineer can use these assessments to set thresholds on the comparison algorithms or to focus attention on less complex solutions or those with less impact on the ontology.

Whereas the locality, the cycle detection and the user-defined heuristics are necessary to reduce the search space, the ranking heuristics are mutually exclusive. Typically a particular ranking heuristics is chosen by an ontology engineer at the start of the ontology evolution process. Moreover, the selected ranking heuristics is used during the search through the graph, which leads to a more effective and efficient search. Let's assume that the minimal number of removal is selected as the ranking heuristic. This heuristic can be applied after one solution is

found. The solution is represented as a goal node in an evolution graph. Each node in the evolution graph that might be a part of another solution is compared to the goal nodes. The comparison is done by checking the number of removals on the path to the root node. If this number is greater than the number of removals to the goal nodes, this node cannot be taken into the consideration anymore. Otherwise, if this number is smaller and the considered node is the goal node, then it becomes the new solution and the previously found solution is rejected.

There is a trade-off between the amount of work it takes to derive all solutions and how accurately found solutions realise a request of an ontology engineer. The price that has to be paid is the time required to perform the request. The heuristic knowledge guiding the searching process to the desired solutions has to be exploited. Whereas the mandatory heuristics prevent the generation of an exponential number of the solutions, the ranking heuristics improve the performance in the general case. The application of the ranking heuristics on the results obtained through the mandatory heuristics may be inefficient when the number of chosen solutions is much smaller than the number of the produced solutions. Thus, we combine the mandatory and the ranking heuristics during the request resolution, which leads to more efficient searching. Indeed, the defined heuristics are sufficient to make the approach usable in practice.

The benefits of the presented approach are manifold: an easier extension of the ontology evolution system with new composite changes, a more efficient maintenance of the ontology evolution system providing several ranking heuristics in order to choose the most suitable solution, to name but a few.

4.3.6 Complexity

An ontology change is an extremely expensive process both in terms of time as well as system resources. A simple change such as a concept removal for a large number of instances of that concept can take long for processing. Hence a complex change as specified by the declarative request specification can take even longer. Further on, we present a short discussion about the complexity of the declarative approach for the semantics of change.

The declarative approach is based on the graph search method. Search methods generally fall into the class of NP-hard problems since they require examination of large search spaces. Therefore, they are usually used when the problems cannot be solved directly (i.e. without searching) by simply applying the appropriate algorithms. Knowing such limitations of the search methods and the fact that the ontology evolution is a very practical problem, we tried to define the tractable solution. Indeed, we introduce a set of heuristics (see section 4.3.5) used to guide the search to the goal.

The complexity of the approach depends on two factors:

- how many steps it takes to find a solution (i.e. the number of nodes generated/expanded);
- how many nodes it takes to solve a problem in all possible ways (i.e. maximum number of nodes in a graph).

The first factor represents the number of the generated changes needed to achieve the consistency whereas the second factor determines the number of the possible solutions. A more careful analysis of these factors indicates that the complexity can be measured in terms of:

- *md* – maximum depth of an evolution graph, i.e. the length of a path between a source node and a goal node (i.e. a solution),

- *b* – maximum branching factor of the evolution graph.

Therefore, the complexity is $O(b^{md})$. The worst-case for *md* is the concept removal, which causes the removal of all subconcepts. It means that the value for *md* is equal to the worst-case for the procedural approach for the semantics of change (see section 4.2.5). The branching factor indicates the number of ways for resolving an inconsistency. The worst-case for *b* is *3* since the inconsistency related to the orphaned concepts (see invariant IC_4*: Concept-Closure Invariant*) may be resolved in three ways (see the resolution point RP1 and its resolution ways in section 4.2.3). For other inconsistencies there is either one or two resolution ways.

Although the approach is very complex, a good heuristics can lead to dramatic improvements. In previous section we mentioned some heuristics used for restricting the search space. They are based on the idea to model meaningful solutions. Here we introduce the additional heuristic information concerning the nodes that are most promising to pursue and the way in which this is to be done. This information, which guided the implementation of the system (see section 7.2), is in the form of a heuristic function:

$$h: NO \rightarrow N^+,$$

where *NO* is a set of nodes in the evolution graph and N^+ is a set of positive integers. The heuristic value is simply derived from the properties of a node *n* in the following way:

$$h(n) = \sum_{in} weight(in) * fulfilment(in, n), 0 \le in \le |IN|,$$

where:

- *IN* is a set of all possible inconsistencies that can be derived from the ontology consistency definition (see section 2.3);
- *weight:* $IN \rightarrow N^+$ and *weight(in)* indicates the number of possible resolution ways for the inconsistency *in*;
- *fulfilment:* $IN \times NO \rightarrow \{0,1\}$, where *NO* is a set of nodes of an evolution graph. The value *1* of the function *fulfilment(in, n)* points out that the ontology in the node *n* has the inconsistency *in*. The value *0* is used when the inconsistency *in* does not exist in the node *n*.

This heuristic function is used to inform the search about the direction to a goal. A search algorithm to define the order of a node expansion and the way in which a node is expanded uses it. Our search algorithm selects a node with the minimum value for the heuristic function *h* since it appears to be closest to the goal. In this way, the search algorithm can find the solution with fewest edges (i.e. additional changes) first.

Further, the number of the nodes in the evolution graph varies tremendously of the order in which the inconsistencies are resolved. Therefore, the node is expanded with an edge (i.e. a change) that resolves an inconsistency with the minimum *weight*. Namely, the complexity of the evolution graph is exponential in the branching factor. By defining the ranking between inconsistencies and selecting the inconsistency with the minimum weight, the smallest breadth of an evolution graph is achieved.

Practically, we adopted the A* search algorithm [110] that can find optimal path as the first path. The definition of the optimality is specific for the ontology evolution problem. This problem-specific information is built into the node selection and the node expansion. The specificities of the node selection are included in the definition of heuristic function *h*. The node expansion is realized through the ranking of inconsistencies based on the function *weight*.

The evaluation of this system on the real-world examples (see section 4.4) showed that these heuristics are helpful enough to solve the problem in practical situations. It is worth pointing out that the analysis of the complexity has given an insight to the designer of procedure with a clear indication of the problems that are difficult to deal with as well as general methods to cope with them. It led directly to practical implementation by the search method of better control strategies (i.e. the heuristic function *h*) and optimisation techniques (i.e. the ranking of inconsistencies through the function *weight*).

4.4 Comparison of the Procedural and the Declarative Approaches

In this section we provide the comparison of the two proposed methods for the semantics of change phase of the ontology evolution process. The assessment is based on the following assumptions:

- both systems are able to perform the same set of changes (see Table 1 and Table 4);
- they offer the same possibilities for controlling and customizing the change resolution (see evolution strategies in section 4.2.3 and meta-information in section 4.3.5).

To compare the proposed approaches for the semantics of change we define the following subjective criterion:

- *efficiency* - the degree to which a system performs the desired request for a change with minimum consumption of resources (e.g. time, space, etc.).

This criterion is chosen to assess how an ontology evolution system meets the needs of ontology engineers that use an ontology evolution system to modify an ontology. They are end-users for the ontology evolution system. By taking into account ontology engineers we are able to measure the benefits at the run-time, i.e. benefits from the usage point of view.

4.4.1 Efficiency of the Approaches for the Semantics of Change

The efficiency of a system is its ability to do a task within certain time and space constraints. It is related to the bottlenecks that may arise in a system when doing certain tasks. Since the ontology evolution is a practical problem, the semantics of change must be guaranteed within run-time. The time needed to perform a request consists of the time for the request specification and the time for the request resolution. Therefore, we split this comparison criterion into two specialised measures:

- *the request specification efficiency* – it estimates how easily and inexpensively a request might be specified;
- *the request resolution efficiency* – it estimates the ability of a system to realise a request without wasting time and effort.

We performed an experiment to compare both the proposed approaches for the semantics of change with respect to these two types of efficiency.

Experiment Setup

For the experiment we presented the subjects with an ontology and asked them to modify it in a specific way. The task required them to iterate through a set of requirements for the update,

understanding what they were by representing them as ontology changes and to use ontology editors to apply these changes.

Source Ontology: We have used the VISION ontology that consists of more than 100 concepts, 50 properties and about 900 information resources (the web page of concrete persons, projects, etc.). Each of the information resources is related to a concrete instance in the ontology (e.g. to the project "*OntoLogging*"). This ontology was developed in the scope of the EU-funded VISION[54] project, which should provide a strategic roadmap towards the next-generation organisational knowledge management.

Since the acquisition of the information related to the domain was an ongoing process, it happened that new information could not be represented in the existing ontology. For example, one request was to include industrial projects. We had to split the concept "*Project*" into the concepts "*Research Project*" and "*Industrial Project*" and to move all existing instances of the concept "*Project*" to the concept "*Research Project*".

Tools: We have used three different versions of the KAON tool suit (see chapter 7 for more information about implementation details):

- *KAON-Basic* – the version of the KAON system that provides only one way for resolving changes, as all the other ontology evolution systems;
- *KAON-Procedural* - the version of the KAON system that incorporates evolution strategies allowing the users to customise an ontology evolution process;
- *KAON-Declarative* – the version of the KAON system that allows the declarative specification of a request.

Users: There were 9 users in the experiment. They were split into three groups: 3 subjects used the KAON-basic, 3 subjects used the KAON-Procedural and 3 subjects used the KAON-Declarative. Approximately half of the users had experience with the KAON tool suit. We gave the other half a brief tutorial on using the KAON system, focusing on the tasks that they will need for the experiment.

Training: Prior to the experiment we gave users a small ontology as an example and asked them to spend a short time (not more than 10 minutes) familiarizing themselves with the corresponding tool. We did not explain to them what all implemented changes and/or evolution strategies and/or meta-knowledge meant. We left it to them to figure it out during the training. We assumed that the user interface and the naming convention were intuitive enough to be able to understand what they meant.

Task: We gave all groups the same ontology and a set of requests specified in a natural language they had to perform. For example, one task was to move all instances of the concept "*Company*"[55] to its sibling concept "*Public sector*". Another task was to delete the concept "*Research Organisation*" so that all its instances[56] were attached to its super concept "*Organisation*".

We measured the time it took them (i) to specify the request, (ii) to realise the request and (iii) to complete the task. Afterwards we calculated the average number of changes for the request formalisation and the number of undoing changes.

[54] www.km-vision.org
[55] There are 121 instances of the concept "*Company*".
[56] There are 67 instances of the concept "*Research Organisation*".

Results and Discussion

Table 7 shows the summary of the results. The declarative approach saves time by specifying changes declaratively but it requires more time to calculate possible solutions. On the other hand, for a complex change, the non-declarative specification of a request is a time-consuming and an error-prone task. It took the group using the KAON-Procedural more time on average to specify and, consequently, to complete the task than the KAON-Declarative group.

Regarding the request resolution, it should be noted that the KAON-Declarative is much less efficient than the KAON-Procedural. This is because the KAON-Declarative tool is purposely general and thus applicable to a wide variety of changes. On the contrary, the KAON-Procedural is designed for a specific problem (i.e. ontology changes). This pre-defined set of ontology changes is "compiled" into the procedures, resulting in highly efficient performance. In addition, the meta-level control (i.e. evolution strategies) ensures that the system performs only computations that increase utility.

Finally, even though the computational complexity of the declarative approach is greater, its usage leads to the faster realisation of the user's request. The above-mentioned tasks, for example, heavily depend on the number of instances. The KAON-Procedural requires the specification of a separate change for each instance, which can take more time for a large number of instances. For example, the first previously mentioned task about moving all instances of the concept "*Company*" to its sibling concept "*Public sector*" requires the 121 applications of *MoveInstance* change, since the concept "*Company*" has 121 instances. The evolution strategies built in the KAON-Procedural enables to shorten the number of requested changes. For example, the second task related to the deletion of the concept "*Research Organisation*" while attaching its instances to the concept "*Organisation*" can be specified with only *RemoveConcept("Research Organisation")* change by selecting the elementary evolution strategy EES12. This strategy enables the reconnection of instances of a concept that has to be deleted to its parent concept. However, the choice of the right evolution strategy is not a trivial task.

Table 7. The result of the evaluation. The first column shows the time needed to perform a task. It includes the time for specifying all necessary changes as well as the time for resolving them. The second column indicates the number of the changes needed to realise a request. The last column demonstrates whether the requests were fulfilled or not.

	Average time per task in sec[57]			**Average number of steps per task**	**Average number of undoing changes per task**
	a)	**b)**	**c)**		
KAON-Basic	47,3	8,4	55,7	15,8	1,5
KAON-Procedural	18,6	6,4	25	10,4	0
KAON-Declarative	5,1	15,2	20,3	1,7	0

[57] a) and b) denote time which is required for the request specification and the request resolution, respectively. Total time is denoted by c).

On the contrary, the declarative approach enables the grouping of requests through the usage of the extended changes. For example, the first previously mentioned task can be solved by the application of only one change *MoveInstance("*","Company",* "*Public sector*"*)*, which will be automatically translated into a set of changes for the concrete instances.

The worst result was shown by the KAON-Basic tool. The average time per task in the KAON-Basic group was twice as long as in the other groups. This tool simulates how the ontology evolution is realised in all existing ontology evolution systems. The result indicates that both proposed ways for resolving the semantics of change problem (i.e. the procedural approach and the declarative approach) are better that the existing systems.

The evaluation study showed that the complexity built into the ontology change resolution improves users' abilities to fulfil an ontology evolution task. The declarative approach is especially useful in the case of more complex users' requests. Such requests are easier (i.e. faster) to specify declaratively without considering how to realise them. However, if a request can be fulfilled with only one elementary or composite change, it would be better to use the programmatic approach since all solutions will be found in a shorter time.

Additionally, we analysed the time variance for the members of the same group. It showed that most measurements for the KAON-Declarative and the KAON-Procedural are very close to one another, while the distribution of measurements is much greater for the KAON-Basic tool. This observation indicates that the KAON-Declarative and the KAON-Procedural stimulate people to work in a very mechanical manner. The larger distribution for the KAON-Basic tool could also be an indicator that people try more to understand the semantics of changes. Further, the experiment also showed that users of the KAON-Procedural changed the evolution strategies often. This points out that the meaning of the evolution strategies was understandable to most of the users.

The main advantage of the declarative approach is its reduction of the user's involvement in an ontology evolution task. The user has to specify only the request for a change and, eventually, to select the solution in the case that many equally ranked solutions for a given user's request exist. Consequently, the declarative approach can be considered as an automatic one. On the contrary, other approaches require user's interventions, which make them more complex from the user's point of view. For example, in these approaches, an additional change has to be specified for each instance that has to be moved.

Regarding the undoing the requests for a change, only the users of the KAON-Basic tool had to perform the same requests multiple times since the order of changes and the semantics of changes were not intuitive to them.

Conclusion

Based on the experimental results related to efficiency here we discuss the advantages and disadvantages of the procedural and the declarative approaches for the semantics of change.

The procedural approach can be considered as a command-based approach since an ontology engineer has to understand the semantics of ontology changes (e.g. the difference between the *RemoveProperty* and *AddProperty* change and the *MoveProperty* change – the first request causes all the property instances to be lost whereas the second one preserves some property instances). Therefore, she needs to be careful in choosing which changes to use. The problem is that this approach focuses on the editing *process* rather than the editing *result*. For that reason, the efficiency of the request specification depends on the complexity of a request. If a request can be fulfilled with one (elementary or composite) change, then the efficiency of the request specification is high. However, for more complex requests this kind of efficiency is

very poor. On the other hand, the efficiency of the request resolution is satisfactory. Due to the direct connection of a resolution procedure to a change, the procedural approach for the semantics of change is highly efficient in terms of computation. Neither the number of changes nor evolution strategies introduce an appreciable decline in this efficiency.

Another alternative, which is realised through the declarative approach, is to allow an ontology engineer to modify an ontology as necessary and then to calculate all the additional changes, thereby focusing on the editing result rather than the editing process. The advantage is that the ontology engineer can specify her request declaratively, focusing on producing the correct new entity definitions without worrying about the exact process used to create those new definitions. Therefore, the efficiency of the request specification is always high. In comparison with the procedural approach, the declarative approach is more efficient (with the respect to the request specification) since it has smaller variance.

The disadvantage is that the system must now infer how the entities have changed instead of being explicitly told. The inference processes heavily depend upon the number of rules necessary to characterise the desired behaviour of the system. The more rules exist, the slower the system becomes due to seek and match time. We have developed algorithms to perform these inferences by searching in the evolution graph search space to identify the additional changes, needed to preserve the consistency. The evolution graph search algorithm uses the ontology consistency definition and a set of heuristic knowledge to infer the needed changes. The search over any problem space is combinatorily explosive. Improved efficiency is achieved by the heuristics, which decrease the range of the overall search. Although, experimentation with the proposed algorithms has demonstrated that they can identify a wide variety of changes successfully, the resolution efficiency is still critical. It is limited by the time required for the reasoning procedure to terminate. Consequently, the declarative approach for the semantics of change requires more time for the request resolution that it is needed for the procedural approach (see also Table 7).

Therefore, each approach has its own advantages and disadvantages. The choice of the most suitable approach primary depends on the problem that has to be resolved.

4.5 Related Work

Heflin [53] points out that ontologies on the Web will need to evolve and he provides a new formal definition of ontologies for the use in dynamic, distributed environments. He presents SHOE, a web-based knowledge representation language. He maps it to the first order logical theory. Moreover, the author defines features of the SHOW language that address the problem of evolving ontologies. He discusses the relevance of each type of revision (i.e. the addition or the removal of components) to data sources (i.e. instances) and queries, but not on the ontology itself. Although the author is aware of the fact that, e.g., the concept removal may provoke that some query cannot be asked or will return fewer results, he does not propose how to resolve that problem. Moreover, the author claims that any entities can be safely added to an ontology since the addition results in new clauses to the logical program and the first-order logic is monotonic. However, for instance, experiences from the knowledge-based systems show that (i) a rule can cause a cycle in the reasoning; (ii) two rules might be identical or contradictory, and, (iii) one rule might subsume other rule, etc. Therefore, the monotonicity of the first-order logic does not mean that the consistency constraints cannot be corrupted. It only indicates that the results derived from querying a new version of a model, which can be an inconsistent model as well may be different from the results of querying the original model. These results may contain additional answers that were not originally intended but also the wrong answers due to the inconsustency of a model. However, this problem is not

treated in the work of Heflin. Moreover, the ontology consistency constraints are not defined at all and consequently there is no approach to forcing them after applying a change. Further, the author formalises neither changes nor the way of their resolution.

In contrast to the ontology evolution that allows access to all data (i.e. to ontology itself and to dependent artefacts) only through the newest ontology, the ontology versioning allows access to data through different versions of the ontology. Thus, the ontology evolution can be treated as a part of the ontology versioning mechanism. The ontology versioning is analysed in [66]. The authors provide a survey of causes and consequences of the changes in an ontology. However, the most important flaw is the lack of a detailed analysis of the effect of specific changes on the interpretation of data, which is a constituent part of our work.

In [65] the authors describe an ontology versioning system that is based on the comparison of two ontology versions in order to detect changes. Even though there are many ways to transfer an ontology into a new version, this system generates only one solution based on the set of heuristics. In contrast to that approach, we may find all ways of resolving changes since the first version can be represented as the start node in the evolution graph whereas the new version is a node in the set of the goal nodes.

Other research communities also have influenced our work. The problem of the schema evolution and of the schema versioning support has been extensively studied in relational and database papers. [113] provides an excellent survey of the main issues concerned. We took these approaches as a foundation for our work just as we investigated the ways in which the techniques and approaches from database schema evolution could be adapted to the ontologies.

A semantic approach to the specification and management of database with evolving schemata in introduced in [37]. The authors formalise a generic object-oriented model for the schema versioning and evolution, define the semantics of schema changes and show how interesting reasoning tasks can be supported. This approach is very similar to our declarative approach for the semantics of change since both of them can deal with arbitrary complex changes and allow the formal checking of the evolution. While our approach is based on the usage of the problem-solving methods, this approach is based on the description logic reasoning. Consequently, the expensive reasoning about impact of changes cannot be avoided. Further, the authors do not address the effects of schema changes on the underlying data instances, otherwise covered by our approach.

A sound and complete axiomatic model for dynamic schema evolution in object-based systems is described in [104]. This is the first effort in developing a formal basis for the schema evolution research. The approach takes into account the key features of types and inheritance. The model can infer all schema relationships from two sets associated with each type. These sets are the known as the *essential supertypes* and the *essential properties*. Essential supertypes are those supertypes in the class hierarchy that must be included in the definition of a type while the essential properties are those properties that cannot be dropped as schema changes are made. This work also describes various dynamic schema policies used by the TIGUKAT to support evolution and how these policies can be defined using axioms. The axiomatic model has been demonstrated to provide a method to support dynamic schema evolution in the object based system by serving as a common, formal underlying foundation for describing evolution in existing systems. We follow this suggestion and apply the similar strategy to ontologies since ontology models exhibit similar characteristics. The formal model is not achieved only through the formal definition of the ontology consistency, preconditions and postconditions of ontology changes. Instead, we model the change resolution as well by definition the evolution graph and by applying the forward search method to it.

An approach to removing a class in object-oriented databases is proposed in [11]. The authors suggest a declarative primitive, *Remove*, that guides the update by the designers' requests. We extend this idea in several ways. Firstly, ontology engineers should be able to control all changes, not only the concept removal. Therefore, we analyse all possible ways for realising all ontology changes and derive the knowledge that allows ontology engineers to set how to do a particular change. Secondly, we abstract the level of the control by introducing resolution points and grouping them into evolution strategies. Finally, the advanced evolution strategies offer means for choosing the most suitable evolution strategy for a given request for a change.

Most of the existing evolution systems provide a fixed taxonomy of possibly complex changes to users, instead of giving them flexibility, extensibility and customisation. This is offered by the SERF approach [16]. The SERF framework allows users to perform a wide range of complex user-defined schema transformations. It combines existing schema evolution changes using OQL (object query language) as the glue logic. However, it requires that the user should specify the actions that have to be done in order to resolve a request, which is an error-prone process providing no guarantees for consistency of the database. Our approach lets the system itself to find solutions based on the preconditions of the changes and the user-defined conditions. In this way we separate the specification from the realisation of a request for a change.

The tool ICC [25] automatically checks consistency in the O_2 system. As input parameters of the tool, the authors define a set of generic modifications called "high level updates". To capture their semantics, the "parameterised updates" are introduced. The user uses them as a syntactic way to communicate with the tool. All the parameterised updates are not primitive changes. The internal purpose of a tool is to decompose the parameterised updates into elementary changes for which the tool knows how to perform the consistency checking procedures. However, this mechanism is not explicit and reusable by the schema manager in order to apply the user-defined modifications. Our declarative approach for semantics of change performs exactly that.

Moreover, research in the ontology evolution can also benefit from the many years of research in knowledge-based system evolution [82]. There are a vast number of techniques (e.g. knowledge refinement, theory revision, validation and verification, etc.) for assisting the development of knowledge-based systems. They follow the paradigm of detecting problems in a knowledge-based system and suggesting repairs. Most of them attempt to correct errors when a knowledge-based system is being developed. In contrast, the purpose of our approach(es) for the semantics of change is to guide an ontology engineer in modifying an ontology that is initially correct. The inconsistencies to be corrected were introduced during the evolution process. Hence, our approach relies on knowledge about this process in correcting the inconsistencies.

A common technique for the knowledge maintenance is to reflect over dependencies between knowledge elements. The question is how to represent the dependencies between them. There are three different approaches: (i) procedural approach, (ii) logical approach, and (iii) network approach. All of them are based on the dependency graph analysis. The differences between these approaches are discussed in [81]. The open issues are how to create a graph and how to use it. We propose two methods for the ontology maintenance that are based on different dependency graphs, i.e. the change dependency graph and the evolution graph. Furthermore, due to trade-off between the request specification and the request resolution, we suggest a hybrid solution that combines the procedural and the declarative approaches. The procedural approach should be applied in the case that there is a change that represents a request. On the other hand, we propose the declarative solution that does not restrict the changes that can be done. The trade-off for this gain in flexibility is the time needed for the request resolution.

However, the evaluation study (see section 4.4) showed that for an ontology engineer it took more time to specify a complex request than to resolve it.

One example of the procedural approach for the evolution of the knowledge-based systems is given in [43]. A knowledge-based system includes a domain model as well as problem-solving methods for achieving goal in that domain. The domain model describes concepts, relations and their instances. A problem-solving method consists of a capability description that indicates the goal that the method is able to achieve, and a body that describes a procedure for achieving those goals. The ETM (EXPECT's Transaction Manager) system identifies typical sequences of changes to knowledge base and represents them in the form of scripts. These scripts capture the way in which the related portions of a knowledge-based system can be changed co-ordinately. Therefore, the system is similar to our procedural approach for the semantics of change. However, unlike the knowledge-scripts that assist users in performing all of the required changes, we go step further by allowing the user to control the way of completing the overall modification as well as by suggesting him the changes that could improve the ontology.

In [140] the authors propose a library of knowledge acquisition (KA) scripts. The scripts capture typical modification sequence that users follow when they modify knowledge bases. The authors identified a good number of KA scripts and found a set of useful attributes to describe and organise them. To find the right level of the generality of scripts, the authors carried on three steps. The first analysis resulted in the KA scripts for each possible syntactic modification. This level was too general to provide useful guidance to users. The second analysis produced KA scripts for typical modification performed by the users. However, this level did not cover a complete set of KA scripts. The third analysis tried to combine the results of the previous two analyses by improving the user support as well as coverage.

We came to similar conclusions during the development of the ontology evolution support. We start by defining ontology changes that ontology engineers can get when modifying an ontology. Further, we found that providing only one way for resolving an ontology change is not satisfactory for most of ontology engineers since they are not able to represent their needs. Therefore, we introduce evolution strategies that allow ontology engineers to choose how to follow up a change. Since this solution is not applicable for an arbitrary request for a change, we define the declarative approach that allows ontology engineers to specify their request at a more conceptual level and guaranties their resolution in the way most suitable for them. In this way both the requirements, i.e. the coverage, which is represented as a set of changes that are supported, and the users' support, which is achieved by providing meta-information to control change resolution, are fulfilled.

4.6 Conclusion

Since modifying an ontology requires a coherent sequence of ontology changes, there is a need for methods that support ontology engineers in co-ordinating these changes and carrying out them correctly. In this section we propose two approaches for dealing with this problem:

1. the procedural approach that allows an ontology engineer to adapt an ontology according to her needs in an effective and efficient way;
2. the declarative approach that addresses the two most important causes of inefficiency of existing ontology evolution systems, namely: (i) how to specify a request for a change in a declarative way and (ii) how to resolve a change to satisfy an arbitrary need of an ontology engineer.

The first solution extends the simple fixed-semantic ontology changes with the predefined set of possibilities to resolve them in the fastest way. This kind of approach has also been shown to be useful for developing an intelligent assistant for common tasks. The second solution represents an effort to formalise the ontology evolution problem. Moreover, we compare the proposed approaches in terms of efficiency.

5 Change Propagation

Ontology development is difficult and time-consuming, and this is even worsened by the fact that ontologies are mostly created from scratch, resulting in the "Babel of Ontologies" problem. This leads to many ontologies modelling the same thing. In order to speed up the engineering and to enable interoperability, ontologies should be built by reusing smaller, well-defined components. However, there is a lack of methods and tools supporting and facilitating ontology reuse.

Although through the reuse the cost of creating ontologies is decreased, the ontology reuse has led researchers to confront problems of local modification. Ontologies are rarely static, but are being adapted to changing requirements. Hence, an infrastructure for management of ontology changes, taking into account dependencies between ontologies is needed.

In this chapter we first describe a typical scenario with multiple ontologies (section 5.1). Then we address both the aforementioned problems. In section 5.2 we tackle the first problem by discussing means for reusing ontologies. Section 5.3 is dedicated to the second problem i.e. it considers different aspects of the evolution of multiple and distributed ontologies. Finally, the MeSH case study (see section 5.4) shows the benefits of the proposed approach.

5.1 Problem Description

To understand better the overall problem of managing multiple and distributed ontologies on the Web [70], we consider the following integration scenario through the whole section. Let's assume that the European Union (abbr. EU) wants to publish on-line all information about projects that it has funded, as well as information about participants in these projects. To enable other players in the market place (such as the Defence Advanced Research Projects Agency – DARPA, the education and science ministry of different countries (e.g. BMBF), research institutes etc.), to semantically process this information, it creates a basic ontology (*BO*) and bases its portal on it. The simplified version of this ontology is shown in Figure 37.

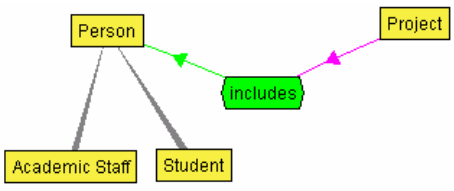


Figure 37. The basic ontology (*BO*) about projects and their participants

Let's assume that some institute specialises in industrial projects and also wants to publish its information on-line. To achieve that, it will create the project ontology (*PO*) (shown in Figure 38) for the description of its projects. In doing so, it should reuse as many definitions as possible from existing ontologies to speed up the engineering and to enable interoperability. However, it is not clear how to reuse the definition from *BO* within *PO*.

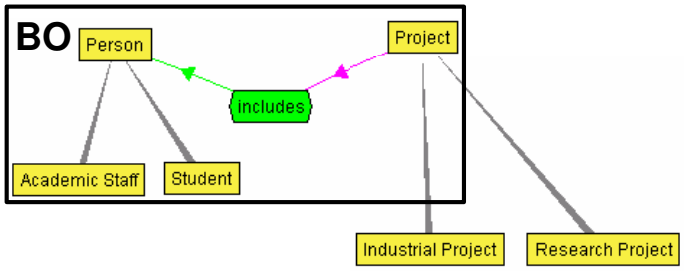


Figure 38. The project ontology (*PO*)

Assuming that this problem is solved, after reusing *BO* as the basis for *PO*, a further problem arises when *BO* needs to be adapted due to a change in business requirements. Thus, the fundamental question of how to evolve dependent ontologies[58] arises. This problem is worsened by the fact that ontologies are distributed in the Web.

To demonstrate the complexity of the problem, let's assume that the same institute also creates the staff ontology (SO) based on *BO*. This ontology models the information that is important from human resource management point of view. It is shown in Figure 39.

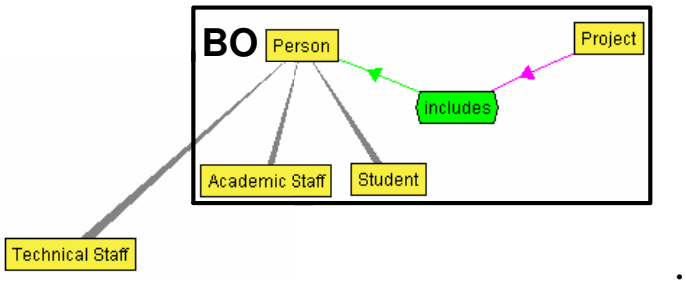


Figure 39. The staff ontology (*SO*)

Finally, in order to get an integrated view of one research institute it not sufficient to model only projects and staff. Rather, it is necessary to model other activities of employees such as lectures they offer, topics they research, etc. Further, the modelling of the organisation of an institute in the form of research groups can also be useful to discover possibility for synergy between groups or a gap between them. Therefore, it is necessary to combine all available ontologies into an integrated institute ontology (*IO*) based on both *PO* and *SO*. However, the dependencies now form a directed acyclic graph, which significantly complicates the synchronisation between dependent and distributed ontologies.

[58] A dependent ontology is an ontology that includes ontologies located at the same node on the network. A distributed ontology is an ontology that includes ontologies located at different nodes on the network.

From this scenario we derive two important ontology infrastructure components: first, means for reusing distributed ontologies are required. Second, we consider methods supporting the consistent evolution of distributed ontologies as crucial for the success of the distributed system in the long run. The rest of this section describes an integrated approach for managing multiple and distributed ontologies. The approach has been implemented within the KAON ontology management framework as described in chapter 7.

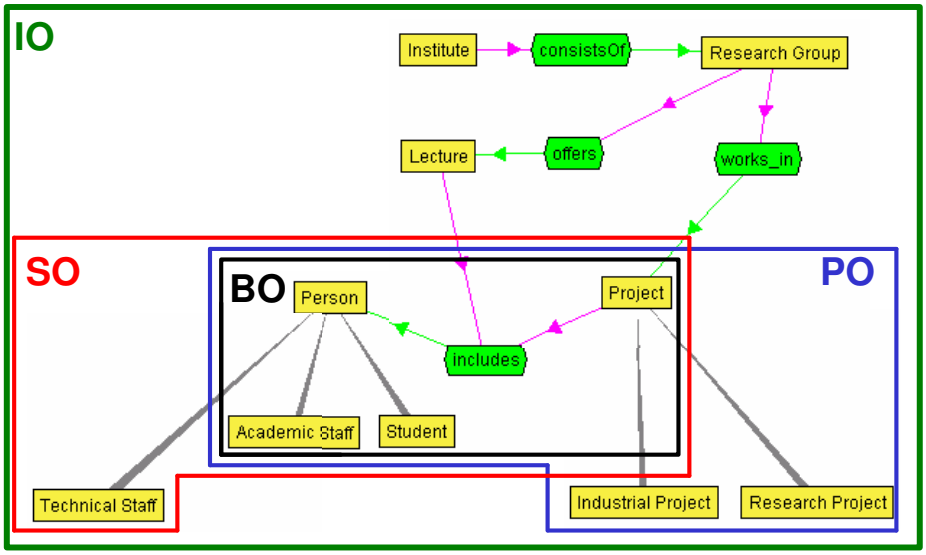


Figure 40. The institute ontology (IO)

5.2 Ontology Reuse

It is inefficient and error-prone to build ontology always from scratch. Rather, these models should be built by reusing smaller, well-defined components. An ontology can reuse concept and instance definitions from other ontologies through modularization. In this section we first define the ontology modularization. Then we discuss means for reusing existing ontologies while building new ontologies.

5.2.1 Modularization

Modularization is simply a process of reorganising software into modules. Modularization implies that the software is divided into separate parts that can be treated individually. Typical criteria used for the modularization process are cohesion and coupling [2]. Whereas cohesion tries to maximise the logical similarity of the entities that belong to the same module, coupling tries to establish the relationship between modules.

The same criteria have to be applied to the ontology modularization. As already mentioned in section 2.2, the KAON ontology language provides support for the modularization since one of the prerequisites for our ontology model was to support multiple, dependent ontologies. In this language, the request for the cohesion is interpreted as the self-containment of included ontologies. In order to facilitate the reuse of individual ontologies, we have to make sure that

the modules are self-contained. This results in the set of additional constraints that the modularization puts on the ontology model.

Definition 34 ***Self-Containment Constraints***: If *OI-model OIM* reuse some other *OI-model* OIM_1 (with elements denoted by subscript 1), that is, if $OIM_1 \in INC^{59}(OIM)$, then the following modularization constraints must be satisfied:

- $E_1 \subseteq E$, $C_1 \subseteq C$, $P_1 \subseteq P$, $R_1 \subseteq R$, $T_1 \subseteq T$, $INV_1 \subseteq INV$, $HC_1 \subseteq HC$, $HP_1 \subseteq HP$
- $\forall p \subseteq P_1\ domain_1(p) \subseteq domain(p)$
- $\forall p \subseteq P_1\ range_1(p) \subseteq range(p)$
- $\forall p \subseteq P_1,\ \forall c \subseteq C_1\ mincard_1(c, p) = mincard(c, p)$
- $\forall p \subseteq P_1,\ \forall c \subseteq C_1\ maxcard_1(c, p) = maxcard(c, p)$
- $I_1 \subseteq I$, $L_1 \subseteq L$
- $\forall c \subseteq C_1,\ instconc_1(c) \subseteq instconc(c)$
- $\forall p \subseteq P_1,\ i \subseteq I_1\ instprop_1(p, i) \subseteq instprop(p, i)$

From this definition it can be derived that (i) an including ontology *OIM* may only extend the included ontologies (e.g. OIM_1) and (ii) the entire including ontologies (e.g. OIM_1) are included. Therefore, it is not possible to include a part of an ontology.

The second request for modularization, which is related to the coupling, puts some constraints on the possibilities to realise the reuse between ontologies.

Definition 35 ***Multi-Ontology Constraint***: The ontology inclusion graph is a directed acyclic graph:

$\neg\exists\ OIM \in INC^*(OIM)$, where INC^* is the transitive closure of *INC*, and *OIM* is an *OI-model* (see Definition 2).

The multi-ontology constraint defines that the cyclical inclusions of ontology models are not allowed, which means that a graph whose nodes are OI-models and whose edges point from including to included models must not contain a cycle. The inclusion graph for the *IO* ontology depicted in Figure 40 is shown in Figure 41. The root node of the inclusion graph is the ontology *IO*, since it is not reused. The set of leaf nodes contains only the ontology *BO*, since only this ontology does not include other ontologies.

The role of the ontology modularization is to encourage ontology engineers to re-use and integrate already developed ontologies during the creation of their own conceptualisation. It allows the creation of a library of ontologies. The library should contain ontologies that are well separated and coherent with respect to their functionality. An ontology engineer can thus develop her own ontology by taking advantage of the predefined ontologies from the library without having to develop the underlying model manually. A library of reusable ontologies reduces the time and costs of developing an ontology. Moreover, it increases the quality of

[59] The function *INC* returns a set of included OI-models for a given OI-model. If $OIM_2 \in INC(OIM_1)$ then OIM_1 is including OI-model of the OI-model OIM_2 and OIM_2 is included OI-model of the OI-model OIM_1.

ontologies as well since general well-known ontologies, evaluated from many ontology engineers and applied in different applications, are used as foundation for building more specific ontologies. For example, the IEEE Standard Upper Ontology (SUO) Working Group[60] has invested tremendous effort, working with a large number of ontology engineers, to create standard top-level ontologies to enable various applications, such as data interoperability, information search and retrieval, automated inferencing, and natural language processing. Their ontology library system contains a group of classified ontologies, such as ontologies in SUO-KIF, formal ontologies and linguistic ontologies/lexicons. Among other ontologies, this library contains the Enterprise Ontology[61] that has content areas for organisation, agents, activity, time, etc.

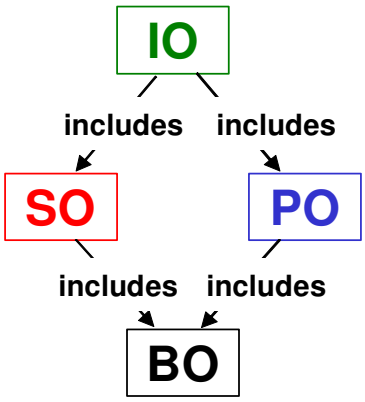


Figure 41. Inclusion relations between ontologies shown in Figure 40

Another example is the Foundational Ontology Library that has been developed within the European WonderWeb[62] project. The role of this library is to serve as a starting point for building new ontologies and to create a foundational framework for analyzing, harmonizing and integrating existing ontologies and metadata standards. The first module of this library is DOLCE (Descriptive Ontology for Linguistic and Cognitive Engineering) that aims at capturing the ontological categories underlying natural language and human commonsense. It is designed to be minimal in that it includes only the most reusable and widely applicable upper-level categories (e.g. it treats time as a fundamental category), rigorous in terms of axiomatization and extensively researched and documented.

5.2.2 Means for Ontology Reuse

In our approach we have identified two basic building blocks for realising reuse. First, ontology inclusion allows reusing ontologies available within the same node. Second, ontology replication enables inclusion in the case when ontologies are distributed on different ontology servers (nodes). These approaches are discussed below.

[60] http://suo.ieee.org/
[61] http://www.aiai.ed.ac.uk/project/enterprise/
[62] http://wonderweb.semanticweb.org

Ontology Inclusion

In the traditional software systems significant attention is devoted to keeping modules well separated and coherent with respect to functionality, thus making sure that changes in the system are localised to a handful of modules. Reuse is seen as the key method in reaching that goal. One of the main focuses of all existing reuse mechanisms is to eliminate completely the copy-and-paste reuse, which is seen as the prominent source of problems in software projects. The ontology-based systems in the Web are just a special class of software systems so that the same principles apply. If reuse is performed through duplication, problems arise when the reused ontology changes, as these changes must be applied to various multiple copies. Paraphrasing the open-closed reuse principle [84], each ontology should be a closed, consistent and a self-contained entity, but open to extensions in other ontologies.

These goals can be achieved by incorporating an explicit mechanism for including ontologies by reference to ontology languages and tools. Our approach is based on the KAON ontology language that is introduced in section 2.2.

Inclusion is supported by allowing an OI-model to include other OI-models, thus obtaining the union of the definitions from all included models. All definitions from included ontology are automatically available in the including ontology. Due to the multi-ontology constraint (see Definition 35), cyclical inclusions are not allowed because evolution of cyclically dependent OI-model would be too difficult (see section 5.3.1). Inclusion is performed by-reference. It means that models are virtually merged, which is formally defined as:

Definition 36 A dependent ontology *OIM* is an extension of the union of the definitions of all its included ontologies OIM_i, i.e.

$$OIM \supseteq \bigcup_i OIM_i,\ OIM_i \in INC(OIM), 1 \le i \le |INC(OIM)|.$$

The set of entities in a dependent ontology can be split into internally defined entities *INT(OIM)* and externally defined entities *EXT(OIM)*. It holds:

$$OIM = INT(OIM) \cup EXT(OIM),\ INT(OIM) \cap EXT(OIM) = \varnothing$$

where EXT(OIM) is a set of included entities and *INT(OIM)* is a set of entities defined in the ontology *OIM*. This separation enables representing the information about the origin of each entity explicitly. In practice it is achieved by introducing the following extensions of the ontology definition:

- each ontology has an identifier attached to it that identifies this ontology in a unique way;
- each ontology entity has its own identifier, which is a unique within this ontology (see the invariant IC_1*: Distinct Identity Invariant* defined in section 2.3).

Therefore, the name of an entity is written in the form:

ns:local_name

where ns is shorthand for a namespace prefix and represents the ontology identifier. The local_name is the entity identifier within the ontology with the identifier ns. By including this prefix to the entity name, accidental name clashes are easier to avoid. Indeed, even when two ontologies use the same identifier to define (either similar or different) things, the references

to the entities are guaranteed to be unambiguous. For example, it might happen that two ontologies contain the same concepts (e.g. "*Project*"). Due to the unique ontology identity, they would be considered as completely independent concepts. If the identifier of the first ontology is *ONTO1* and the second one *ONTO2*, then the including ontology would comprise two different concepts namely "*ONTO1:Project*" and "*ONTO2:Project*".

Currently we do not support resolving semantic heterogeneities between included models (e.g. establishing equivalencies between the "*Project*" and the "*das Projekt*" concepts).

Returning to the scenario from section 5.1, Figure 41 presents four example OI-models (*BO* -- basic ontology, *PO* -- project ontology, *SO* -- staff ontology and *IO* -- institute ontology). *PO* and *SO* each include *BO*, thus gaining an immediate access to all of its definitions. However, the information about the origin of ontology entities should be retained. Thus, the following distinctions exist:

- In *BO* and *SO* the concept "*Project*" concept does not have any sub- or superconcepts (excluding the root concept). However, in *PO* it has two subconcepts "*Research Project*" and "*Industrial Project*". These subconcepts are visible in *IO* as well.
- Similarly, only *SO* and (through the ontology inclusion) *IO* ontologies contain the "*Technical Staff*" concept.
- *IO* ontology has its own entities (e.g. concepts "*Lecture*", "*Institute*", "*Research Group*", properties "*offers*", "*works_in*", "*consistsOf*", etc.) that are not accessible for other ontologies.
- In *BO* the property "*includes*" has only the concept "*Project*" as domain concept whereas in *IO* it has an additional domain concept, i.e. the "*Lecture*" concept.
- Relationships between concepts also belong to appropriate OI-models. Hence, it is possible to establish that the subconcept relationship should be set up in IO, although the concepts themselves might be defined in separate included ontologies. This same strategy can be applied to any ontology primitive that relates two or more entities (e.g. property domains, property ranges, instantiation, property instances etc.).

A direct acyclic inclusion graph between OI-models is shown Figure 41. *BO* is indirectly included in *IO* twice (once through *PO* and once through *SO*). However, *IO* will contain all *BO* elements only once (in *IO* there will be only one "*Project*" concept). The possibility of including an ontology through multiple paths has significant consequences on the ontology evolution, as discussed in subsection 5.3.

This example demonstrates the open-closed principle. Each OI-model keeps track of its own information and is a consistent, self-contained and closed unit. On the other hand, each OI-model is open to reuse. In that case, any part of its structure can be extended as long as the original model itself is not changed.

Our approach is currently limited to including entire models rather than including subsets. Also, when a model is reused, information can only be added, and not retracted, and we currently do not deal with semantic inconsistencies between included ontologies. Although such advanced features may sometimes be useful, we deliberately limit our approach. By allowing inclusion of a part of a model it would be much more difficult to ensure the consistency of the including model since it is not clear which additional elements from the original model must be included. For example, if the "*includes*" property is not included in *IO*, it is not clear how to treat instances having this property instantiated. Further, changing ontologies becomes more complex because it is not clear how to propagate changes from the included ontology (e.g. *BO*) to the including one (e.g. *IO*).

Reusing Distributed Ontologies

Ontology inclusion allows reusing ontologies available within one node in the system. However, we envisage the Semantic Web where ontologies are spread across many different nodes (servers), so the inclusion mechanisms cannot be used directly. There are two possible solutions how to achieve reuse in this case: dynamic reuse and reuse through replication. Table 8 summarises advantages and disadvantages of both solutions.

The first solution is to make all ontologies accessible through an ontology server[63], which could integrate the information from included ontologies virtually (on-the-fly) by accessing the servers of these ontologies. Such solution has the benefit that all changes in the included ontologies are immediately visible in the including ontologies. While this desirable feature increases the consistency, it has several serious drawbacks:

- Servers are tightly coupled. Therefore, a failure of one system will cause failure of all servers that include the ontology;
- Standard top-level ontologies will be reused in many ontologies. Servers hosting them will therefore be overloaded because they will be often contacted by many other servers;
- Because answering every query requires distributed processing, the performance of the system using today's infrastructure would be unacceptable.

Table 8. Two approaches for reusing distributed ontologies

Dynamic Reuse	Reuse through Replication
Query the included server on demand	Included ontology copied once to the including server
Data always up-to-date	Data not always up-to-date (distributed evolution is needed)
Tight coupling (no server may fail)	Weak coupling – autonomy of ontologies preserved
Huge communication overhead	Low communication overhead
Complex approach - the performance would be unacceptable	A more practical solution

Therefore, a more practical solution to the problem in the WWW context is to replicate distributed ontologies locally and to include them in other ontologies. Replication eliminates above-mentioned problems but introduces significant evolution and consistency problems, which we further discuss in section 5.3.2. The most important constraint is that replicated ontologies should never be modified directly. Instead, the ontology should be modified at the source and changes should be propagated to replicas using the distributed evolution process.

Figure 42 summarises the differences in realising ontology reuse in the centralised and decentralised systems. When ontologies are within the same node (server), the reuse is done by the reference. When ontologies are distributed across the Web, the reuse is achieved through the replication. In this way, the trade-off between performance and maintainability is significantly shortened. The left part of Figure 42 corresponds to the centralised systems. The

[63] The task of an ontology server is to store and maintain ontologies and to provide data interchange facilities.

right part of Figure 42 shows the distributed system with two nodes. The first node contains only the ontology O_1. The second node contains the replica of the ontology O_1, i.e. the ontology O_1', and the ontology O_2 that includes this replica.

It is very important that the ontology replication should not be understood as simple copying of the original ontology. If this operation is performed in an ad-hoc way, evolving replicated ontologies will be impossible. Distributed evolution process requires associating meta-data with each ontology, which must be maintained during the replication process. This is described in further detail in subsection 5.3.2. In order to replicate an ontology, it must be physically accessed. Ontologies on the Web are typically known under a well-known identifier, i.e. URI, which can be used to access the ontology through appropriate protocol (e.g. HTTP). However, this introduces problems when the ontology is replicated since the URI used to access the ontology and the URI under which the ontology is originally known become different. To handle this consistently, we establish two different URIs for each ontology:

- The logical URI is unique for each ontology and is always the same, regardless of the ontology's location. The uniqueness of the URI is typically achieved by incorporating into it the Internet name of the organization that created the ontology;
- The physical URI unambiguously identifies the location of the ontology and contains all information necessary to access the ontology, such as the protocol to be used or relevant connection parameters.

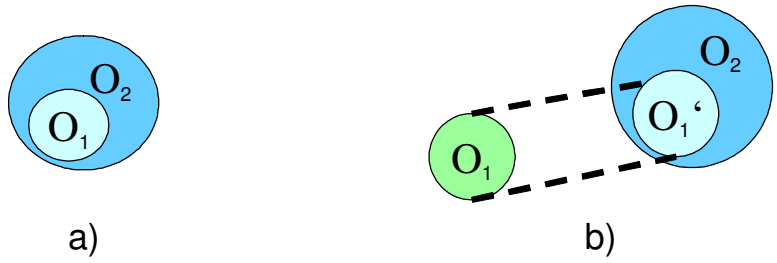


Figure 42. Two ways for realising ontology reuse: (a) ontologies are within the same ontology server; (b) ontologies are distributed across the Web or different servers in companies

For example, the *BO* from our example may have the logical URI http://europa.eu.int/bo. No other ontology with that URI exists anywhere in the world. However, the ontology may be replicated to the file system, and the physical URI will be file:/c:/bo.kaon. If the ontology is stored in the database, then its physical URI may be jboss://wim.fzi.de:1099?http://europa.eu.int/bo.

Therefore, each ontology has to two different URIs attached to it. We define the following functions to distinguish them:

- the function *logicalURI(OIM)* that gives the logical URI of a given ontology *OIM*;
- the function *physicalURI(OIM)* that returns the physical URI of a given ontology *OIM*.

To locate an ontology by the logical URI, this URI must be resolved to a physical URI. This can be done through the ontology registry described in [71]. After replication, the ontology

registry is not needed any more, so the registry does not represent a single point of failure of the proposed system.

5.3 Evolution of Multiple Ontologies

As we have already defined, the ontology evolution can be defined as the timely adaptation of an ontology as well as the consistent propagation of changes to the dependent artefacts (e.g. ontology instances on the Web, dependent ontologies and application programs using the changed ontology). In this section we investigate the effects that changes in an ontology might have for dependent ontologies. Indeed, we extend the single ontology evolution approach (described in chapter 4) to take into account the multiple ontologies.

It is worth mentioning that there is no mature tool support for tracking changes in ontologies and thereby for controlling consistency and effects on dependent ontologies. Human investigation is by now the only way to control evolution between dependent ontologies correctly.

To automate this process, we started by identifying two dimensions of the overall ontology evolution problem. This is summarized in Figure 43. The first dimension defines the number of the ontologies being evolved whereas the second dimension specifies the physical location of evolved ontologies. Since it is not possible to fragment [102] one ontology across many nodes, ontology evolution can be discussed at three levels. For evolution of single ontologies the essential phase is the semantics of change phase whose task is to maintain ontology consistency. This was elaborated in chapter 4. Here we focus on maintenance of multiple, dependent ontologies. In subsection 5.3.1 we extend the change propagation and capture phases of the ontology evolution process (see chapter 3) to cover the evolution of multiple dependent ontologies within a single node. Finally, in subsection 5.3.2 we extend the change capturing and change implementation phases of the dependent evolution process to support evolution of distributed ontologies.

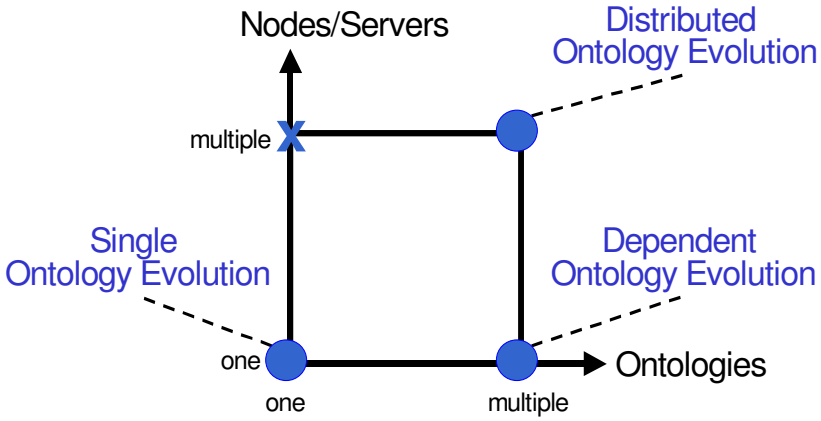


Figure 43. Levels of Ontology Evolution Problem

5.3.1 Dependent Ontology Evolution

In this subsection we extend the single ontology evolution approach to take into account the inclusion relationships between ontologies within one node. An ontology that depends on

(includes) ontologies residing at the same node on the network is called the *dependent ontology*. As the included ontology is changed, the consistency of the dependent ontology may be invalidated.

Returning to the example of Figure 40, let's assume that the "*Project*" concept in the *BO* ontology should be deleted. Further, let's consider only the *BO* ontology independently of the ontologies that reuse it. To prevent inconsistencies, before deleting it, the "*Project*" concept must be removed from the domain of the "*includes*" property. Since the ontology consistency definition might specify[64] that properties without domain concepts are not allowed, the property must be deleted as well. To do that, the "*Person*" concept must be removed from the range. Finally, the "*Project*" concept has to be detached from the root concept in the concept hierarchy. The complete list of necessary changes obtained in the semantics of change phase of the ontology evolution process is presented in Figure 44.

However, if the "*Project*" concept from *BO* is deleted, the ontology *PO*, and through transitivity of inclusion the *IO* as well, will be inconsistent. The "*Research Project*" and "*Industrial Project*" concepts will have a parent concept that is not defined. Moreover, the property for the domain concept "*Lecture*" does not exist anymore. The same holds for the range concept for the "*works_in*" property.

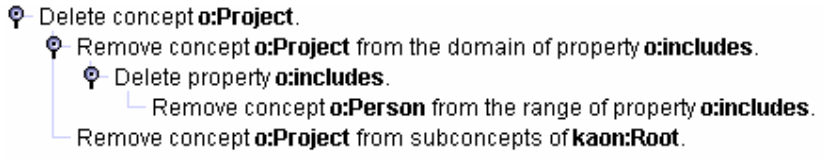


Figure 44. Generated Changes in *BO*

In the previous example, a request for a change is applied to the ontology (i.e. *BO*) where the entity taken into consideration (the concept "*Person*") is defined. Moreover, the additional changes are generated by considering the *BO* ontology in isolation without any context in which it is reused.

Let's now consider the application of the same change (i.e. the deletion of the "*Project*" concept) in an ontology that reuses *BO*, i.e. *IO*. It is important to notice that applying this change to *IO* only is not sufficient. In *IO* the "*includes*" property has two domain concepts, so removing one of them will not trigger the removal of the property. Therefore, if *BO* is considered independently, it is inconsistent since the "*includes*" property will not have a domain concept in this ontology. The list of changes generated in this case[65] is shown in Figure 45, which is quite different from the changes shown in Figure 44.

The previous example shows that maintaining consistency of a single ontology is not sufficient; dependency between ontologies must be taken into account as well. We define the consistency of the dependent ontology in the following way:

[64] It depends on whether the user-defined constraint UC1: Domain/Range Property User-defined Constraint (see section 2.3) is an element of the ontology consistency definition or not. It is up to ontology engineers to specify which soft- and user-defined constraints must be fulfilled. However, invariants are hard-constraints and they must not be invalidated.

[65] The list of changes depends on the selected evolution strategy. In this case the evolution strategy contains the elementary evolution strategy EES1 for the resolution point RP1 and the elementary evolution strategy EES17 for the resolution point RP7.

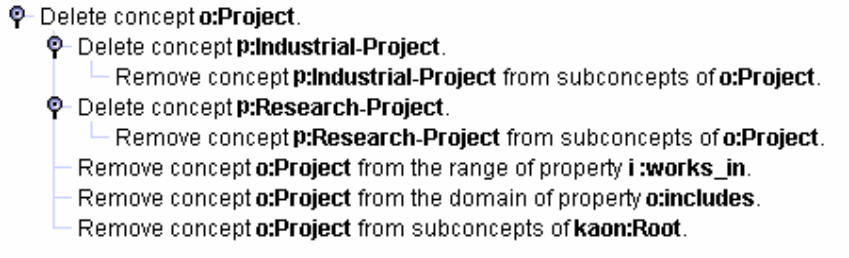


Figure 45. Generated Changes in *IO*

Definition 37 ***Dependent Ontology Consistency*** - A dependent ontology *OIM* is consistent if the ontology itself and all its included ontologies ($\forall i\ OIM_i \in INC(OIM)$), observed alone and independently of the ontologies in which they are reused, are single ontology consistent. Formally, it is defined as:

$$dependentConsistency(OIM) \leftrightarrow$$

$$consistency^{66}(OIM) \wedge \forall OIM_i \in INC(OIM)\ dependentConsistency(OIM_i)$$

The synchronisation between dependent ontologies can be achieved by propagating changes from the changed ontology to all ontologies that include it. There are two ways of doing that [7]:

- *Push-based approach*: Changes from the changed ontology are propagated to dependent ontologies as they happen;
- *Pull-based approach*: Changes from the changed ontology are propagated to dependent ontologies only at their explicit request.

The pull-based approach is better suited for less stringent consistency requirements. Using this approach the dependent ontologies may be temporarily inconsistent. This makes recovering of the consistency of dependent ontologies difficult as the information about the original state of the changed ontology is lost. For example, when the concept "*Project*" is deleted, its position in the concept hierarchy is lost and is not available for resolution of inconsistencies of the "*Industrial Project*" concept in *PO*.

The push-based approach is suitable when strict dependent ontology consistency is required since the information about the original state of the changed ontology is available for the evolution of the dependent ontology. For example, the removal of the concept "*Project*", among other additional changes, requires previous resolution of the consistency of the "*Industrial Project*" concept in *PO* as shown in Figure 45. We choose to take this approach since the permanent consistency of ontologies within one node is of paramount importance.

By adopting the push-based approach, there are three different strategies for choosing the moment when changes are propagated [105]. Using the periodic delivery, changes are propagated at regular intervals. Using ad-hoc delivery, changes are not propagated according to a previously defined plan. Both of these strategies are unacceptable for dependent ontology evolution since they cause temporary inconsistencies of dependent ontologies. Therefore, we propagate changes immediately, as they occur.

[66] This function is defined in chapter 4.

We incorporate the push-based approach by extending the change propagation and change capturing phases of the single ontology evolution process as shown in Figure 46. The role of the *Ontology Propagation Order component* is to determine to which dependent ontologies the changes should be propagated and in which order this should be done. The role of the *Change Filtering component* is to determine which changes must be propagated to which ontologies. The *Change Ordering component* determines the order in which changes must be handled by each ontology.

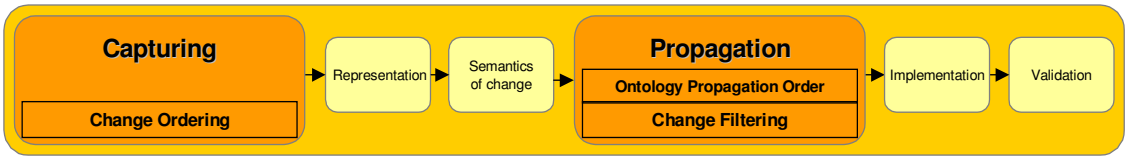


Figure 46. Dependent Ontology Evolution Process

Ontology Propagation Order

When propagating changes to dependent ontologies on a single node, the following three aspects relating to the ontology propagation order must be considered:

- As changes occur in an ontology, they must be pushed to all ontologies that either directly or indirectly (through other ontologies) include the changed ontology;
- In order to propagate a change to an ontology, the change must previously be processed by all ontologies on the path between the source and target ontology. Therefore, all the ontologies on a single node are topologically[67] sorted according to their inclusion relationship. The topological order organises the dependent ontologies in such a way that for each O_1 and O_2, if O_1 includes O_2 directly or indirectly, then O_2 occurs before O_1 in a linear ordering;
- Since all ontologies at a node are topologically ordered, when changes are propagated to dependent ontologies, only those ontologies that include the changed ontology and that follow the changed ontology in the topological order must be visited. Note that if cyclical inclusions of OI-models were allowed, the propagation order would contain cycles and would be extremely hard to manage.

Returning to the example shown in Figure 40, changes in *BO* must be propagated to *PO, SO* and *IO* (since *IO* includes *BO* indirectly though *PO* and *SO*). Further, several different topological orders may exist (e.g. *BO*, *PO*, *SO*, *IO* or *BO*, *SO*, *PO*, *IO*) since some ontologies are independent on each other (e.g. *PO* and *SO*). The propagation of changes must be performed in either one of these orders. On the other hand, assuming the first topological order (*BO*, *PO*, *SO* and *IO*), that is shown in Figure 47, a change in *PO* is propagated only to *IO*. Although *SO* is after *PO* in the topological sort, it does not include *PO* so it does not receive *PO*'s changes.

[67] The topological order of a directed graph is an ordering of graph's nodes where each node occurs after all of its predecessors.

Change Filtering

As a change from the source ontology *S* is propagated to a dependent ontology *D*, in order to maintain the consistency of *D*, additional changes will be generated as explained in chapter 4. These changes must also be propagated further up the ontology inclusion topological order. However, only induced changes should be forwarded. If original changes were propagated as well, then ontologies that include *D* would receive the same change multiple times: directly from *S* and indirectly from all ontologies on any inclusion path between *D* and *S*. This would result in an invalid ontology evolution process since the same change cannot be processed twice. In order to prevent that, propagated changes are filtered.

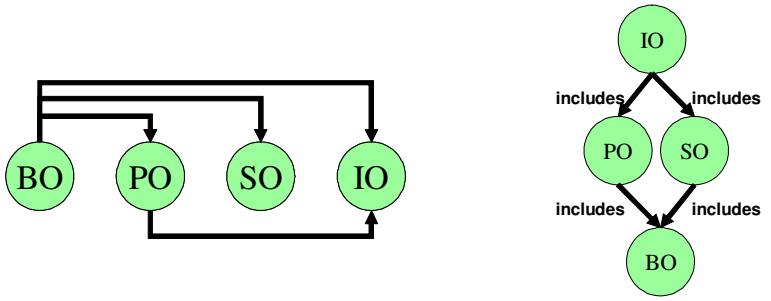


Figure 47. Change propagation order for the ontologies shown in Figure 40

As shown in Figure 45, deletion of the "*Project*" concept in *BO* is propagated to *PO* resulting in a set of new changes such as the removal of the "*Industrial Project*" concept as the subconcept of the "*Project*" concept. That change is propagated to *IO*. Removal of the concept "*Project*" is propagated to *IO* from *BO* directly and must not be propagated from *PO*. This is shown in Figure 48. Notice that change filtering is not done for the sake of performance: if the request for the deletion of the "*Project*" concept were propagated to *IO* from *PO* as well, then *IO* would receive the same change twice, and the second change would fail since the concept has already been deleted.

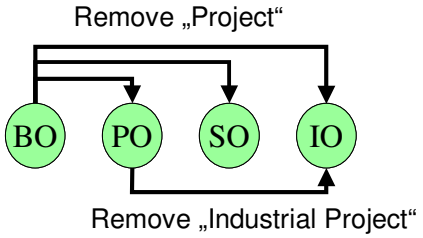


Figure 48. Change filtering for the ontologies shown in Figure 40

Change Ordering

The order of processing changes in each ontology is important. Let's assume that *S* is the ontology being changed, *I* is some ontology that directly includes *S* and *D* is some ontology that directly includes *I*. It is important that *D* processes changes generated by *I* before changes

generated by *S*. Otherwise, if *D* receives changes from *S* before changes from *I*, *S*'s changes will generate additional changes in *D* that include those that will later be received from *I*. This in turn will also lead to processing the same change twice. This approach is recursively applied when *D* and *S* are connected with paths of length greater than two.

Returning to the example shown in Figure 40, *IO* should process the removal of the "*Project*" concept after processing the removal of the subconcept "*Industrial Project*" from *PO*. This is shown in Figure 49. If this were not the case, processing removal of "*Project*" in *IO* would generate removal of the subconcept "*Industrial Project*" in *IO*, which will then be later received from *PO*.

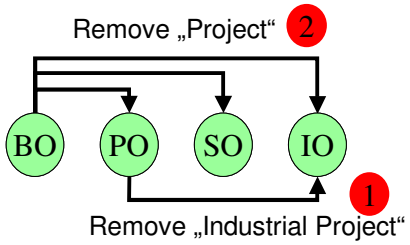


Figure 49. Change ordering for the ontologies shown in Figure 40

Algorithm for the Dependent Ontology Evolution

The algorithm for the evolution between dependent ontologies within one node is presented in Figure 50. It processes all changes that are requested by the user through the procedure ***processChange*** (cf. 2--4). This procedure resolves a change by generating the additional changes needed to keep the consistency of the ontology ***o*** for which the method was called (cf. 7--9). Only changes generated in ***o*** are propagated (cf. 11) to the all ontologies including ***o*** according to the topological order of all the ontologies within the node (cf. 13--14). The recursive call (cf. 16) of the ***processChange*** procedure for the filtered change and topological order of dependent ontologies guarantees that the receiving ontologies will process the changes from the directly included ontologies before changes from the indirectly included ontologies. Finally, the change is applied to the ontology ***o*** (cf. 21).

The complexity of this algorithm depends on the number of the ontologies that follow the ontology that is changed in a topological sort. It increases in a linear way, i.e. $O(N)$, where N is the number of all ontologies including the ontology that is changed. The time complexity of the creation of the topological sort based on the inclusion graph is $O(V + E)$, where V is the number of nodes (i.e. ontologies) in the inclusion graph and E is the number of inclusion relationships between these ontologies. However, although it could influence the performance of the dependent evolution algorithm, it is not critical for the change propagation since it is not done every time a change is propagated. Rather, the topological sort has to be updated only when an ontology is needed to be included or excluded from any other ontology. Further, this can be sped up by attaching a separate topological sort for each ontology.

As can be noticed, our algorithm does not propagate changes in a layer-by-layer manner. Instead of propagating changes only to the ontologies that directly include the ontology that is changed, the changes are propagated to all dependent ontologies by considering the topological graph. The main reason is that the inclusion graph might be of arbitrary complexity as shown in Figure 51:

- one ontology (O_2) may be included directly and indirectly (through O_5) in the same dependent ontology (O_6);
- it may be several different paths between two dependent ontologies. For example, there are two paths between ontologies O_1 and O_6;
- diverse paths between the same ontologies may be of different length, where the length is determined by the number of the intermediate ontologies. For example, one path between ontologies O_1 and O_6 contains, besides these two ontologies, the ontology O_5 as well. The second path is of the length of 4 since it includes the ontologies O_3 and O_4.

Algorithm Dependent Ontology Evolution Algorithm

```
EVOLVEONTOLOGIES(ℒ𝒞, o)
Require: ℒ𝒞 - list of changes, o - ontology being changed
 1: 𝒯𝒮 = topological sort of ontologies at the node
 2: for all c ∈ ℒ𝒞 do
 3:     PROCESSCHANGE(c, o)
 4: end for

PROCESSCHANGE(c, o)
Require: c - change to process, o - ontology being changed
 5: es = evolution strategy for o
 6: /*Semantics of Change*/
 7: while generated change gc by es for c in o do
 8:     processChange(gc, o)
 9: end while
10: /*Change Filtering*/
11: if c is generated in o then
12:     /*Ontology Propagation Order*/
13:     for all ontology d after o in 𝒯𝒮 do
14:         if ontology d includes o then
15:             /*Change Ordering*/
16:             processChange(c, d)
17:         end if
18:     end for
19: end if
20: /*Change Implementation*/
21: change ontology o according to c
```

Figure 50. Dependent Ontology Evolution Algorithm

There are two additional constraints: (i) each change can be applied once and only once (see change filtering activity); (ii) each change has to be performed after all its consequences (see change order activity). If the changes are broadcasted only to direct parents, it would be impossible to control the order of performing changes since it is never known whether all the changes are received or not due to differences in the path lengths.

In order to prevent these problems, all dependent ontologies receive the change immediately. However, the recursive realisation of the dependent ontology evolution algorithm guaranties that induced changes are propagated, and therefore processed, firstly (cf. 8 in Figure 50). Moreover, the recursion assures the ontologies more close to the ontology broadcasting a change will process this change before remoted ontologies (cf. 16 in Figure 50). For example, assuming the topological order O_1, O_2, O_3, O_4, O_5, O_6 for the inclusion graph shown in Figure 51, O_3 will receive changes from O_1 before these changes will be received by ontologies O_4

and O_6 that include O_3. Note that closeness between two ontologies O_i and O_j is measured in the number of nodes (i.e. ontologies) between them, where a node is considered only if it includes the ontology O_i and is included in the ontology O_j.

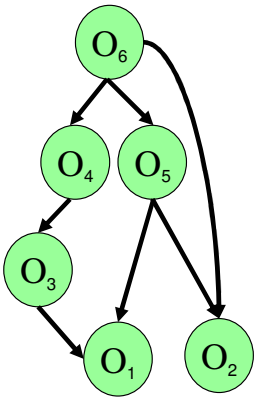


Figure 51. A very complex inclusion graph

Discussion

It is not always possible to propagate changes. For example, in a dependent ontology O_2 an included ontology O_1 might be extended with the subconcept relationship between the concepts "*X*" and "*Y*" defined in the ontology O_1. If O_1 is changed with the subconcept relationship in other direction (the concept "*Y*" is a subconcept of the concept "*X*"), then the propagation of this change from O_1 to O_2 would provoke a cycle in the concept hierarchy. This example shows that the change propagation phase of the ontology evolution process cannot be fully automated since some changes would provoke inconsistencies that cannot be resolved.

Therefore, the change propagation is a partially automated activity [100]. However, the ontology evolution system has to provide support for avoiding the problems such as previously mentioned cycle in a concept hierarchy. For example, an ontology engineer should be able to accept or to reject changes. There are several options [138] which clarify the resolution:

- break the dependency between ontologies O_1 and O_2, i.e. the ontology O_2 is an autonomous ontology which is extended with the "copy" of all entities from the ontology O_1;
- try to compensate the requested change (e.g. *AddSubConcept("Y","X")*) by generating the inverse change (e.g. *RemoveSubConcept("X","Y")*) in the dependent ontologies before applying the requested change;
- make a replica of the ontology O_1 and include this replica into the ontology O_2 instead of the original. In this way the dependent ontology O_2 stays compliant with the last version of the modified ontology.

Note that there is no problem with the usage of the same identifier for the entities in different ontologies due to different URIs of the ontologies they belong to.

5.3.2 Distributed Ontology Evolution

A distributed dependent ontology is an ontology that depends on an ontology residing at a different node on the network. The physical distribution of ontologies is very important since it creates additional problems that are not encountered when the ontologies are collocated. This additional complexity stems from the fact that reusing distributed ontologies is achieved through replication (see section 5.2.2). Since the original ontology is updated autonomously and independently of replicas, this in turn introduces an additional type of ontology consistency, so-called the replication ontology consistency.

To explain the notion of replication ontology consistency we assume a distributed system of replicated ontologies as shown in Figure 52. Service provider A uses its ontology server to develop a basic ontology (*BO*). Service provider B may find this ontology appropriate to be used as a foundation for *PO*. To reuse *BO*, B replicates it from the A's ontology server into its ontology server (cf. *BO'*). Replication is done to decouple ontology servers. To develop *PO*, B uses the modularization facilities of its ontology server, making all definitions from the *BO'* available in *PO*. Similarly, for the creation of the *SO*, the server provider C creates its own copy of the *BO* ontology (cf. *BO''*). Finally, in order to reuse the *BO* and *SO* ontologies, the service provider D creates the copies of these ontologies (cf. *PO'* and *SO'*). Even though the ontologies *PO'* and *SO'* contain its own replicas of the original *BO* (i.e. *BO'* and *BO''*) the ontology *IO* contains only one copy of *BO* (i.e. *BO'''*), which implies that the replicas *BO'* and *BO''* have to be equivalent.

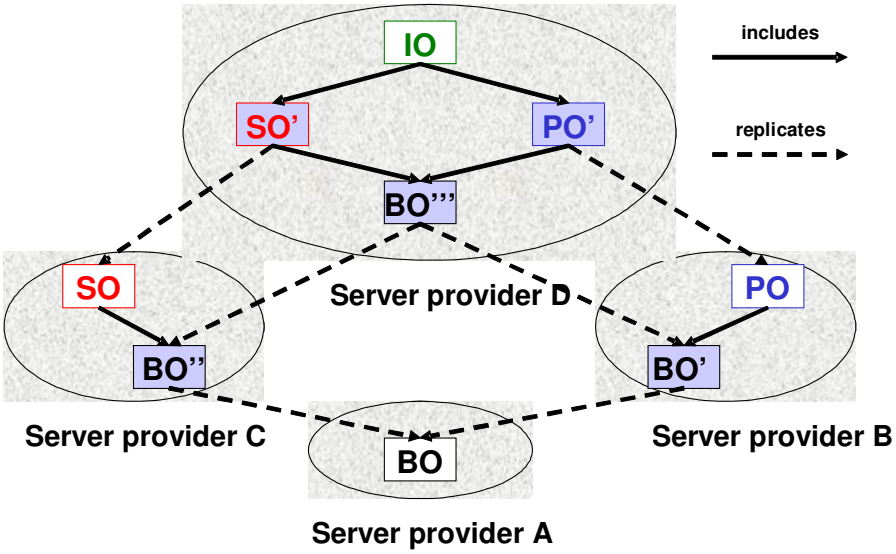


Figure 52. Dependencies between ontologies shown in Figure 40 in a distributed scenario

As the business requirements change, A will change the *BO*. For example, A may refine the hierarchy of the concept "*Project*" by distinguishing EU and national projects. At this stage the entire system is inconsistent – B(C) has not updated yet *PO* (*SO*) or its local copy of *BO*. In the WWW context such situations simply cannot be avoided, due to numerous factors, such as the number, geographical distribution and technological heterogeneity of subjects involved, independence of their business cycles, etc. Therefore, we must simply accept the fact that the service provider B(C) works with an outdated version of the ontology - a request directed to B(C) must be formulated using concepts from the version of *BO* that B(C) currently

understands or the system will produce incorrect results. Although inconsistencies are inevitable, the infrastructure should provide means for their easier detection and management. In our case this means that B(C) may check for newer ontology versions at leisure. When ready, B (C) needs to determine which included ontologies have been updated and to apply the distributed ontology evolution algorithms to bring both the *PO* (*SO*) and the local copy of *BO* up-to-date.

Therefore, after a change in the ontology *BO*, the ontology *BO'* (*BO''*) at service provider B(C) is replication inconsistent if it has not been updated according to changes in its original at the service provider A. This implies the replication inconsistency of *PO* (*SO*) at service provider B(C) (since *PO* (*SO*) includes *BO'* (*BO''*) which is replication inconsistent). Finally, this implies the replication inconsistency of *IO* at the service provider C in the same way.

We already introduced the idea of the replication ontology consistency, here we define it formally. Further, we define formally the additional consistency requirements arising from the distributed scenario.

Definition 38 ***Replication Ontology Consistency*** - An ontology is replication consistent if it is equivalent to its original and all its (directly and indirectly) included ontologies are replication consistent. Formally, it is defined as:

$$replicationConsistency(OIM) \leftrightarrow$$
$$equivalent^{68}(OIM, original^{69}(OIM)) \wedge$$
$$\forall OIM_i\ OIM_i \in INC(OIM)\ replicationConsistency(OIM_i)$$

The further complexity related to the evolution of the distributed, dependent ontologies lies in the fact that an ontology O_1 might include the same ontology O_2 through the several different ontologies O_i, where each of these ontologies O_i has its own replica of the original ontology O_2. These replicas might reflect the different state of the original ontology O_2. However, in order to achieve the compatibility and to make ontology evolution feasible, these replicas must be equivalent.

The possibility of including the replica of the same ontology through multiple paths has significant consequences on the ontology evolution. We define the version[70] of an ontology, which means that each ontology has a version number attached to it. It is incremented each time that an ontology is changed. Thus, checking the equivalence of the replica and the original can be done by simple comparison of these numbers. Therefore, the equivalency between two arbitrary ontologies is defined as:

$$equivalent\ (OIM1, OIM2) \leftrightarrow$$
$$version(OIM1) = version(OIM2) \wedge$$
$$logicalURI(OIM1) = logicalURI(OIM2)$$

[68] The function *equivalent*(O_1,O_2) returns true if all entities of the first ontology O_1 are contained in the second ontology O_2 and vice versa. Otherwise it returns false.

[69] The function *original(O)* returns the original ontology for a given ontology *O*. If the given ontology *O* is the original, then this ontology is retrieved. Note that this function is performed by mapping the logical URI of an ontology into its physical URI (see section 5.2.2) since an ontology and all its replicas must have the same logical URI whereas the physical URI depends on their location.

[70] The function *version(O)* gives a version number of a given ontology *O*.

where the function *logicalURI* is defined in section 5.2.2.

To avoid problem that included ontologies might contain different versions of the same original, we define the additional ontology consistency constraint:

Definition 39 ***Replication Constraint*** - An ontology *OIM* must contain only one version of each included ontology.

$$\forall OIM_1 \in INC^*(OIM) \quad \forall OIM_2 \in INC^*(OIM) \setminus \{OIM_1\}$$
$$logicalURI(OIM_1) = logicalURI(OIM_2) \rightarrow version(OIM_1) = version(OIM_2)$$

The previous constraint does not prevent the same ontology to be included several times. The efficient management of changes in such a system (in terms of space and time) requires the further condition:

Definition 40 ***Uniqueness Condition*** - An ontology *OIM* must contain one and only one copy of each included ontologies.

$$\forall OIM_1 \in INC(OIM) \quad \neg\exists OIM_2 \in INC^*(OIM) \setminus \{OIM_1\}$$
$$logicalURI(OIM_1) = logicalURI(OIM_2)$$

Definition 39 and Definition 40 are not contradictory. The first one is used as a precondition for the "*AddOI-model*" change to avoid dealing with different versions of the same ontology in one model. It considers the whole inclusion graph between ontologies. The second condition requires that included ontologies are virtually merged. It means that the distributed, dependent ontology has to include only one replica of any ontology independently of the number of paths between these ontologies. This condition can be satisfied only if the first one is fulfilled. Further, it is more related to the practical realisation of an ontology evolution system. Therefore, it is not an ontology consistency constraint since for the consistency only the "equivalence" between the replicas of an ontology is important (and not the number of these replicas). Rather, this condition assures an efficient management of changes in the system.

The previously mentioned problems do not exist in a centralised environment (see section 5.2.2) since the reuse is achieved through inclusion. This means that the original entities from the included ontology are referenced instead of their replicated entities.

In a distributed environment the problem is even made worse by the fact that a distributed, dependent ontology can extend its replicas independently of the original. For the distributed, dependent ontologies there is a trade-off between having autonomy over the extension of the replica and conforming to the original to enjoy benefits of interoperation (since the original can be considered as a shared ontology) [100]. If the distributed ontology is motivated to conform, then the burden lies on the distributed ontology which must manage its own changes and incorporate the changes of the included ontology at periodic intervals or at an explicit request. This proceeding is called the distributed ontology evolution. It includes the resolution of two problems:

1. replication ontology consistency (see Definition 38), i.e. the propagation of changes from an original to its replica;
2. dependent ontology consistency (see Definition 37), the propagation of changes from a replica to an ontology that includes it.

To resolve replication inconsistencies between ontologies, first a way of synchronising distributed ontologies is needed. Table 9 discusses the pros and cons of two well-known approaches [7] for synchronising distributed systems. Although seemingly similar, there is a significant difference in the approaches described in subsection 5.3.1 since we are here dealing with a distributed system.

Table 9. Push vs. pull synchronisation of ontologies

	Push	**Pull**
Dependency Information	centralised	local
Complexity of Management	high	medium
Type of Consistency	strict	loose
Communication Overhead	high	optimised

Under push synchronisation the changes of originals are propagated to ontologies including replicas immediately. We identify several drawbacks of using this approach for realistic scenarios on the Web. First, to propagate changes, the information about ontologies that reuse each ontology should be available. Thus, an additional centralised component managing inclusion dependencies between ontologies is needed. Second, with the increase in the number of ontologies and of subjects reusing them, the number of dependencies will grow dramatically. Managing them centrally will be too expensive and impractical as the problem of evolving dependencies is raised. Third, forcing all ontologies to be "strictly" consistent at all times reduces the possibility to express diversities in a huge space such as the Web. Subjects on the Web may not be ready to update their dependent ontologies immediately and may opt to keep the older version deliberately. Finally, the changes are propagated one-by-one, thus introducing significant communication overhead. Grouping changes and sending them on demand will perform better.

Therefore, in the distributed environment we advocate the use of the pull synchronisation. Under this approach information about included ontologies is stored in the dependent ontology, thus eliminating the need for central dependency management. Original ontologies are checked periodically to detect changes and collect deltas. During this process, it may be possible to analyse changes and to reject them if they do not match the current needs. Thus, we propose a "loosely" consistent system since replication consistencies are enforced at request. Permitting temporary inconsistencies is a common method of increasing performance in distributed systems [102]. Hence, we use the pull approach for synchronising originals and replicas whereas we use the push approach for maintaining consistency of ontologies within one node. Thus, our solution employs a hybrid synchronisation strategy combining their favourable features while avoiding their disadvantages.

Regardless of the synchronisation approach, a question about how to resolve replication inconsistencies remains open. We note that replication inconsistencies cannot be resolved by simply replacing the replica with the new version of the original. This will cause inconsistencies of the dependent ontologies, as discussed in subsection 5.3.1. Instead, replication and dependency inconsistency must be resolved together in one step.

Definition 41 ***Distributed Ontology Consistency*** – A distributed, dependent ontology *OIM* is a consistent ontology if it is dependent ontology consistent and replication ontology consistent.

$$distributedConsistency(OIM) \leftrightarrow$$
$$replicationConsistency(OIM) \wedge dependentConsistency(OIM)$$

The replication ontology consistency ensures that only up-to-date replicas are included. On the other hand, the dependent ontology consistency guaranties that each ontology, considered independently of either ontologies that it includes or ontologies that are included in it, satisfies the set of ontology constraints (see Definition 6). Therefore, an ontology is distributed consistent if it included the latest, consistent version of included ontologies as well as it is a consistent dependent ontology.

The distributed ontology consistency can be achieved by applying dependent evolution algorithms on deltas. Deltas are changes that have been applied to the original since the last synchronisation of the replica. By using the pull synchronisation strategy and by applying the dependent evolution process (see Figure 46) to deltas, we derive the distributed ontology evolution process through three extensions. This process, shown in Figure 53, is responsible for propagating changes from originals to replicas. We extend the implementation phase by introducing the *evolution log* for keeping information about performed changes. Further, we extend the change capturing phase by three components. During *identification of changed originals* we identify which original ontologies have changed. In *extraction of deltas* we identify the changes performed at the original and not at the replica by reading the evolution log. Finally, during *merging of deltas* we generate a cumulative list of changes that must be performed at the replica.

Logging Changes

In order to resolve replication inconsistencies, two known ways of identifying deltas between originals and replicas are known [102]:

1. the full content of the original ontology may be compared to the replica;
2. the history of changes to the original may be kept explicit.

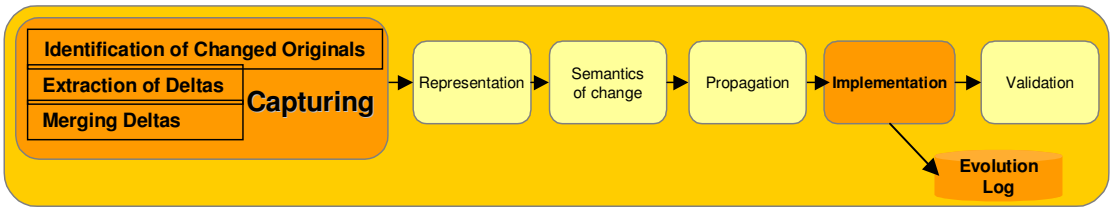


Figure 53. Distributed Ontology Evolution Process

The first solution requires extracting changes from differences between the original and the replica, which is a complicated and time-consuming process. An algorithm for comparing two versions of ontologies is given in [65]. It is based on a set of heuristics that take into account syntactical differences. However, for the evolution of dependent ontologies it is not sufficient only to detect the differences between original and its replica. It is essential to discover the

order in which these changes are applied to the original since different order of changes might have completely different consequences on the ontologies that include the replicas.

Further, to compare ontologies the current version of the original must be copied temporarily to the replica's node. This may incur unnecessary communication overhead. For example, the WordNet ontology described in [71] is very large. If a concept is added to it, it is better to transfer only the information about this addition, instead of transferring the whole ontology.

To avoid these drawbacks, we follow the second option. For each distributed ontology, a specialisation of the evolution ontology (see section 3.2.4) is created. This is the so-called evolution log[71] that tracks the history of changes to the ontology. The utility of these formal models (i.e. the evolution ontology and the evolution log) is demonstrated by their usage in the distributed synchronisation. They enable an ontology engineer to synchronise the local versions of an ontology (i.e. replicas) with the shared version (i.e. the original) from which they were derived. In [100] the authors defined essential factors for such a data interchange format. We adopted this approach in the following way:

(i) the format must be agreed on, which is attained through the evolution ontology;

(ii) the format must faithfully follow the accepted data model, which is achieved by defining the hierarchy of the concept "*Change*" in the evolution ontology;

(iii) the data must be structured consistently in the ways that are readily computer processable, which is accomplished by introducing the properties "*hasPreviousChange*" and "*causesChange*" (see section 3.2.4) enabling to reconstruct a sequences of performed changes and the concept "*LOG*" that is the end of a list of performed changes and indicates the last one with the property "*lastChange*".

Apart from the distributed evolution, the evolution log is also used to provide undo-redo capabilities. Therefore, it is necessary to distinguish reverted changes. Namely, it might happen that after synchronisation between an original and its replica, the changes in the original are cancelled. Although the current state of the original is equivalent to the state before applying this change, the information about reverting a change (and all its consequences) has to be explicitly stored in a log in order to inform replicas about this modification since the changes that are reverted might be already applied to some of replicas.

The evolution log is used to discover changes performed on the original but not applied on the replica. Note that replicas cannot be changed. Rather, they could be only extended in the ontologies that include them, as can be concluded from the ontology modularization definition (see Definition 34).

The kernel of the synchronisation between original O and its replica O' is to find deltas, i.e. changes that are necessary and sufficient to transform O' into an updated version of O such that ontologies O and O' are equivalent ontologies[72]. This puts an additional requirement on the evolution ontology. Each required ontology change has a version number associated with it, indicating the current ontology version. Moreover, as a result of the ontology modularization, for each ontology change it is needed to define a model it is applied to. Therefore, for each ontology change two additional attributes are defined: the version attribute "*versionChange*" and the ontology attribute "*OI-model*".

All changes in an ontology are stored in the evolution log of this ontology as instances of subconcepts of the concept "*Change*" defined in the evolution ontology as well as the

[71] The evolution ontology specifies allowable changes together with their inputs, effects, and constraints. The evolution log specifies data that are tracked regarding changes made to the domain ontology that is changing.

[72] The function *equivalent* is defined at the beginning of this section.

corresponding property instances as shown in Figure 17. However, since one change might provoke many additional changes, the evolution log does not have a linear structure. It is organised as a list of performed changes where the order of changes in a list is defined through the "*hasPreviousHistoryChange*" property. Each element of a list is represented as a tree of all its consequences (i.e. generated changes) where the property "*causesChange*" represents the cause-consequence relationship between performed changes. This is shown in Figure 54. There is one and only one instance of the concept "*LOG*" (cf. *InstLog*) that indicates the end of the list, i.e. the lastly required change. We note that each $InstCh_i$ denotes a performed change. Further, both relationships (cf. "*hasPreviousHistoryChange*" and "*causesChange*") are required for the realisation of the distributed ontology evolution. The first one enables the navigation through the evolution log whereas the second one enables the reconstruction of the request and all its consequences. More information about these properties is given in section 3.2.4.

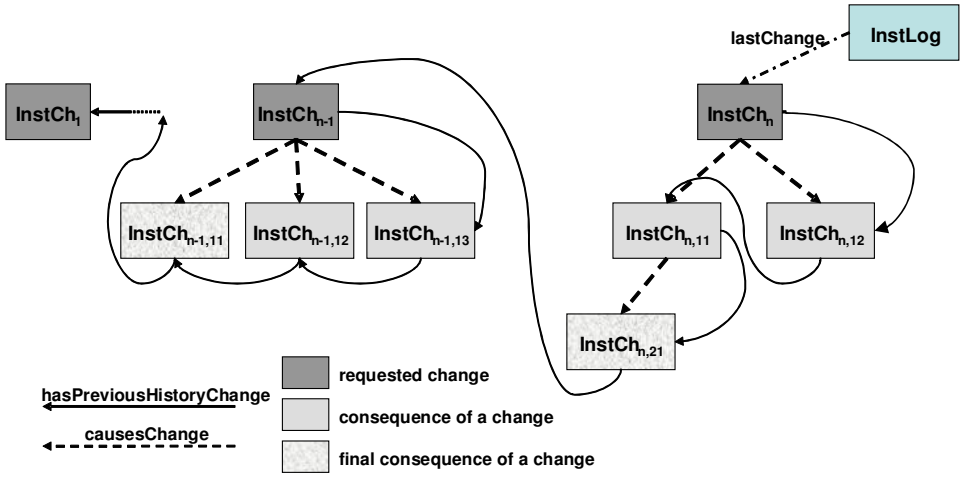


Figure 54. Evolution log as a list of trees

The fact that one change is explicitly requested (and therefore is not a consequence of other changes) is represented explicitly through the "*requestedChange*" attribute defined in the evolution ontology. This information can be considered as redundant since it can be always derived from an evolution log based on the rule that each change that is not caused by any other change is a requested change. However, it is tracked in the evolution log in order to optimise the performance on the procedure for the traversing the evolution log. The main reason is that the evolution log might contain a huge number of instances. Therefore, reasoning on them is a time-consuming activity. There is no optimisation technique that can help due to the need for negation in a particular query[73] for the requested changes, which implies that all instances have to be taken into account for each evaluation of this query.

For the evolution log we define a procedure for the logging of changes. We assume that a requested change and all its consequence (i.e. induced changes) are organised in a tree-like structure. Since these changes have to be applied (and tracked) in a sequence determined by the cause-consequence relations between them, there is a need to transform a tree into an

[73] By adopting the F-Logic syntax for queries, all the required changes can be found by the following query: FORALL X <- X:Change AND NOT EXISTS Y Y:Change AND Y[causesChange->X].

ordered list. We apply the depth-first[74] traversal method in order to create the list i.e. to instantiate the "*hasPreviousHistoryChange*" property between changes. This procedure is shown in Figure 55.

Algorithm Depth First Traversal

DFT($\mathcal{CH}$, $\mathcal{TREE}$, $\mathcal{LIST}$)

Require: $\mathcal{CH}$ = change; $\mathcal{TREE}$ = tree of cause-consequence changes; $\mathcal{LIST}$ = list of changes

1: /*Add change CH into list LIST*/
2: VISIT($\mathcal{CH}$, $\mathcal{LIST}$)
3: /*Find next change that has to be visited*/
4: **for all** change $\mathcal{ICH}$ from $\mathcal{TREE}$ such that $\mathcal{ICH}$ is caused by $\mathcal{CH}$ directly **do**
5: **if** $\mathcal{ICH}$ was not visited yet **then**
6: DFT($\mathcal{ICH}$, $\mathcal{TREE}$, $\mathcal{LIST}$)
7: **end if**
8: **end for**

Figure 55. A procedure for transforming a tree of changes into an ordered list of changes

As can be noticed we applied a preorder[75] traversal [110] that consists of first visiting the root, and then executing a preorder traversal on each of the root's children (if any).

Figure 56 illustrates the depth-first traversal of the tree shown in Figure 45. The procedure starts from the *RemoveConcept("Project")* change and progresses by expanding the first child node of the search tree that appears, i.e. the *RemoveConcept("Industrial Project")*. It goes deeper and deeper until it hits a node that has no children, i.e. *RemoveSubConcept ("Industrial Project","Project")*. Then the search backtracks and starts off on the next node, i.e. *RemoveConcept("Research Project")*. The procedure continues until it reaches a point at which there is no node in a tree that has not already been visited.

The given procedure defines how to transform a request for a change and all its consequences from a tree-like structure into an ordered list. The first element of the list is the required change. The last element of a list is called the final consequence[76]. The property *"hasPreviousHistoryChange"* is instantiated between successive elements of the list. The next step would be to connect the list with the already performed changes that are stored in an evolution log. To do that, the last performed change (referenced by *InstLog* in Figure 54) becomes the previous change for the final consequence and the requested change turns into the last performed change.

Resolving Replication Inconsistencies

As shown in Figure 53, resolving replication inconsistencies is performed through three additional components. It is initiated by specifying an original whose included replicas should be updated. Subsequently, we describe how it is performed.

[74] It is a graph search algorithm that extends the current path as far as possible before backtracking to the last choice point and trying the next alternative path.
[75] The term *preorder* refers to the fact that a node is visited *before* any of its descendents.
[76] It is indicated by the instantiation of the attribute "*finalConsequence*" that is defined in the *Evolution Ontology*.

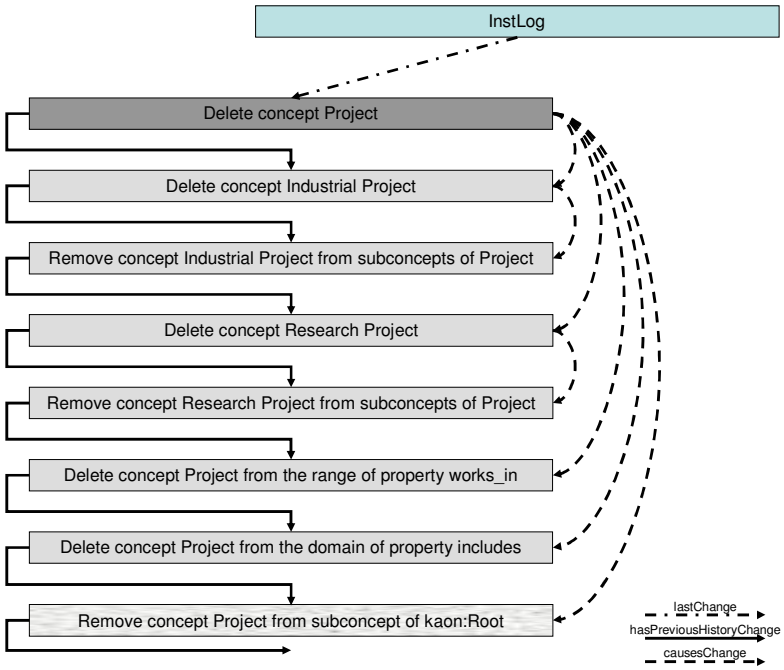


Figure 56. The order of visiting changes from Figure 45

Identification of Changed Originals

This step first checks whether the resolution of the replication inconsistency can be performed at all. If for some directly included replica the original has the replication inconsistency, then the process is aborted. Otherwise, a list of directly included replicas having the pending replication inconsistency (but whose original is replication consistent) is determined. Since the dependent ontology consistency for the ontologies on the same node is required, this approach is recursively applied on the all ontologies that include the ontology whose replication inconsistency is resolved.

More formally, the distributed ontology evolution of an ontology *OIM* can be performed if the following condition is satisfied:

$$\forall OIM_i \in INC(OIM)\ replicationConsistency(OIM_i).$$

Let's assume that the service provider D from Figure 52 wants to resolve the replication inconsistency of *IO*. Its directly included replicas, namely *PO* and *SO* are examined. For each of them the replication consistency of the original is checked. If *PO* at the service provider B has the replication inconsistency (due to changes from the service provider A in *BO* which have not been applied at B's replica of *BO*, i.e. *BO'*), then the process is aborted. If *PO* at the service provider B is replication consistent, but *PO* at service provider D is not (since *PO* at B has been changed), then *PO* is scheduled for further analysis. The consistency of the *PO'*s original is required since *IO* will obtain changes from *BO* through *PO'*s and S*O'*s evolution log.

In order to optimise this step, the set of directly included ontologies to be taken into account may be reduced by eliminating all the directly included ontologies that are available through some other paths. In the case that the ontology *IO* directly includes the ontology *BO*, the ontology *BO* would be eliminated from further consideration since it can be obtained through *PO* and *SO*.

The replication consistency is performed by determining the equivalence of the ontology with its original and by recursively determining the replication consistency of included ontologies. The following information is needed to perform that:

- Each ontology contains a physical URI of its original;
- Each ontology contains a physical URI of its evolution log;
- The version number of the replica and the original are used for checking the equivalence. It can be done by a simple comparison of their version number.

Extraction of Deltas

After determining directly included replicas to be updated, the evolution log for these ontologies is accessed. The location of the evolution log is specified within each ontology and is copied to replicas. For each log the extracted deltas contain all changes that have been applied to the original after the last update of the replica as determined by the version numbers.

Here we define the synchronisation between an original and its replica formally:

Definition 42 ***Synchronisation Set***: Given an ontology *O*, its replica *R*, the evolution log *ELO* of the ontology *O* and the evolution log *ELR* of the ontology *R*, a synchronisation set *Deltas(O,R)* is a set of changes that applied to *O* results in *R*.

$$Deltas(O,R)=\{Ch_i \mid Ch_i \in ELO \wedge Ch_i \notin ELR\}$$

A set of changes *Deltas* satisfies the additional constraints:

$$\forall Ch_i \in Deltas(O,R)\ \exists N\ (Ch_i, \text{“versionChange”}, N) \in ELO \wedge N > version(R) \wedge$$
$$\exists OIM\ (Ch_i, \text{“OI-model”}, logicalURI(OIM)) \in ELO \wedge OIM \in O \cup INC^*(O),$$

where “*versionChange*” and “*OI-model*” are properties of the concept “*Change*” defined in the evolution ontology (see section 3.2.4). The triplet *(Ch_i, “versionChange”, N)* specifies that the instance “Ch_i” has the value “*N*” for the property “*versionChange*”. The property “*OI-model*” indicates the ontology the change is applied to. The logical URI of the ontology (i.e. *logicalURI(OIM)*) is used as a target value of the property instance.

Therefore, the synchronisation set *Deltas* contains only changes Ch_i (i) whose version number *N* is greater than the version number of the replica *R* (i.e. *version(R)*) and (ii) which are applied to either to the original *O* or to the ontologies that it includes (i.e. $INC^*(O)$). This formal definition enables the formal verification whether the distributed evolution can be performed since it might happen that evolution logs are corrupted (e.g. manually changed). In this way it is possible to guide an ontology engineer through the evolution by providing additional information such as why the evolution did not succeed or what else she has to do in order to finish it.

We note that the synchronisation set delta might not be minimal[77] since it may be a result of undo-redo modifications. However, the minimal synchronisation set would not reflect the history of modifications of an ontology (e.g. undo/redo operations), which is necessary for the distributed evolution.

[77] It means that the removal of some changes from this set might result in the set that is also the synchronisation set from O to R.

Further, it is not sufficient only to extract deltas from an evolution log. The changes forming delta have to be organised in the same way as they are performed. More formally:

$$R = Ch_n \circ \ldots \circ Ch_{i+1} \circ Ch_i \circ \ldots \circ Ch_2 \circ Ch_1 \circ O =$$

$$Ch_n(\ldots(Ch_{i+1}(Ch_i(\ldots(Ch_2(Ch_1(O)))))))),$$

where:

$$Ch_i \in Deltas \wedge Deltas \subseteq ELO \wedge Deltas \cap ELR = \varnothing$$

The order of changes forming delta is defined by the property "*hasPreviousHistoryChange*" in the following way:

- Ch_n is the last change applied to the original as indicated by the unique instance "*instLog*" of the concept "*LOG*" (see Figure 54), i.e.

  $$(instLog, \text{“}lastChange\text{”}, Ch_n) \in ELO,$$

  where "*lastChange*" is a property defined in the evolution ontology, $instLOG \in instconc(\text{“}Log\text{”})$ and "*LOG*" is a concept defined in the evolution ontology;

- Ch_i , $1 \leq i < n$, is calculated in the following way:

  $$(Ch_{i+1}, \text{“}hasPreviousHistoryChange\text{”}, Ch_i) \in ELO$$

  where the Ch_1 has to satisfy addition conditions:

  $$\exists N\ (Ch_1, \text{“}versionChange\text{”}, N) \in ELO \wedge N = version(R) + 1 \wedge$$

  $$(Ch_1, \text{“}finalConsequence\text{”}, \text{“}true\text{”}) \in ELO.$$

Therefore, changes Ch_i from the synchronisation set *Deltas* are ordered in a sequence according to the *"hasPreviousHistoryChange"* property. The sequence of changes starts with the change Ch_1 that was the first change applied to the original after the last synchronisation as indicated by the version number *N* and the attribute "*finalConsequence*". The sequence of changes terminates with the lastly applied changes Ch_n that is referenced by *InstLog*.

Merging Deltas

Deltas extracted from evolution logs in the previous step are merged into a unified list of changes. Since an ontology can be included in many other ontologies, its changes will be included into evolution logs of all of these ontologies. Hence, the merging process must eliminate duplicates. Also, changes from different deltas caused by the same change from a common included ontology should be grouped together.

For example, if the ontology *BO* is changed, the evolution logs of the *PO* and *SO* will contain these changes as well as their own extensions. Hence, when changes from *PO*'s and *SO*'s logs are merged in order to update *IO*, the changes to *BO* will be mentioned twice. Thus, only one change to *BO* should be kept while discarding all others. However, changes in *PO* and *SO* caused by the same change in *BO* must be grouped together.

We note that it would not make more sense to update the evolution log only with changes local to the current ontology and follow evolution log pointers for imported ontologies to see "the complete changes". This solution would require the access to the evolution logs of all replicas included either directly or indirectly. There are several disadvantages of this approach. The inclusion relationship may be complex which leads to the reading of the huge number of the evolution logs. Further, the evolution logs may be also distributed since the

included ontologies may be on different nodes. Thus, the access to the several nodes on the network is needed. Finally, the extraction of the changes requires one query per each evolution log. Consequently, the proposed solution may be a time-consuming activity. On the contrary, the proposed approach can accelerate this process significantly since it extracts changes only from the evolution logs of the first level replicas. The price that has to be paid is the elimination of duplicates. However, the merging changes into one common list can be performed very fast since the lists of changes that have to be merged are already sorted.

Algorithm for the Distributed Ontology Evolution

The algorithm of our approach for the evolution between distributed ontologies is presented in Figure 57. It starts with the identification of changed originals (cf. 2). This includes the checking whether the evolution can be performed at all and returns all included replicas that are out-of-date (cf. 11--19). For each of these replicas, the evolution log is accessed in order to extracts deltas (cf. 5--6). These changes are merged with the changes from the other evolution logs (cf. 8). Finally, this integrated list of deltas from all out-of-date replicas is processed using the dependent ontology evolution algorithm (cf. 10) shown in Figure 50.

The complexity of this algorithm is not critical at all. The complexity of the identification of changed original is $O(N)$, where N is the number of directly included ontologies. The module for extracting deltas depends on the number of changes in a linear way. The complexity of the merging of two sorted list of size M_1 and M_2 is $O(M_1+M_2)$ since we consider each element of each list exactly once. Therefore, the processing time is directly proportional to the combined number of elements in the two lists.

Algorithm Distributed Ontology Evolution Algorithm

```
UPDATEDISTRIBUTEDONTOLOGY(o)
Require: o - ontology that have to be updated
 1: /*Identification Of Changed Originals*/
 2: inconsistentReplicas=identificationOfChangedOriginals(o)
 3: for all inconsistentReplica in inconsistentReplicas do
 4:    /*Extraction of Deltas*/
 5:    evolutionLog=findEvolutionLog(inconsistentReplica)
 6:    deltas=readEvolutionLog(evolutionLog)
 7:    /*Merging Deltas*/
 8:    changes=mergeDeltas(deltas)
 9: end for
10: evolveOntologies(changes,o)

IDENTIFICATIONOFCHANGEDORIGINALS(o)
Require: o - ontology that have to be updated
11: replicas=findFirtsLevelReplicas(o)
12: for all replica in replicas do
13:    includedOntologies=findAllIncludedOIModels(replica)
14:    for all includedOntology in includedOntologies do
15:       if includedOntology is not replication consistent then
16:          generate exception("Included models are not updated yet!")
17:       end if
18:    end for
19: end for
```

Figure 57. Distributed Ontology Evolution Algorithm

Discussion

Similarly to the dependent ontology evolution (see section 5.3.1), in the distributed case also it is not always possible to propagate changes from an original to its replica. For example, it may happen that the original and the replica (i.e. an ontology that includes the replica) are independently changed in the same way. Let's consider that the developer of the original found out that it should be extended with the subconcept relationship between two concepts. The developer of the dependent distributed ontology, which includes the replica, came to the same conclusion. They also established the subconcept relationship between the same concepts without knowing that the original is updated as well. Note that the extensions in the original and in the dependent ontologies are different due to the fact that the identifier of each ontology element consists of the identifier of the ontology it belongs to and of its own identifier. Since the subconcept relationships are added in the different ontologies (i.e. in the original and in the dependent distributed ontology) they are considered as independent.

In the case that the developer of the dependent distributed ontology wants to actualise her copy of the original, the synchronisation cannot be performed. The preconditions for the addition of the subconcept relations include, among others, the condition that this relation is not in a model. There, the attempt to apply "*AddSubConcept*" change that is performed on the original and therefore is tacked in the evolution log, would fail since the subconcept relation is already defined in the dependent distributed ontology. In order to resolve this problem, an ontology evolution system has to offer several alternatives:

- to break dependencies between the original and its replica;
- to initiate the inverse change (e.g. "*RemoveSubConcept*"), which would prepare conditions for synchronisation;
- to leave the ontologies as they are, which means staying compliant with the last version of the original.

The most frequent problem is related to the cardinality constraints, which define how many times a property may be specified for instances of a concept. Namely since the included ontology and the including ontology might instantiate instances independently, it might happen that synchronisation between them is not possible due to an exaggerated number of property instances. By introducing different levels of the ontology consistency (see section 2.3), the resolution point can be deferred. Moreover, an ontology engineer is informed about temporary inconsistency and possible ways of dealing with this problem.

5.4 Case Study

To demonstrate the usefulness of our approach for the evolution between dependent ontologies, we applied it to the MeSH[78] (MEdical Subject Headings). MeSH is a controlled vocabulary used for indexing medical documents. The goal of the MeSH is to provide a reproducible partition of concepts relevant to biomedicine for the purpose of organising knowledge and information. In biomedicine and related areas, new concepts are constantly emerging, old concepts are in a state of flux and terminology and usage are modified accordingly. To accommodate these changes, the MeSH has to be updated as well as the articles indexed by the MeSH. Indeed, the main reason for using the MeSH as a case study is that the National Library of Medicine (NLM) produces the MeSH with an annual update

[78] http://www.nlm.nih.gov/mesh/

cycle. Since the MeSH is used in real medical systems, management of its change is a critical issue.

The NLM has produced the MEDLINE[79] database since 1966. The MEDLINE database includes over 10 million literature quotations of articles written in 41 languages. Each article is indexed with the MeSH descriptors assigned by an individual who reads the article in its original language and assigns the descriptors to indicate what the article is about. About 400.000 articles are indexed per year. The MeSH is now in its 40th year of production and is added to and otherwise modified on an annual basis. Beginning in 2002, over 2.000 completed references are added daily each Tuesday through Saturday, January through October (over 460,000 added last year). These modifications are then applied to the MEDLINE database; articles are not re-indexed, but the database is kept current with the current version of the MeSH. This is a time-consuming activity since two months (November and December) are needed to make the transition of the NLM to a new year of the MeSH vocabulary used to index the articles.

According to the official MeSH web site[80], the following changes are applied on the MeSH version from 2003:

- 666 descriptors were added representing topics with no directly corresponding descriptors in the MeSH version used in 2003. The most recent examples of such additions are "Severe Acute Respiratory Syndrome" and "SARS Virus";
- 109 descriptors were replaced with more up-to-date terminology;
- 20 descriptors were deleted;
- 484 terms were added.

The practical experiences with the MEDLINE show that it is easy to add something (either a descriptor to the MeSH or an indexed article to the MEDLINE), but it is hard to modify data that are already in the system. The authors of the MEDLINE system found out that meaning of change is important and that there is a need for an update model [90].

The goal of the MeSH/MEDLINE case study was to show that:

- our ontology evolution system is able to work with large ontologies such as the MeSH. The newest version of the MeSH (MeSH 2004) contains 22.568 descriptors, 83 qualifiers and 137.557 supplementary concept records. The meaning of the MeSH entities is described in section 5.4.1;
- the dependent/distributed ontology evolution might be applied on the MEDLINE since the MeSH itself consists of several independent parts and the medical articles are only annotated by the MeSH;
- formal semantics provided by an ontology might be useful to improve the indexing in the existing MEDLINE system.

Our work regarding the MeSH can be split into three phases:

- Phase 1 – the representation of the MeSH in the form of the KAON ontologies;
- Phase 2 – the evaluation of the applicability of the ontology evolution support on the MeSH/Medline;
- Phase 3 – the suggestions for the continual improvement of the MEDLINE.

[79] http://www.nlm.nih.gov/pubs/factsheets/medline.html
[80] http://www.nlm.nih.gov/pubs/techbull/nd03/nd03_mesh.html

These phases are subsequently described.

5.4.1 Phase 1

Our first task was to transfer all information available in the MeSH repository into the KAON system (see section 7.2) in order to verify whether our ontology evolution system can be used at all. This required (i) the understanding the MeSH and (ii) the creation of the KAON ontologies that mimic the MeSH.

Understanding the MeSH requires an understanding of its structure. There are three major components to the MeSH:

- descriptors;
- subheadings (also known as Qualifiers);
- supplementary concepts.

Descriptors (e.g. "*Headache*") are the main headings. Qualifiers (e.g. "*Therapy*", "*Diagnosis*", etc.) are used with descriptors and afford a means of grouping together the documents concerned with a particular aspect of a subject. Indeed, qualifiers are used to modify (refine) descriptors by indicating particular aspects. They are used in indexing, cataloguing, and online searching to qualify the MeSH descriptors by pinpointing some specific aspect of the concept represented by the descriptor. For example, "*LIVER/drug effects*" indicates that the article or book is not about the liver in general but about the effect of drugs on the liver. Supplemental (e.g. "*Ametohepazone*") is added daily and is largely chemicals.

The MeSH structure is centred on descriptors, concepts, and terms [89]. A descriptor is viewed as a class of concepts, and a concept as a class of synonymous terms within a descriptor class. Indeed, a descriptor class consists of one or more concepts closely related to each other in meaning. For example, for the "*Headache*" descriptor the concepts "*Headache*" and "*Sharp Headache*" are defined. For the purposes of indexing, retrieval, and organisation of the literature, these concepts are best lumped together in one class. Each descriptor has a preferred concept. Further, one of the terms naming that concept is the preferred term of the preferred concept, and takes on the role of naming the descriptor. Each of the subordinate concepts also has a preferred term, as well as a labelled (broader, narrower, related) relationship to the preferred concept. Terms meaning the same are grouped in the same concept. For the previously mentioned descriptor "*Headache*" following terms among others are defined "*Head Pains*", "*Head-Pain*", "*Cephalgias*".

An example is shown in Figure 58. It can be seen that concept classes II and III are respectively, narrower and related to concept class I (the preferred concept), but are not equivalent to each other. Each concept class could be given its own definition if desired. It can also be seen that "*HIV Encephalopathy*" and "*AIDS Encephalopathy*" are synonymous terms within the same concept class.

Relationships among concepts can be represented explicitly in the thesaurus, most notably as relationships within the descriptor class. Hierarchical relationships are represented as broader or narrower (parent-child) relationships between concepts within descriptors. Other types of relationships include associative relationships such as the Pharmacological Actions or see-related cross-references as well as forbidden combination expressions such as the Entry Combination. For example, the MeSH concept "*Headache*" is broader than the MeSH concept "*Bilateral Headache*", the MeSH concept "*Sharp Headache*" is narrower than the MeSH concept "*Head Pains*" or the MeSH concepts "*Headache*" and "*Head Pains*" are related.

AIDS DEMENTIA COMPLEX [Descriptor Class]
Concept Class I - Preferred Concept
Terms: AIDS Dementia Complex (Preferred Term)
HIV Dementia
HIV-Associated Cognitive Motor Complex
Dementia Complex, AIDS-Related
Concept Class II - Subordinate Concept (narrower)
Terms: HIV Encephalopathy (Preferred Term)
AIDS Encephalopathy
Concept Class III - Subordinate Concept (related)
Terms: HIV-1-Associated Cognitive Motor Complex (Preferred Term)

Figure 58. An example of the MeSH descriptors

Three kinds of informative references may be found in descriptor records: "*see related*", "*consider also*", and "*main heading/subheading combination*" references. "*See related*" references indicate the presence of other descriptors that are conceptually related to the topic. The "*consider also*" notation is primarily used on anatomical descriptors. The "*main heading/subheading combination*" notations refer an invalid (and prevented) combination of descriptors.

Based on the analysis of the MeSH structure, we develop several ontologies. They are shown in Figure 59. The goal was to model all information that exists in the MeSH model including the implicit knowledge. Therefore, the approach can be summarised as follows:

- the model of the MeSH is transformed into the MeSH ontologies;
- "hidden" (hard-coded) knowledge embedded in the MeSH is translated into a set of rules in the corresponding ontologies and is used in typical inferencing tasks.

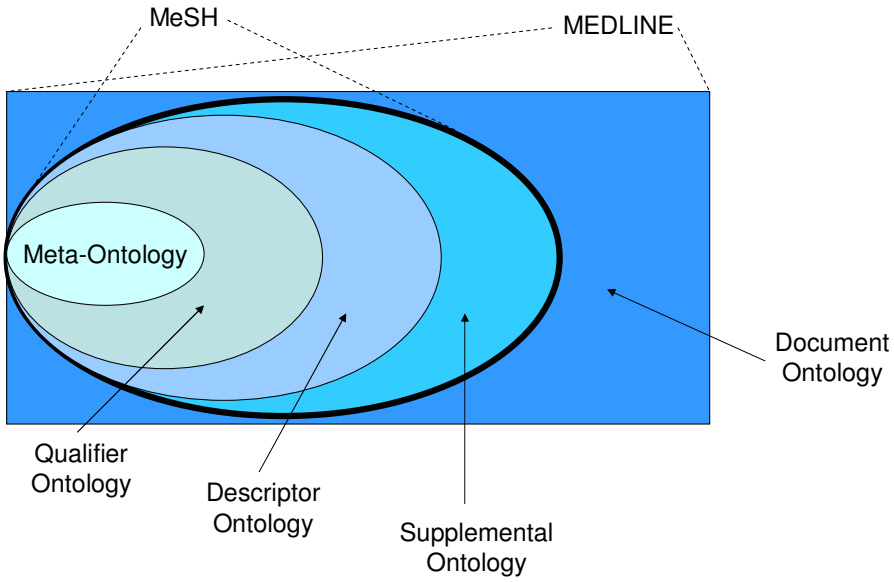


Figure 59. Representation of the MeSH and the Medline as KAON ontologies

The *Meta-Ontology* shown in Figure 60 represents the conceptual model of the MeSH. It contains concepts such as "*Descriptor*", "*Qualifier*", "*Supplemental*", "*Concept*", "*Term*", etc. The relationships between them are defined according to the MeSH model. We extend this model by representing explicit all information that was implicit in the MeSH model. The semantic of the MeSH model is implicit, hidden in the XML files and difficult to discover. By providing explicit semantics of the MeSH relationships, it is possible to perform the formal verification of a model. Such an approach is described in section 5.4.3.

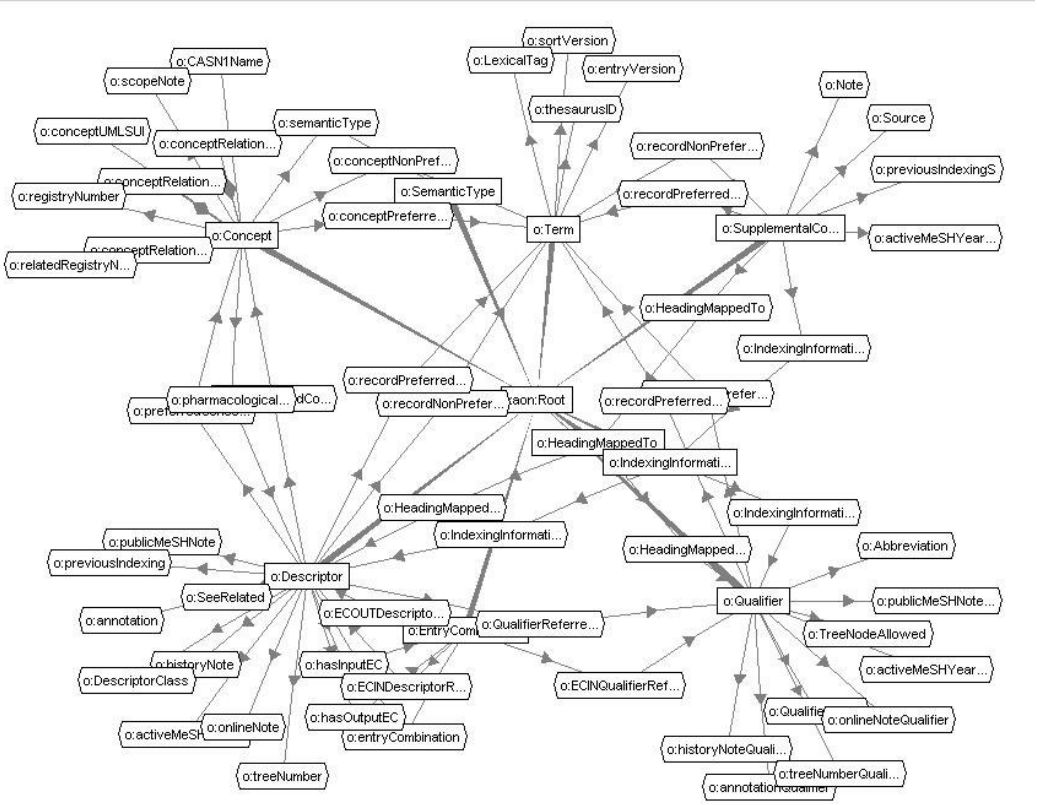


Figure 60. The Meta-Ontology representing the conceptual model of the MeSH

Actually, in an ontology there are two types of implicit knowledge: the *axioms* and *general rules*. *Axioms* are the standard set of rules such as rules for symmetric, transitive and inverse properties. For example, if A "*is related to*" B, B "*is related to*" C, and "*is related to*" is a transitive property, then the ontology system can infer that A "*is related to*" C as well. Thus, we do not need to express this information explicitly. *General rules* are domain specific rules that are needed to combine and to adapt information available in the ontology. They are used to specify the relationship between ontology entities in the form of rules. For example, if C "*is preferred* ***concept*** *for*" D and T "*is preferred* ***term*** *for*" C, then it can be concluded that T "*is preferred* ***term*** *for*" D.

In general, axioms and rules are used to infer new knowledge. The possibility to derive information makes the model of a domain more concise, more accurate, and easier for maintenance. Obtaining and formalising the non-explicit but available knowledge about the knowledge model of the MeSH ensures the advantages over other medical systems.

The implicit knowledge of the MeSH is explicitly modelled through rules. For example, for the "*see related*" relationship, the symmetry axiom may be exploited when searching for information. Without the definition of this axiom, searching for related descriptors might

depend on the way metadata was provided for them. If one defines that some descriptor named "*X*" has the "*see related*" relationship with some other descriptor named "*Y*", there is no possibility (without programming or explicit specification) to find out that the descriptor "*Y*" also has the "*see related*" relationship with the descriptor "*X*". Further, it is impossible to conclude that the descriptors "*X*" and "*Y*" cannot be in the "*main heading/subheading combination*" relationship to each other.

The *Qualifier ontology* is based on the Meta-Ontology since it defines the structure of the MeSH/MEDLINE system. It contains all the MeSH qualifiers that are represented as subconcepts of a concept "*Qualifier*" that is defined in the Meta-Ontology.

The *Descriptor ontology* contains information about the concrete MeSH descriptors. Therefore, it also reuses the Meta-Ontology. Further since one descriptor may reference to the qualifier concepts, the Descriptor ontology includes the Qualifier ontology as well. For example, the descriptor "*Calcimycin*" has a reference to the qualifier "*abnormalities*".

The *Supplemental ontology* reuses the Qualifier and the Descriptor ontologies directly and indirectly the Meta-Ontology through both of the directly included ontologies. It specialises the concept "*Supplemental*" defined in the Meta-Ontology according to the MeSH context. Moreover, it establishes the reference between qualifiers, descriptors and supplemental concepts.

The MeSH is used for indexing biomedical articles. This information is stored in the MEDLINE. Each index (or annotation in the Semantic Web terminology) consists of the MeSH *headings* and *chemicals*. Each MeSH *heading* contains one pair or more pairs of descriptors and qualifiers. Each pair defines a main topic of the article and is considered as a whole. On the other hand, *chemical* contains a supplemental concept that describes more specific topics of an article. To model this information we have developed the so-called *Document ontology*. It contains only the metadata about biomedical articles and not the articles themselves. It includes all the previously mentioned ontologies. We note that we transfer all information about the MeSH but only about 100.000 indexed documents. The reasons for selected the MEDLINE subdomain are discussed later.

5.4.2 Phase 2

In the second phase we evaluate the possibility to apply our ontology evolution system to the set of the ontologies generated from the MeSH/MEDLINE system. It is worth noting that we cannot compare our system with the existing MEDLINE system due to two reasons:

1. There is no MEDLINE maintenance system that enables keeping consistency. For example, after removal of some descriptor from the MeSH, it might be possible that some articles are still indexed with the descriptor that does not exist any more. All changes are performed manually. Thus, any modification is a time-consuming and error-prone activity;
2. The MeSH is available on the Internet NLM home page at http://www.nlm.nih.gov/mesh/filelist.html. However, the MEDLINE[81] can be searched free of charge but access to the MEDLINE services is provided by organisations that lease the database from NLM. Therefore, we were not able to work with the full content of the MEDLINE. We manually downloaded about 100.000 articles and their annotation by making query about "*Headache*" and parsing the XML output. Even though this restriction is made, the evaluation results are applicable.

[81] http://www.nlm.nih.gov

As already mentioned, the goal of this phase is to demonstrate that the evolution of medical vocabulary can be automated. The first application of our ontology evolution system was during the creation of the KAON version of MeSH/MEDLINE ontologies. We found several anomalies (such as redundancies, inconsistencies and undefined entities) in the existing MeSH/MEDLINE data. For example, several descriptors were defined twice. Moreover, in the XML file each reference is stored through two elements: entity ID and entity names. Since one entity is referenced in several entities, different names are used for the same entity. Note that synonyms are represented as terms. Finally, we found references to the undefined entities. This problem may be a consequence of a syntax error in the XML file or may be a consequence of the manual change propagation procedure since the people might not find all effects of a change.

Since the ontology evolution system was applied during the creation of the MeSH/MEDLINE ontologies, all these anomalies were prevented. Here we show how the consistency can be enforced when the initial consistent ontologies already exist.

Therefore, the result of the first application of the ontology evolution system is a set of consistent MeSH/MEDLINE ontologies. Then, we try to modify these ontologies using our ontology evolution system. We decide to modify the Descriptor ontology since descriptors are created for the purpose of indexing the medical literature. Since the worst case is the concept deletion, we measured time needed to perform this change and the number of generated changes. Note that there is no goal system that can be used for comparison. In the MEDLINE system the semantics of change as well as the change propagation are performed manually. Therefore, we only wanted to show that the removal could be performed in acceptable time, which is much faster and more accurate than in the existing system. Since we selected the subdomain of "*Headache*" diseases and included articles about this topic and their annotation into the Document ontology, the descriptor "*Headache*" is chosen for removal. It is represented as the concept "*Headache*" in the Descriptor ontology. It is visible in all the ontologies that reuse the Descriptor ontology. Thus, the request for the removal of the concept "*Headache*" might have consequences on the Supplemental ontology and the Document ontology as well (see Figure 59). Consequently, the ontology evolution between dependent ontologies has to be applied since the synchronisation between the Descriptor ontology and the ontologies that include it is necessary for a consistent and, therefore, efficient, effective and accurate system.

Note that the Meta, the Qualifier, the Descriptor and the Supplemental ontologies are stored within one ontology server and the Document ontology is stored on a separate ontology server. Thus, by changing the Descriptor ontology, we were able to apply all ontology evolution "types". The single ontology evolution is applied to the Descriptor ontology, the dependent ontology evolution is applied to the Supplemental ontology since it reuses the Descriptor ontology through the inclusion while the distributed ontology evolution is applied to the Document ontology since it reuses the Supplemental ontology through the replication. Consequently, the results that are obtained for the Descriptor ontology and the Supplemental ontology are completely correct whereas the results obtained for the Document ontology are only the approximation since this ontology contains only a part of all MEDLINE articles.

Even though the Descriptor ontology contains 2.417.584 entities, our ontology evolution system was able to perform the deletion of the concept "*Headache*" in this ontology in 218 seconds. The removal of that concept in the Supplemental ontology (that includes the Descriptor ontology as well) lasted about 50 seconds longer since there are not so many entities in the Supplemental ontology that have reference to the concept "*Headache*" from the Descriptor ontology. The removal in the Document ontology took 1.583 seconds since almost all documents are annotated. Note that the complexity of the dependent ontology evolution

depends on the number of instances in a linear way. Therefore, the existence of more instances (i.e. annotated articles) will linearly increase the time needed to perform a change.

The following set of additional changes was generated:

- 58 changes in the Descriptor ontology;
- 13 changes in the Supplemental ontology
- more than 100.000 changes in the Document ontology.

The changes in the Descriptor ontology cover the removal of properties defined for the concept "*Headache*" and their consequences. Moreover, there are several subconcepts of the "*Entry Combination*" concept that establish the reference between the descriptor headache and corresponding qualifiers. All of them have to be removed as well. In the Supplemental ontology the descriptors are referenced through the property "*hasReferencedDescriptor*" and its specialisation. Therefore, the request for the removal of the concept "*Headache*" in the Descriptor ontology requires the removal of this concept from the range of all these properties. Finally, all the annotated articles were about headache. Therefore, the annotation of all of them must be updated.

We believe that the usability of the MEDLINE management system might be significantly improved by incorporating the approach presented in this thesis. It does not only guarantee consistency. Rather, it improves the usability of the system by informing the responsible persons about all the consequences of a change since only in that way would they be able to comprehend the impact of a change and undo the unnecessary changes. In the next section we discuss the way in which the formal semantics provided by an ontology can be further exploited.

5.4.3 Phase 3

The assignment of MeSH topics to articles of the MEDLINE system represents the state-of-the-art in human indexing. The professional indexers who perform this task have been trained for at least 1 year. Ten to twelve topics in the form Descriptor/Qualifier are associated to each article. Although such annotations help in searching for articles, the MEDLINE suffers from information overloading. For example, searching the MEDLINE using the MeSH topic "*common cold*"[82] yields over 1,400 articles written in the last 30 years. Finding a relevant article might take 20-30 minutes.

We applied the data-driven change discovery (see section 3.4.2) to improve annotations in the MEDLINE, since they are made manually. Since we assume that an annotation must be consistent with the underlying MeSH system, the "quality" of the annotation is assessed through the existence of redundancy, inaccurate or incomplete information. Note that we assume that the annotations are valid, i.e. all the metadata in the annotation is consistent with the MeSH ontologies. This is guaranteed by applying the dependent ontology evolution as described in the previous section, which provides support for finding inconsistencies and resolving them.

Three quality criteria are defined in the following way:

- *Compactness* – A semantic annotation[83] is not compact or it is redundant if it contains more metadata than it is needed and desired to express the same "idea". In order to

[82] The example is taken from http://www.ovid.com.

[83] An annotation consists of a set of ontology instances. We use term metadata as a synonym for an ontology instance.

achieve compactness (and thus to avoid redundancy), the annotation has to comprise the minimal set[84] of the metadata without exceeding what is necessary or useful. The repetition of the metadata or the usage of several metadata with the same meaning only complicates maintenance and decreases the system performance;

- *Completeness* – An annotation is incomplete if it is possible to extend the annotation only by analysing the existing metadata in the annotation in order to clarify its semantics. It means that the annotation is not finished yet and requires that some additional metadata have to be filled in;
- *Aggregation* – An annotation is aggregative if it contains a set of metadata that can be replaced with semantically related metadata in order to achieve a shortened annotation, but without producing any retrieval other than the original annotation.

Note that assessment is performed on the annotation level and that the MeSH structure (i.e. a set of the MeSH ontologies) is the basis for all measures. This assessment can help refine and improve the annotation in the MEDLINE.

In order to clarify the meaning of the criteria here we give a short example that simulates the real MEDLINE system. It is shown in Figure 61.

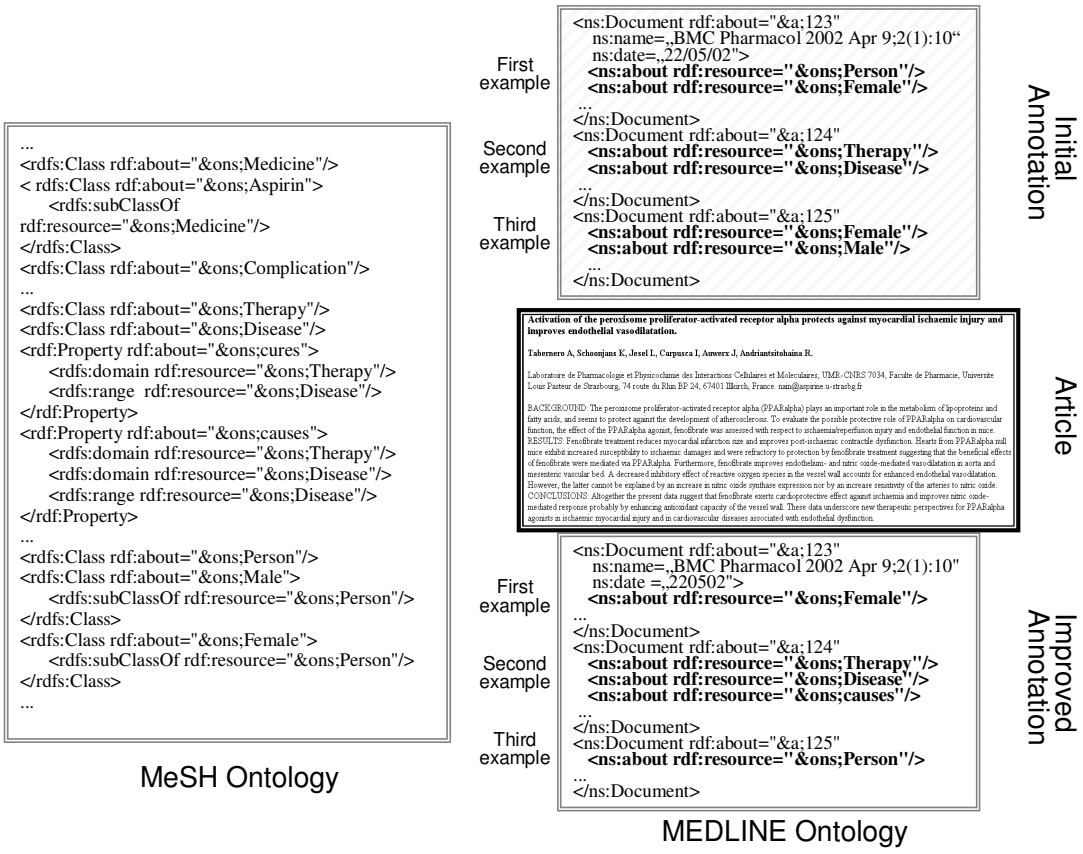


Figure 61. Annotation refinement based on the analysis the ontology structure and the existing annotations. The ontology is depicted in the left part. The right part shows downward the initial annotation, corresponding articles and the improved semantic annotation

[84] An annotation is not minimal if excluding metadata results in the same retrieval for the same query, i.e. if precision and recall remain the same.

Compactness

The concept hierarchy and the property hierarchy from the domain ontology are used to check this criterion. The first example in Figure 61 represents the incompact annotation because the article is annotated, after all, with the concept "*Person*" and its subconcept "*Female*". When someone searches for all articles about "*Person*", she searches for the articles about all its subconcepts (including "*Female*") as well. Consequently, she gets this article (minimum) twice. Moreover, such annotation introduces an ambiguity in the understanding of the content of an article, which implies problems in knowledge sharing. Let us examine the meaning of the annotation of a medical document using the set of metadata "*Person*", "*Female*", "*Aspirin*" and "*Complications*". Does it mean that the article is about complications in using aspirin only in females, or in all persons? When the second answer is the right one, then this article is also relevant for the treatment of male persons with aspirin. This implies new questions: is the annotation using metadata "*Female*" an error, or the metadata "*Male*" is missing? Anyway, there is an ambiguity in annotations, which can be detected and resolved by using our approach.

In order to prevent this, an article should be annotated using as special metadata as possible (i.e. more specialised sub-concepts). In this way, the mentioned ambiguities are avoided. Moreover, the maintenance of the annotations is also alleviated because the annotation is more concise and because only the changes linked to the concept "*Female*" (first example in Figure 61) can provoke changes in the annotation.

Completeness

This criterion is computed based on the structure of the ontology. For example, one criterion is the existence of a dependency in the domain ontology between the domain entities, which are already used in the annotation. The second example in Figure 61 contains concepts with many relationships between them (e.g. properties "*cures*" and "*causes*" exist between concepts "*Therapy*" and "*Disease*"). The interpretation is ambiguous since it is a question whether the articles are about how a disease (i) can be cured by a therapy, or (ii) caused by a therapy. In order to constrain the set of possible interpretations, the annotation has to be extended with one of these properties.

This problem is especially important when the repository of articles contains a lot of articles annotated with the same concepts because the search for knowledge retrieves irrelevant articles that use certain concepts in a different context. Consequently, the precision of the system is decreased.

Aggregation

This pattern for the annotation refinement occurs when an article is described with all subconcepts of one concept (e.g. concepts "*Female*" and "*Male*" as shown in the third example Figure 61). From the searching for articles point of view, it is the same whether an article is annotated using the combination of the concepts (e.g. "*Female*" and "*Male*") or using only the parent concept (e.g. "*Person*"). It is obvious that the second case of annotation makes the management much easier. Moreover, since the standard approaches to the ranking results of querying [133] exploit conceptual hierarchies, for example in a querying for persons an article annotated using "*Female*" and "*Male*" will be placed at the same level as an article annotated using only one of these concepts. However, it has to be ranked on the top level (level of the concept "*Person*") because it covers all subtypes of the concept "*Person*".

5.5 Related Work

Reusing ontologies in the Semantic Web context is hindered by the fact that the primary Semantic Web language – RDF(S) – does not provide any means for including elements from other ontologies. Within RDF(S) there is no notion of the model representing a subset of the statement on the Web. Instead, each RDF fragment can freely refer to any resource defined anywhere on the Web. This presents serious problems to tool implementers since it is not possible to reason over the entire Web. Recognising this shortcoming, many ontology languages, including but not limited to OIL [99], DAML+OIL [23], and OWL [103], provide means for declarative inclusion of other models.

However, most tools simply use these declarations for reading several files at the beginning and then create an integrated model. OilEd – a tool for editing OIL, DAML+OIL and OWL ontologies developed at the University of Manchester – does exactly that: importing an ontology actually inserts a copy of the original ontology into the current one. As mentioned above, this has drawbacks related to ontology evolution. On the other hand, tools such as Ontolingua [30] offer support even for cyclical ontology inclusion. However, to the best of our knowledge, these tools do not provide evolution of included ontologies. Protege-2000 [96] – a widely used tool for ontology editing developed at Stanford – provides the best support for ontology inclusion so far. In Protégé, it is possible to reuse definitions from a project by including an entire project. However, the implemented inclusion mechanism is too crude as it does not allow extension of included entities. For example, it is not possible to reclassify or add a slot to a class in the including model. Further, only the outermost model may be changed, thus making the evolution of dependent ontologies impossible.

Regarding the evolution between dependent ontologies there are only a few approaches. In [138] the authors define the management of the dependency between so-called component ontologies in an ontology as a whole. The dependency is investigated to see how many types exist and how to manage each of them. They found only two dependency types: (1) super-sub relation where there are at least two concepts in a subconcept relationship and each of them belongs to a different ontology, and (2) referring-to relation where a concept in one ontology refers to a concept in other ontology as a class constraint. This approach has several disadvantages. Firstly, two proposed types of dependency do not cover all possibilities that might arise on the Web. Here we give several examples for ontology dependency that can not be realised by using the proposed approach: (a) an ontology may extend the included ontology only by defining instances of the concepts from included ontology; (b) an ontology may specialise and generalise the concepts from included ontology at the same time; (c) an ontology may define domains, ranges, or property instances for a property in other ontology; etc.

The second drawback is related to the way of realising the reuse. The reuse is achieved through the copy of only included entities in the dependent ontology. Therefore, a dependent ontology does not know anything about other entities from the dependent ontology, which implies that any reasoning task (e.g. a query) requires a distributed query processing which is a time-consuming activity. It is even made worse by the fact that an included ontology might include other ontologies and so on, which results in a need for reasoning over the entire Web. Therefore, the reuse is achieved in only a syntactical way resulting in the same problem as mentioned for the RDF language at the beginning of this section.

The advantage of this approach is the provision of several ways to resolve inconsistency between dependent ontologies. They include the prohibition of changes that influences other ontologies as well as the modification of the influenced ontologies by accepting or rejecting

changes. However, the authors do not consider changes that cannot be applied due to the contradictions that they cause in the dependent ontology.

The problem of ontology revision, which is necessary in a dynamic environment such as the Web, is discussed in [53]. The authors describe the versioning mechanism that copes with this. They present SHOE, a web-based knowledge representation language that supports multiple versions of ontologies. Ontology reuse in SHOE is accomplished by extending general ontologies to create more specific ontologies. SHOE focuses on impact of an ontology change on the results of a query and on the instances. It maintains each version of the ontology and an instance must state which version it is referencing. However, the problem of the change propagation is not treated in this work. Therefore, an ontology engineer has to (i) discover that an ontology, that is referenced, is modified and (ii) check whether the instances are compatible with the new version or not. On the contrary, our approach provides means for helping an ontology engineer to perform these tasks in a semi-automatic way. The approach is not fully automated since some changes cannot be propagated due the independencies (in the modification) of the included and including ontologies. However, in this case our system offers several possibilities for resolving a particular problem.

In [134] the authors address the problem of guaranteeing the integrity of a modular ontology in the presence of a local change. They propose a strategy for analysing changes and guiding the process of updating compiled information. Ontology modules are connected by conjunctive queries. In order to make local reasoning independent of other modules, the authors use a knowledge compilation approach. The result of each mapping query is computed off-line and added as axiom to the ontology module using that result. Once a query has been compiled, the correctness of reasoning can only be guaranteed as long as the concept hierarchy of the queried ontology module does not change. The authors propose a heuristic change detection mechanism that analyses changes with respect to their impact on the concept hierarchy. The set of changes they consider is not complete while they focus only on changes regarding the concept hierarchy. Since their work is based on the description logic and the ontology module contains a set of objects (i.e. instances) as well, a change regarding an instance might provoke its reclassification, which might result into non-monotonic queries. The main problem is that the definition of the monotonicity given in this paper does not take into account instances. Further, the specification of effects of changes is not complete in the sense that it describes the "worst case" scenario's and that for some changes the effect is "unknown" (i.e. unpredictable).

In [147] the problem of reuse of ontologies is considered and a handcrafted adaptation of an ontology is presented. This work applies one time transformation of the ontology to fit the requirements of the target application. The notion of maintenance is not considered in this work as changes to the specific source ontology are infrequent.

An approach for the engineering distributed, loosely controlled and evolving ontologies is presented in [106]. The authors propose a process template for the harmonisation of the ontologies expanding the same core ontology. It consists of five main activities: build, local adaptation, analysis, revision and local update. The result of the build activity is the initial ontology. However, the whole process is already realised in our ontology evolution system. The result of the build phase is the initial ontology. Since the creation of an ontology from scratch can be considered as the evolution of an empty ontology, it can be performed by applying the single evolution approach. During the local adaptation activity a new ontology as the extension of the initial ontology is created. This phase is covered by replicating the core ontology in the local repository and by extending this replica. While the analysis activity in this process is performed manually, our approach enables a semi-automatic detection of the potential changes that might be introduced in the next version of the core ontology. This is discussed in the next section in details. The revision activity corresponds to the evolution of

the single ontology since in this phase the changes, which are identified in the analysis activity, are applied to the core ontology. The last phase is wrapped by the distributed ontology evolution. Even though the proposed approach gives an excellent methodology for distributed, loosely-controlled and evolving engineering of ontologies, the main disadvantages is that is considers a pretty simple ontology inclusion graph, which consists of one shared ontology and many ontologies that include it. On the contrary, our approach is applicable for an inclusion graph of arbitrary complexity.

In [93] the authors define the transformation set that is used to find a minimal set of changes between two ontology versions. The definition of this set is different from the definition of synchronisation set delta (see section 5.3.2). There are two differences. Firstly, the authors assume that a log does not exist and therefore they try to discover changes by comparing ontologies. Secondly, the authors claim that change can be performed in any order, with one exception: all changes that create new concepts, properties, and instances are performed first. On the other hand, our work is based on the assumption that an evolution log exists and the order in which changes are performed can be discovered by following corresponding properties.

In [100] the authors analysed the ability to make copied (and possibly changed) versions of ontologies up to date with a remotely changed ontology. This might be desirable when an ontology is copied and the original is consecutively changed but it is still necessary to work with the original one, too. This requires that all consecutive changes in the original ontology are carried out in the local copy. This process is called synchronisation. Since the goal of this process is to "update the local vocabulary to obtain the benefits of shared-vocabulary updates, while maintaining local changes that serve local needs", it can be considered as a distributed ontology evolution process. This process is very common in the health-care domain where local hospitals use adapted versions of national or international terminology standards. Keeping the local versions up-to-date with the evolving global version is necessary to stay up-to-date with new insights and in order to be able to exchange information with other users of the vocabulary. The main difference in comparison to our model is that (i) the underlying model is very simple, which results in few changes; (ii) the inclusion graph has low complexity which significantly reduces the problem; (iii) there are no differences in the synchronisation in the centralised systems and in the decentralised systems, etc.

Moreover, research in distributed ontology evolution can also benefit from the research in distributed systems [102]. In [7] the authors describe the techniques that combine push and pull synchronisation in an intelligent and adaptive manner while offering good resiliency and scalability. We extend this approach by taking into account not only the coherency maintenance of the cached data but the maintenance of the dependent and replication consistency as well.

The local update problem in the UMLS Metathesaurus[85] is analysed in [144]. The authors described what they called "the local dilemma", pointing out the problems that developers would face if they enhanced the Metathesaurus locally. They recognised that local enhancements would increase the burden of maintenance when new versions of the Metathesaurus were released, and predicted that the effort required to integrate local enhancements with Metathesaurus updates could easily exceed the effort required to add the

[85] http://www.nlm.nih.gov/pubs/factsheets/umlsmeta.html
The UMLS Metathesaurus is one of three knowledge sources developed and distributed by the National Library of Medicine as part of the Unified Medical Language System (UMLS) project. The Metathesaurus contains information about biomedical concepts and terms from many controlled vocabularies and classifications used in patient records, administrative health data, bibliographic and full-text databases and expert systems.

local enhancements in the first place. In this section we presented an approach for the propagation of changes from the included ontology to all ontologies that include it. This process can be triggered by applying the local enhancement to the included ontology as well. It is possible to extend this approach by discovering useful changes from multiple extensions of the included ontology.

The change propagation problem exists in the evolution of database schema. Although it only considers how schema changes are reflected in the instances, we can compare this work with ours by assuming that dependent or distributed ontologies consist only of instances of entities from included ontology. In order to keep the instances meaningful, either the relevant instances must be coerced into the new definition of the schema or a new version of the schema must be created leaving the old version intact. Three main approaches have been identified and employed in the past [104]. Immediate (conversion) and deferred (lazy, screening) propagate changes to the instances only at different times. Filtering is a solution for versioning that attempts to maintain the semantic differences between versions of schema. Our approach combines the above three methods into a hybrid model. For the evolution between dependent ontologies within the same node we apply the push-based synchronisation which is a variant of immediated change propagation. In the distributed environment, the changes are propagated at explicit request, which implies a deferred approach. Further, by attaching a version number for each ontology and by tracking information about performed changes in an evolution log, the versioning is partially supported.

It is possible to view the dependent distributed ontologies as a collection of autonomous databases [102]. From this perspective, evolution in distributed databases is important. Three types of distributed database systems may be identified. Firstly, distributed database systems with a global schema are produced by selecting the independently developed schemas, resolving semantics and syntactic conflicts among them and creating an integrated schema. The addition of a new schema into the system and/or the modification of the existing schema cause problems since a global schema has to be updated as well. In federated database systems heterogeneous databases interoperable without using a global schema. There is a single distinguished component named federal dictionary. Dependencies between schemas are unknown. Consequently, the update of one schema requires discovery of all schema depending on it. Finally, in multiple databases systems, heterogeneous databases interoperable without using a global schema and a centralised structure. Dependencies between schemas are known and stored in dependencies schemas. The addition or removal of a database schema from a multiple database system as well as the update of any included schema requires the maintenance of a centralised structure.

Regarding the multiple ontology evolution, there is neither global schema nor federal dictionary nor a centralised structure since ontologies aim to be used in the distributed environment such as the Web [102]. The dependencies are explicitly represented on the depending side. The inclusion of a new ontology or the breaking dependencies between ontologies is resolved in a systematic way by taking into account distributed ontology consistency definition. Modification of any included ontologies is resolved by broadcasting changes either automatically after their occurrence (which happens in a centralised environment) or at an explicit request for synchronisation (which takes place in a distributed environment).

5.6 Conclusion

Evolving ontologies that reuse other ontologies is a complex problem. We first considered evolving dependent ontologies within one node, where we focus on ontology inclusion going

beyond the simple cut-and-paste inclusion. We focus on various problems arising from the dependent ontology consistency definition and the push-based synchronisation. Further, we extended this solution to the distributed case. Distributed ontology reuse is supported through controlled ontology replication, which is necessary under present technological constraints, such as available bandwidth. We advocate the pull synchronisation mechanism in order to keep the autonomy of each node in the system. We present an approach for evolution of distributed ontologies, which is based on keeping change information available in the form of evolution logs. The overall approach has been implemented within the KAON framework. The MeSH/MEDLINE evaluation study shows the applicability of the proposed approach.

The future work can be directed towards providing more ways for working with multiple ontologies by taking into account other dependency forms such as ontology mapping, ontology merging, ontology alignment and ontology integration. For example, ontology mapping relates similar (according to some metric) concepts and relations from different sources to each other or ontology merging creates a new ontology from two or more existing ontologies with overlapping parts. Each of these dependency forms puts different requirements on the evolution between dependent ontologies. Some of them can be resolved by introducing a special meta ontology that captures relationships between entities from different ontologies. For example, to set up mapping between ontologies, the mapping ontology might be defined. This ontology should contain the “equal” property that can be used for establishing equivalence between concepts from different ontologies. However, other dependency forms require the extension of our approach by relaxing the constraint that only entire ontologies may be reused. Lifting these constraints will have a significant impact on the evolution of dependent ontologies.

6 Change Discovery

Human knowledge rarely stabilises. New experience always gives new insights, which significantly change old knowledge. More experiences means fewer errors but larger models. This is consistent with the situated cognition phenomena, which can be summarised as follows: using knowledge changes knowledge [81].

This holds for ontologies as well. While a good design may prevent many ontological errors, some problems will not be pop out before an ontology is in use. Therefore, the relationship with the users of an ontology-based system is paramount when trying to develop a useful ontology. Adapting an ontology according to the user's preferences is the unavoidable commitment that ontology engineers must face with and when doing so, preferences and goals of the users behind cannot be neglected.

However, the existing ontology evolution systems completely ignore the possibilities to obtain and to examine the non-explicit but available knowledge about the needs of the end-users. In this section we propose such an approach by analysing various data sources related to the end-users' behaviour which include the information about her likes, dislikes, preferences or the way she behaves. Based on the analysis of this information, an ontology engineer can be suggested to make some changes in the ontology that may yield this ontology better suited for the needs of end-users. In this way we discuss the possibility of continuous ontology improvement by semi-automatic discovery of such changes.

We begin this chapter by introducing the problem. Then we present the conceptual architecture of a system for the usage-driven change discovery. Two scenarios arising from typical interactions of the end-users with the ontology-based portals are discussed. We distinguish between querying and browsing. For each of them we propose a set of measures for identifying problems in an ontology (the so-called problem discovery) as well as a set of rules for resolving these problems (the so-called change generation).

6.1 Problem Definition

An application has to be modified in order to reflect changes in the real world, changes in the user's requirements and drawbacks in the initial design, to incorporate additional functionality or to allow for incremental improvement [72]. Some requests for the adaptation might be specified explicitly such as the need for a new type of a customer due to a change in the business strategy. Other changes are implicit and might be discovered from the usage of this application. For example, if none of users was interested in the information about a product in an on-line catalogue in a longer period of time, then, probably, this product should be excluded from the list of products offered by that application. These "discovered" changes are very important for optimising performance of an application, e.g. by reducing the hierarchy of

the products that has to be browsed. Moreover, they enable the continual adaptation of the application to the implicit changes in the business environment.

However, the usage analysis that leads to the change discovery is a very complex activity, as all methods for learning from data [153]. Firstly, it is difficult to find meaningful usage patterns. For example, is it useful for an application to discover that much more users are interested[86] in the topic "*industrial project*" than in the topic "*research*"? Secondly, when a meaningful usage pattern is found, the open issue is how to translate it into a change that leads to the improvement of the application. For example, how to interpret the information that a lot of users are interested in "*industrial-*" and "*basic-research projects*", but none of them are interested in the third type of the projects – "*applied-research projects*"?

Since in an ontology-based application an ontology serves as a conceptual model of the domain [133], the interpretation of these usage patterns on the level of the ontology alleviates the process of discovering useful changes in the application. For example, the above-mentioned pattern can be treated firstly as useless for discovering changes if there is no relation between the concepts[87] "*industrial project*" and "*research*" in the underlying ontology. Moreover, the structure of an ontology can be used as the background knowledge for generating useful changes. For example, in the case that the "*industrial-*", "*basic-research*" and "*applied-research project*" are three subconcepts of the concept "*Project*" in the domain ontology, in order to tailor the concepts to the users' needs, secondly mentioned pattern could lead to either deleting the "unused" concept "*applied-research project*" or its merging with one of two other concepts (i.e. "*industrial-research*" or "*basic-research*").

However, such an interpretation requires familiarity with the ontology model definition, the ontology itself as well as the experience in modifying the ontologies. Moreover, increasing the complexity of ontologies demands a correspondingly larger human effort for its management. It is clear that the manual effort can be time consuming and error-prone. Finally, this process requires highly skilled personal, which makes it costly.

In this section we present an approach for efficient management of an ontology-based application based on the usage of the underlying ontology-based data. The focal point of the approach is the continual adaptation of the model of the application (i.e. the ontology) to the users' needs. As illustrated above, by analysing the usage data with respect to the ontology, more meaningful changes can be discovered. Moreover, since the content and layout (structure) of an ontology-based application are based on the underlying ontology [133], by changing the ontology according to suit the users' needs, the application itself is tailored to the users' needs.

The basic requirement for such a management system is that it has to be simple, correct and usable for ontology managers[88]. Thus, it must provide capabilities for automatic identification of problems in the usage of the ontology in the underlying application and ranking them according to the importance for a user. When such problems arise, a management system must assist the ontology manager in identifying the sources of the problem, as well as in analysing and defining solutions for resolving them. Finally, the system should help determine the ways for applying the proposed solutions.

This approach is realised as the usage-driven change discovery phase of the ontology evolution process (see section 3.4.3). It concerns the truthfulness of an ontology with respect to its problem domain - does the ontology represent a piece of reality and the users'

[86] The interest in a topic might be measured by the number of queries about the corresponding topic.

[87] A topic is treated as a concept.

[88] An ontology manager is a person responsible for administrating an ontology-based application and does not need to be an experienced ontology engineer.

requirements correctly? Indeed, it helps to find the "weak places" in the ontology regarding to the users' needs, ensures that generated recommendations reflect the users' needs and promotes accountability of managers. In this way the ontology evolution system provides an easy-to-use management system for ontology engineers, domain experts, and business analysts since they are able to use it productively, with minimal training. We present two evaluation studies, which demonstrate the benefits of such a system. As known to the authors, none of the existing ontology management systems offers support for (semi-)automatic ontology improvement in response to the users' needs analysis.

6.2 Conceptual Architecture

IBM's Autonomic Computing initiative [59] attempts to address management tasks by providing IT systems with powerful concepts for self-management including new capabilities for self-healing, self-protecting, self-optimising, and self-configuring. The goal is to reduce the burden associated with the management and the operation of IT systems. Autonomic Computing systems just work, repairing and tuning themselves as needed [122].

Similarly, our goal is to free ontology managers from many of today's evolution tasks. We need a system that is not people-intensive anymore, which would result in decreasing of related management costs. Since autonomic computing systems allow people to concentrate on what they want to accomplish rather than figuring out how to rig the computer systems to get them there, we use the analogy with autonomic computing systems and try to apply their principles to the management of an ontology.

Therefore, our management system is realised according to the *MAPE* (**M**onitor **A**nalyse **P**lan **E**xecute) model [59], which abstracts the management architecture into four common functions: collect data, analyse data, create a plan of action, and execute the plan. Indeed, our architecture decomposes the control loop into four parts:

- *Monitor* – mechanism that collects, organises and filters the data about users' interactions with an ontology-based application;
- *Analyse* – mechanism that aggregates, transforms, correlates, visualises the collected data, and makes proposals for changes in an ontology;
- *Plan* – mechanism to structure actions needed to apply the discovered changes by keeping the consistency of the ontology. The planning mechanism uses evolution strategies (see section 4.2.3) to guide its work;
- *Execute* – mechanism to update the underlying ontology-based application according to the changes applied to the ontology.

By monitoring (**M**) the behaviour of the users and analysing (**A**) this data, by planning (**P**) which actions should be taken, and executing (**E**) them, a kind of a "usage loop" is created.

Figure 62 depicts this "usage loop" in an information portal scenario. A user is searching for information by querying and/or navigating through a portal (cf. 1 in Figure 62). The structure and the content of the portal are based on the domain ontology (cf. 2). All activities the user performed are acquired in the Semantic Log (cf. 3), which is structured according to the Log Ontology (cf. 4), and contains meta-information about the content of visited pages. This log data is aggregated and visualised in the *Usage-Driven Change Discovery module* (cf. 5). Moreover, it helps ontology managers discover changes in the ontology, which are mostly important for enhancing the usability of the application. Since the application of a single ontology change can cause inconsistency in the other part of this ontology and on all the artefacts that depend on it, we applied the ontology evolution process (cf. 6) that guaranties

the transfer of the ontology (cf. 2) and dependent artefacts into another consistent state (see chapter 3). Moreover, the ontology manager can select several evolution strategies (see section 4.2.3) in order to adapt the ontology according to some specificities in her need. Finally, since the underlying application is ontology-based, all changes in the ontology (cf. 2) are reflected in the structure of the portal (cf. 1), thus tailoring the portal to the users' needs, which implicitly arose. For example, if none of users were interested in a topic, then our system would recommend the ontology manager to remove the corresponding concept from the topic hierarchy. Consequently, new users will be not "bored" by browsing topics, which are useless for the domain shown in the portal. In that way, our management system tends to be a user-friendly platform that integrates the results from the analysis of the usage data with the tools that guide the process of modifying the ontology.

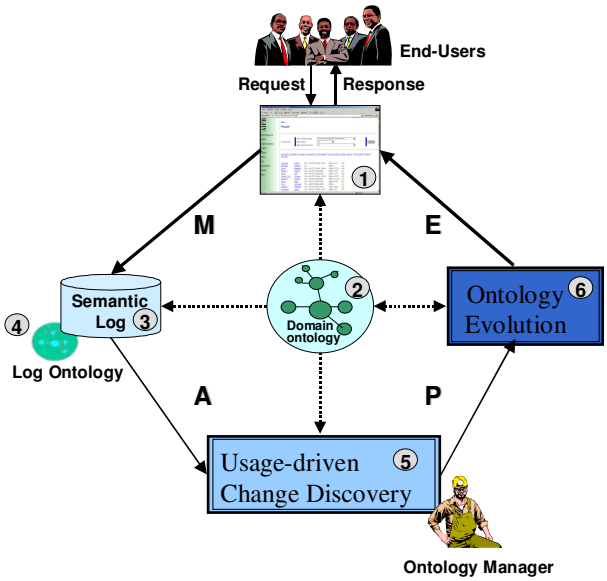


Figure 62. The conceptual architecture of the ontology management system according to the MAPE model

6.2.1 Semantic Log

Traditionally, all the activities of users of an information portal are captured in the standard web server log. However, this log file contains only the information about the address of visited pages, which was not enough for the sophisticated analyses we intended to perform. For example, it is not possible to get information about which query is posted and how many and which results are retrieved. It means that we need the information about the semantics of the visited pages. Moreover, the standard web server log fails in the case of reloading pages from cash and spider's crawling, which cause missing, redundant or incomplete data [132].

In order to resolve these problems we have developed the *Log Ontology* as the backbone for structuring information in a *Semantic Log* [126]. Each user's activity is captured in a *Semantic Log* in the form of instances of concepts of the *Log Ontology*. The *Log Ontology* provides a formal definition of a usage data. It takes into account the semantics of the performed actions. This formal definition of structure and syntax of a log provide more scope for analysis since the prerequisite for the meaningful analysis is that this log information is properly organised and interpreted. Indeed, the structure of the *Log Ontology* is determined in order to enable reasoning about accessibility and usability of the domain knowledge. The additional

advantage is that the existing mechanism for storage and query ontologies can be used for the log as well.

A part of the *Log Ontology*, that is relevant for the rest of this section, is presented in Figure 63a. This ontology models what happens in an application (e.g. in a portal) and why, when, by whom, how it is performed. Each user's activity s represented as an instance of one of the subconcepts of the "*Event"* concept. The structure of the hierarchy of event types reflects the users' activities in an ontology-based application by including all possible types of interactions (e.g. "*Query"*, "*Browse"*, "*Read"*, etc.). Additional information, such as the "*date"* and "*time"* of activity, as well as the identity of the user may be associated through appropriate relations. Information, which provides support for the users' profiling, such as "*sessionID"*, "*clientIP"* etc., may also be included. Entities from the domain ontology are related to instances of the "*Event"* concept through the "*relatedTo"* relation. The dependency between events is represented using the "*previousEvent"* relation.

For each type of event specific properties are defined. For example, an information portal should offer the user an opportunity to find easily relevant information for the topics, which are important for the problem she solves. In other words, the list of retrieved information for a user query should not be empty and should also contain only highly relevant sources. It implies that the semantic log should track the interests of users, as well as the number of answers for the posted queries. Therefore, for the concept "*Query*" two additional properties are defined: "*queryString*" and "*numberOfResults*". Whereas the first property helps in knowing what is preferred by end-users, the second property reflects the total number of related entities. Note that these entities, which represent the query result, are modelled through the property "*relatedTo*".

We note that the presented log file tracks only the interaction, which is related to the user's activity. We assume that only an ontology engineer (manager) has privileges to modify the domain ontology and that the end-users search for information or browse them. Thus, the *Log Ontology* does not contain events regarding the development/evolution of the domain ontology.

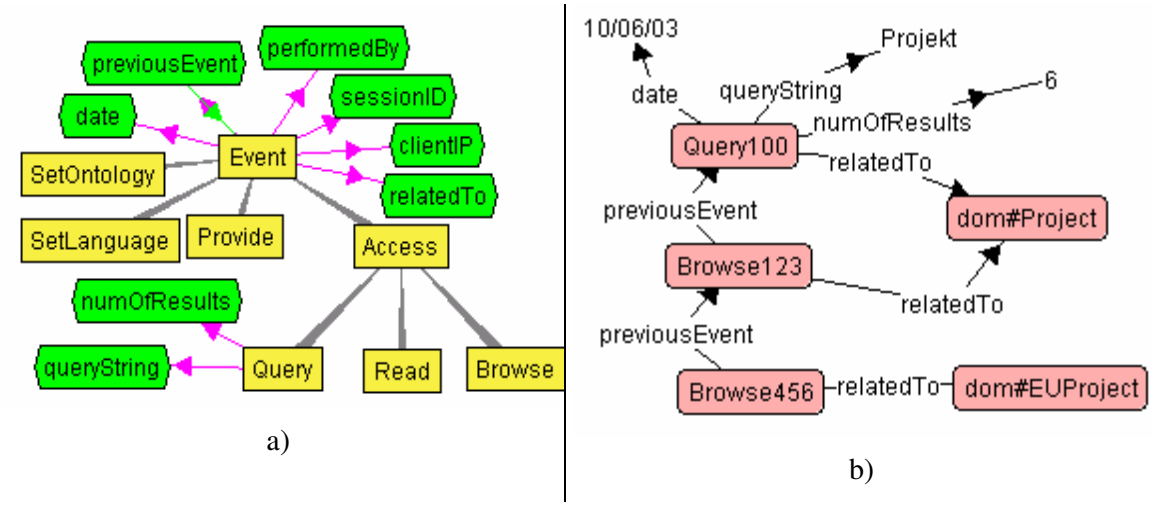


Figure 63. A part of the Log Ontology and the Semantic Log. The conceptual structure of the Log Ontology is represented in the left part. The right part shows several log entries in the form of relation instances.

Figure 63b shows several users' activities stored in the *Semantic Log*. They are the result of user's request for "*Projekt*"[89] and successively browsing activities through the concept "*Project*" and its subconcept "*EUProject*". The instance "*Query100"* captures all-important information regarding the query activity whereas the instances "*Broswe123"* and "*Browse456"* correspond to the navigation activities through the hierarchy of the concept *"Project"*. Note that "dom#" denotes the namespace of the domain ontology.

The *Semantic Log* as all log files might be very large. Although the process of generating is quite simple and straightforward, a log analysis could be a tremendous task that requires enormous computational resources, long time and sophisticated procedures. This often leads to a common situation, when logs are continuously generated and occupy valuable space on storage devices, but nobody uses them and utilises enclosed information. In the rest of this section we elaborate only on the analysis we carried out. The technical realisation of a system is described in section 7.2.3.

6.3 Usage-driven Change Discovery

In this section, we present a novel approach to the ontology evolution that supports the ontology managers in managing and optimising the ontology according to the end-users' needs. The approach incorporates mechanisms for assessing the ontology (and by extension an application based on it) performance with respect to different criteria as well as those enabling us to take actions to optimise it.

One of the key tasks is to check how the ontology fulfils the perceived needs of the end-users. In that way, we obtain an in-depth view of the end-users' perspective on the ontology and the ontology-based application since on the top of this ontology the application is going to be conducted. The technique that can be used to evaluate/estimate the needs of the end-users depends on the information source. By tracking end-users' interactions with the application in a log file, it is possible to collect useful information that can be used to assess what the main interests of the end-users are. In this way, we avoid asking the end-users explicitly since they tend to be reluctant to provide the feedback via filling questionnaires or forms.

Indeed, the approach is based on the analysis of the end-user's interaction with an ontology-based portal. The following assumptions are made:

- a backbone of a portal is a domain ontology (*DO*) (cf. 2 in Figure 62);
- all users' interactions are stored in a semantic log (*SO*) (cf. 3 in Figure 62);
- a semantic log is based on the log ontology (*LO*) (cf. 4 in Figure 62).

We focus on two types of users' activities: querying and browsing. Each query about entities from a domain ontology (*DO*) made by an end-user is stored in a semantic log (*SO*) as an instance of the concept "*Query*" and its property instances. The semantics of this concept is defined in the log ontology (*LO*). Similarly, the end-users' browsing activities through a domain ontology (*DO*) result in an extension of the semantic log (*SO*) with new instances of the concept "*Browse*" defined in the log ontology (*LO*). It is worth mentioning that all instances in a semantic log (*SO*) refer to the domain ontology entities since the end-users were searching for them or browsing. Therefore, a semantic log (*SO*) is a dependent ontology that includes the log ontology (*LO*) as well as the domain ontology (*DO*).

[89] A user can specify a query using German terms (e.g. das Projekt).

The usage-driven change discovery task[90] is split into two subtasks [124]:

- *Discovery of problems* – It focuses on discovering anomalies in the design of an ontology, whose repairing improves the usability of this ontology;
- *Generation of changes* – It focuses on tailoring an ontology to the needs of its users by giving concrete clues on how the ontology should be improved. By making recommendations for the continual improvement of the ontology, the portal based on it is improved as well.

Therefore, the goal of the "*discovery of problems*" task is to find the "weak places" in the ontology, namely those that do not meet the needs of the end-users. To do that, we define several measures, which assess the usability of ontology entities with the respect to the end-users. Thus, these assessment criteria are used to estimate the user's needs.

By later performing the analysis of an ontology by applying the proposed measures on its usage log, it is possible to identify the relevant factors that contribute the success of the ontology from the end-users' point of view. Indeed, the goal of the "*generation of changes*" task is to map the problem, which is found during the "*discovery of problems*" task, into a set of ontology changes that resolve this problem. This is achieved by defining the interpretations of the extreme values[91] of proposed measures in the form of ontology changes. These changes give hints on how the usability of the ontology should be improved.

Indeed, we define the "usable" ontology as an ontology with a structure that conforms to the intuition of end-users accessing this ontology. The intuition of end-users is indirectly reflected in the queries posted by them as well as in the navigation behaviour, as represented in their querying and browsing patterns. By comparing the typical patterns with the ontology itself (which means with the patterns expected by the ontology engineer), it is possible to examine the usability of the ontology and to give concrete suggestions for its improvement.

When we designed this support, we assumed that an ontology update would be only a partially automated process rather than a fully automated process. That is, while certain tasks could be automated, other tasks would have to be supported but not fully automated. For example, we do not want to update an ontology automatically, but rather to make suggestions to an ontology engineer about changes that might be useful. Our experience suggests that this assumption is reasonable. The system generates the recommendations for an update. It is up to an ontology manager to accept or to reject it.

In the rest of this section we elaborate both the types of users' activities in an information portal scenario, namely querying and browsing.

6.3.1 Query-driven Change Discovery

Problem Discovery

A query shows what a user wants and expects to find. Therefore, a query is particularly suited for understanding the user's interests. We define the rate of interest *RateOfInterest(e)* of users for an ontology entity *e* as:

$$RateOfInterest(e) = Frequency(e) * Clarity(e) \quad (1)$$

[90] It is realized in the *Usage-driven Change Discovery* module of the *MAPE* model shown in Figure 62.
[91] The extreme values are values that are greater (smaller) than the given maximum (minimum) threshold value.

Frequency(e) represents the users' interest in an ontology entity *e*, and it is calculated as a ratio between the numbers of the users' interactions with the system related to the ontology entity *e* and the total number of the interactions. Indeed, we use the formula:

$$Frequency(e) = \frac{Q(e)}{Q},\ Frequency \geq 0,$$

whereas *Q(e)* is the number of queries that contains an entity *e*, and *Q* is the total number of queries. *Q* and *Q(e)* are calculated as:

$$Q(e) = |\{ i_1 | i_1 \in instconc("Query") \wedge$$

$$(e \in instprop("relatedTo", i_1) \vee (i_2 \in instprop("relatedTo", i_1) \wedge i_2 \in instconc(e))\}|$$

$$Q = |instconc("Query")|.$$

"*Query*" and "*relatedTo*" are entities from *the Log Ontology* (see section 6.2.1) and the functions *instconc* and *instprop* are already defined in section 2.2 (see Definition 4). Note that the frequency factor is obtained by considering a semantic log (SO).

To estimate the interest of the end-user it is not sufficient to consider only her activities. The structure of an ontology has to be taken into account as well. We model this through the clarity factor. The clarity factor represents the uncertainty to determine the user's interest in a posted query. For example, when a user makes a query using a concept "*Person*", which contains two subconcepts "*AcademicStaff*" and "*Student*", it could be a matter of discussion: whether she is interested in the concept "*Person*" or in its subconcepts, but she failed to express it in a clear manner. Our experiences show that users who are not familiar with the given ontology used to use a more general concept in searching for information, instead of using more specific concepts. In other words, the clarity factor makes the calculation of the users' interest more sensitive to the structure of the ontology by accounting possible "errors" in the query formulation.

The formula for the clarity factor depends on the entity type:

$$Clarity(e) = \begin{cases} k(e) * \frac{1}{numSubConcepts(e) + 1}, & e \in C \\ k(e) * \frac{1}{numSubProperties(e) + 1} * \frac{1}{numDomains(e)}, & e \in P \end{cases}, Clarity > 0,$$

whereas *numSubConcepts(e)* is the number of subconcepts of a concept *e* ($e \in C$), *numSubProperties(e)* is the number of subproperties of a property *e* ($e \in P$) and *numDomains(e)* is the number of domains defined for the property *e*. The following formulas are used:

$$numSubConcepts(e) = |\{x | (x, e) \in Hc\}|$$

$$numSubProperties(e) = |\{x | (x, e) \in Hp\}|$$

$$numDomains(e) = |\{x | x \in domain(e)\}|$$

We note that for the estimation of the clarity factor only a domain ontology (DO) is considered.

The coefficient k is introduced in order to favour the frequency of the usage. It is calculated using the following formula:

$$k(e) = numLevel(e) + 1$$

where *numLevel(e)* is the depth of the hierarchy of the entity e.

Our primary goal is to decrease the impact of the non-leaf concepts since they represent the common view of the set of their subconcepts, as described above. A similar strategy is applied to the properties and their hierarchy. However, the unclearness of reasons for a property usage can also arise when multiple domains for a property are defined. Thus, in order to clarify the context of a property usage, we require the explicit specification of the domain of that property, or otherwise we decrease its clarity factor.

The previous formulas take into consideration only one entity. The recent analyses show that web users typically submit very short queries to search engines and the average length of web queries is less than two "terms" [151]. For an enterprise portal we suppose making of queries that contain 3-4 searching topics. Therefore, it is required to extend the calculation of the *RateOfInterest* factor. For the simultaneous analysis of the set of entities, the modifications of the frequency of interest *Frequency(e1,...,en)* is straightforward. However, the calculation of the clarity factor requires further analysis. Here we mention the most frequently occurring cases:

$$Clarity(c1,c2) = \begin{cases} \dfrac{Clarity(c1) * Clarity(c2)}{numProperties(c1,c2)+1}, & numProperties(c1,c2) \geq 1 \\ 1, & otherwise \end{cases}$$

$$Clarity(c1,c2,...,cn) = \prod_{1 \leq i,j \leq |C|} Clarity(ci,cj)$$

$$Clarity(c,p) = \begin{cases} Clarity(c) * Clarity(p) * numDomains(p), & c \in domain(p) \\ 1, & otherwise \end{cases}$$

where $c_1,c_2,\ldots,c_n$ are concepts (i.e. $c_1,c_2,\ldots,c_n \in C$) and p is a property ($p \in P$). The number of properties between two concepts c_1 and c_2 denoted by *numProperties* is calculated using the following formula:

$$numProperties(c1,c2) = \left| \{ p \mid c1 \in domain(p) \wedge c2 \in range(p) \} \right|$$

Note that the values of all clarity factors are between 0 and 1.

Change Generation

The value for the *RateOfInterest* is calculated for all entities, and two extreme cases are analysed: the frequently used and unused entities. The first extreme corresponds to the entities with the highest rates that should be considered for changes. The formula (1) expresses our experience that the frequent usage of an entity in queries can be a consequence of the bad modelling of the hierarchy of that entity, i.e. in modelling that entity, the hierarchy is not explored in details. For example, in the project domain it may happen that the concept "*Project*" is not split into concepts "*Research Project*" and "*Industrial Project*", although there are a lot of differences between them (e.g. goals, payment, etc.). Consequently, any time the user wants to find information related to either the research or industrial projects, she has to make a query with the concept "*Project*", which results in the huge number of answers. Therefore, our analysis can suggest that the concept "*Project*" should be divided into several subconcepts. An ontology engineer decides whether and how to do that. If the considered concept already has a hierarchy, then its suitability (probability) for a change is decreased by

the clarity factor. This is shown in Figure 64. Even though the frequencies of querying for the concept "*Person*" and the concept "*Project*" are the same, our system suggests only splitting of the concept "*Project*" into a concept hierarchy since the hierarchy for the concept "*Person*" is already defined. A similar strategy can be applied to the properties as well.

In the case that nobody is interested in an entity, i.e. the rate of interest for that entity is equal 0, then the entity should be considered for deleting from the ontology and consequently from annotations. However, the problem arises when there are a lot of resources annotated with that entity. It can be interpreted in various ways, including that the topic is interesting for the community, but not used in past projects, or that employees are very familiar with this topic, etc.

Returning to the example shown in the left part of Figure 64, the usage-driven change discovery would propose the following changes based on the analysis of extreme values of the rate of the interest factor that is shown in the right part:

- since the rate of the interest of the concept "*Project*" is high, it would be useful to split it into subconcepts;
- since the rate of the interest of the concept "*PostDoc*" is low, it would be useful to remove it.

Taking into consideration a set of entities can extend the previous analysis. Again, we interpret the extreme values. The high value of the rate of interest for the set of entities indicates the following changes. If two concepts frequently occur in queries, it means that the users are very interested in the relationship between them. If a property between them does not exist, then the system makes recommendation to create a new property and to set one of the concepts as a domain and the other concept as a range of this property. This recommendation is highly ranked since the clarity factor is 1. In case that such property already exists, it is possible that it is too general, i.e. defined for the concepts on the higher level in the concept hierarchy. This is a frequently occurring example of the inadequate ontology modelling. The problem can be resolved in two ways. One recommendation would be to specialise this property by creating a subproperty and by defining considered concepts as the domain/range concepts of the newly introduced property. Other solution is to adapt the domain and the range of the existing property to the right level of abstraction by taking into account the instantiation of this property.

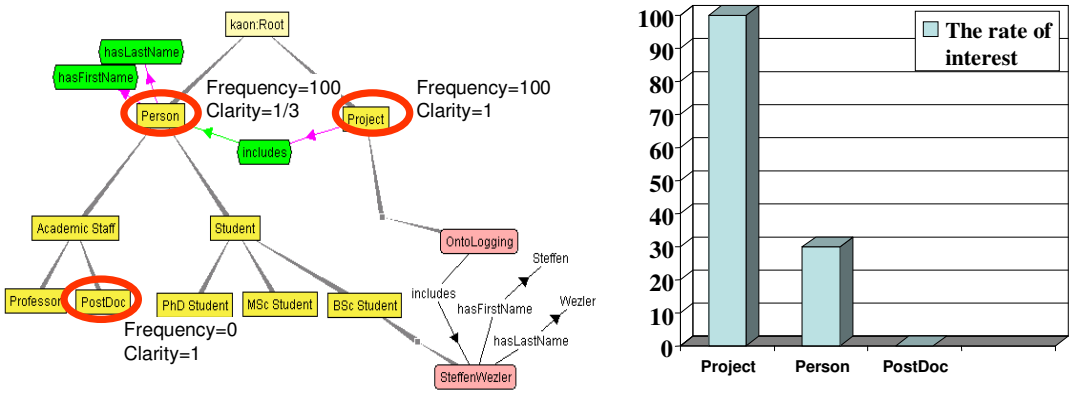


Figure 64. Change discovery from querying

Similar to the previous discussion, the frequent occurrence of *n* ontology entities simultaneously indicates that these concepts are related. However, since the ontology properties are binary, a set of recommendations is delivered. First, a new concept should be created. Second, this concept should be related to each of the *n* frequently occurring concepts through the newly created properties. At the end, the annotations (see section 5.4.3) should be extended in order to satisfy the "quality" criteria mentioned in the section 3.4.2.

A high number of queries related to a concept and a property indicate the importance of the concept for that property. Consequently, if the concept is neither a domain nor a range of that property, it will be recommended to attach the property to the concept. Otherwise, we consider the possibility to specialise the property.

Evaluation

The Semantic Portal (SEAL) [133] is an ontology-based application, which provides a "single-click" access to almost all information related to the organisation, people, research studies and projects of an institute. It is widely used by our research and administrative staff as well as by our students. One of the most usable features is the possibility to search for people, research areas and projects on the semantic basis, i.e. by using the corresponding *Institute Ontology*. The portal provides a very user-friendly interface, which enables formation of arbitrary queries using entities from the underlying ontology. The search is performed as an inference through metadata, which is crawled from the portal pages.

Since the installation of the new version of the portal, the information about users' activities, regarding querying and browsing the portal, are logged in a file. The primary goal was to test the stability of the used version of inference engine. However, we reused the log file in order to evaluate our methods for discovering changes in the ontology. We set up a "what-if" experiment concerning this log file as follows:

1. We rewrote 1000 randomly selected queries under the following hypothetical conditions:

   (a) The hierarchy of the concept "*Person*" that originally had five levels is shortened to only one level including the subconcepts "*Researcher*" and "*Student*";

   (b) The hierarchy of the concept "*Project*" that originally had two levels is deleted;

   (c) The hierarchy of the concept "*Research Area*" is shortened to the first level only. Consequently, we use 20 subconcepts (e.g. "*Knowledge Based Systems*", "*E-Commerce*", etc.) instead of 80 subconcepts in the original hierarchy.

   The hypothetical conditions given above are used for query rewriting. For example, from the original query in the form of ("*Professor*", "*pastProject*", "*Knowledge Acquisition*"), meaning that a user is interested in information about professors whose past project was related to the knowledge acquisition, one gets the rewritten query in the form ("*Researcher*", "*Project*", "*Knowledge Based Systems*")[92].

2. We started searching (i.e. inferencing) using these queries.

3. We calculated the *RateOfInterest* factor for concepts "*Person*", "*Researcher*", "*Project*" and research areas "*Knowledge Based System*" (KBS) and "*E-Commerce*". In order to simplify the analysis, for the coefficient k we used the value 1.

[92] It means that in the original hierarchy the concept "*Professor*" is the subconcept of the concept "*Researcher*" and the set of all subconcepts of the concept "*Knowledge Based System*" includes the concept "*Knowledge Acquisition*".

Table 10 shows the result of our analysis.

Discussion:

We made a hypothetical situation in which the ontology is badly modelled and some hierarchies are not explored at all. A user can select only some restricted, higher-level concepts and for each specialisation she has to use one of higher-level concepts (e.g. for the query about professors she has to use the concept "*Researcher*"). In such a way we modelled the situation in which the underlying ontology did not correspond to the users' needs. The task of our method was to recognise which of badly modelled hierarchies do not reflect users' needs. We discuss several results:

- The concept "*Researcher*" has the highest *RateOfInterest* - it should be considered firstly.

  This is the right decision while a lot of queries contain the concept "*Researcher*" and it has no hierarchy in the hypothetical situation. It means that we could conclude that the concept "*Researcher*" is used as a replacement for the users' need to search for some specialisations of researchers.

- The concept "*Knowledge Based Systems*" should be considered before the concept "*E-Commerce*".

  In our experiment both hierarchies are shortened. However, in the original ontology the first one was larger and therefore should be firstly considered for a change. The number of queries, which contain topic "knowledge-based system", reflects users' needs for more specialised areas of the knowledge-based system.

- The concept "*Person*" has the lowest *RateOfInterest*.

  This is the right estimation, since the concept "*Person*" has one level of the hierarchy, which satisfies the users' needs regarding this concept.

Table 10. The result of the analysis of the rate of the interest

Measure/Concept	Researcher	Project	KBS	E-Commerce	Person
Frequency	202/1000	100/1000	10/1000	2/1000	6/1000
Clarity	1	1	1	1	1/3
RateOfInterest	0,202	0,1	0,01	0,002	0,002

6.3.2 Browsing-driven Change Discovery

Problem Discovery

The main problem we faced in developing ontologies is the creation of a hierarchy of concepts since a hierarchy, depending on the users' needs, can be defined from various points of view and on the different levels of the granularity. For example, by considering gender of a person, the concept "*Person*" can be split into two subconcepts: "*Male*" and "*Female*". However, by considering the person's occupations, the right decision in the university domain would be to specialise the concept "*Person*" into concepts "*Academic Staff*" and "*Student*". Further, the concept "*Student*" can be further decomposed into concepts "*BSc Student*", "*MSc Student*" and "*PhD Student*". Each of these subconcepts can be in turn split into new subconcepts and so on. For some purposes the coarse hierarchy is useful, for others more

details are needed. For example, one might group the concepts "*MSc Student*" and "*PhD Student*" into the concept "*Graduated Student*".

It is clear that the initial hierarchy has to be pruned in order to fulfil the user's needs. Moreover, the users' needs can change over time, and the hierarchy should reflect such a migration. The usage of the hierarchy is the best way to estimate how a hierarchy corresponds to the needs of the users. Consider the example shown in Figure 65.

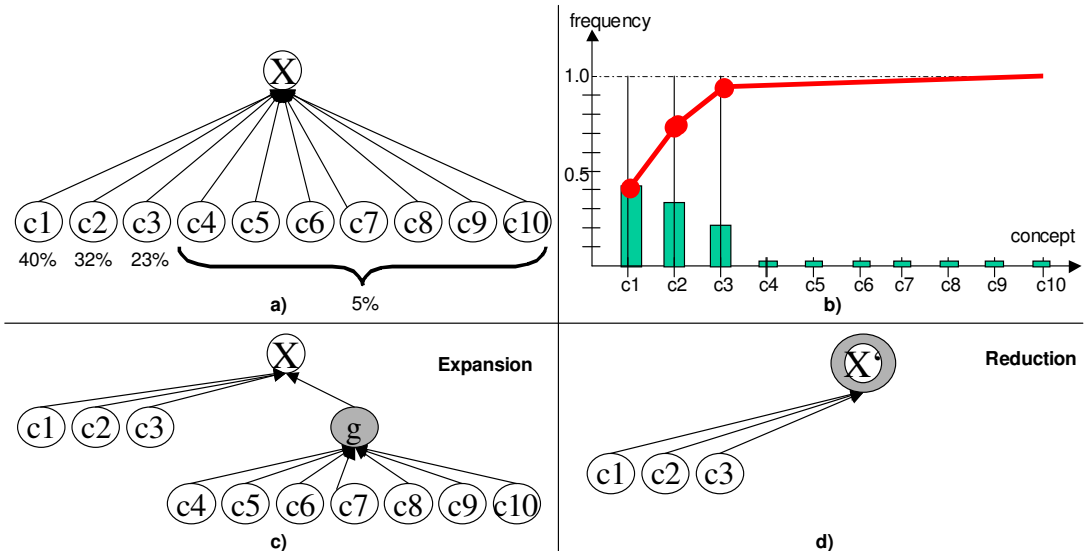


Figure 65. An example of the non-uniformity in the usage of the children. (a) the problem; (b) the Pareto diagram of the problem; (c) the resulting ontology after its extension and (d) the resulting ontology after its reduction.

Let us assume that in the initial hierarchy, the concept "*X*" has ten subconcepts (c_1, c_2,...,c_{10}), i.e. an ontology manager has found that these ten concepts correspond to the users' needs in the best way. However, the usage of this hierarchy in a longer period of time showed that about 95% of the users are interested in just three subconcepts (i.e. 95=40+32+23) out of these ten. It means that 95% of the users obtain 70% (i.e. 7 of 10 subconcepts) useless information via browsing this hierarchy since they find seven subconcepts irrelevant. Consequently, these 95% of the users invest more time in performing a task than needed since irrelevant information can draw their attention away. Moreover, there is a bigger chance to make an accidental error (e.g. an accidental click on the wrong link) since the probability of selecting irrelevant information is greater.

In order to make this hierarchy more suitable to the users' needs, two ways of "restructuring" the initial hierarchy would be useful:

- *expansion* – to put down in the hierarchy all seven "irrelevant" subconcepts, while grouping them into a new subconcept g (see in Figure 65c);
- *reduction* – to remove all seven "irrelevant" concepts, while redistributing their instances into remaining subconcepts or the parent concept (see in Figure 65d).

Through the expansion, the needs of the 5% of the users are preserved by the newly introduced concept and the remaining 95% of the users benefit from the more compact structure. By the reduction, the new structure corresponds completely to the needs of 95% of the users. The needs of 5% of the users are implicitly satisfied. Moreover, the usability of the ontology increased since the instances which were hidden in the „irrelevant" subconcepts are

now visible for additional 95% of the users. Consequently, these users might find them useful, although in the initial classification they are a priori considered as irrelevant (i.e. these instances were not considered at all). Note that the Pareto diagram[93] shown in Figure 65b enables the automatic discovery of the minimal subset of the subconcepts which covers the needs of most of the users.

The problem of post-pruning a hierarchy in order to increase its usability is explored in the research related to modelling the user interface. The past work [8] showed the importance of a balanced hierarchy for the efficient search through hierarchies of menus. Indeed, even though the generally accepted guidelines for the menu design favour breadth over depth [60], the problem with the breadth hierarchy in large-scale systems is that the number of items at each level may be overwhelming. Hence, a depth hierarchy that limits the number of items at each level may be more effective. This is the so-called breadth/depth trade-off [92].

Moreover, organising unstructured business data in useful hierarchies has recently got more attention in the industry. Although there are some methods for an automatic hierarchy generation, such a hierarchy has to be manually pruned in order to ensure the usability of the hierarchy. The main criterion is the "coherence" of the hierarchy [112], which represents some kind of the uniform distribution of resources (e.g. documents) in all parts of the hierarchy. It ensures that the hierarchy is closely tailored to the needs of the intended users.

In the rest of this subsection, we describe how to fine-tune a hierarchy according to the users' needs. Since we consider the concept hierarchy as a graph, we use the terms "node" and "link" as synonyms for a concept and a *direct hierarchy*[94] relation (see H_C in Definition 3), respectively. Moreover, the *parent* (i.e. superconcept) and the *child* (i.e. subconcept) of a *direct hierarchy* relation correspond to the source and the destination node of a link in a graph.

From the structural point of view, there are four basic anomalies that can be accounted in a hierarchy:

(i) a node is missing;

(ii) a node is not necessary;

(iii) a link is missing;

(iv) a link is not necessary.

All other anomalies (e.g. a wrong position of a node in a hierarchy) can be described as a composition of four basic ones. These anomalies correspond to the four basic changes, which an ontology manager can perform on a hierarchy: (1) adding a concept, (2) deleting an existing concept, (3) adding a direct hierarchy relation and (4) deleting a direct hierarchy relation.

From a user's point of view, two problems can arise while using a hierarchy:

- *Problem1*: too many outgoing links from a node;
- *Problem2*: too few outgoing links from a node.

Regarding the first problem, the user has to check/consider too many irrelevant links for her information needs in order to find the most relevant link she should follow. For the explanation about possibilities to resolve this problem see Figure 65. Regarding the second

[93] According to the Pareto principle, by analysing 20% of most frequently used data, 80% of the problems in the ontology can be eliminated.

[94] A set of the direct hierarchy relations is obtained by excluding the transitive closure from a concept hierarchy. It means that only H_C is considered, but not H_C^*.

problem, the user misses some relevant links required to fulfil her information need. Consequently, some new links, which correspond to the users' need, have to be added. Another solution for resolving that problem can be the addition of some new nodes and their linkage to the given node.

However, the discovery of places in the hierarchy, which correspond to the *Problem1* or the *Problem2*, is difficult since we do not expect the explicit feedback from users about the usefulness/relevance of some links. On the other hand, it is difficult to define automatically an "optimal" number of outgoing links for a node since from the user's point of view it is possible that two nodes in a hierarchy have quite different "optimal" number of outgoing links (e.g. 3 vs. 10). A criterion to define this optimality is the usage of these links. Our approach tries to discover such links' usefulness by analysing the users' behaviours on these links, i.e. by using the so-called users' implicit relevance feedback [115]. Additionally, the usage of a link has to be considered in the context of the neighbouring links (i.e. the links which have a common node with the given link). Indeed, the consideration of a link in isolation (from the neighbouring links) can lead to the discovery of "wrong" changes. For example, based on the information that a link is visited 1000 times (which is more than the average visiting), one can imply the need to split the information conveyed through this link, i.e. to split the **destination** node of that link into several nodes. However, in the case that all of its sibling-links (links which have the same source node) are uniformly visited, the "right" change would be to split the **source** node by introducing an additional layer of nodes between it and all the existing destination nodes. The interpretation is that the source node models too many users' needs and has to be refined through more levels of granularity. The process of analysing the usage of a link in the context of its neighbouring' links is called *the discovery of problems*, (see section 6.3).

Another difficulty is the mapping of the resolutions of the *Problem1* and the *Problem2* into the set of elementary changes (1)-(4) since a problem in a hierarchy can be resolved in different ways. This process is called *the generation of changes*. It depends on the specificity of the discovered problem and the intention of the ontology manager. She can always choose between the possibilities to reduce or to expand the hierarchy, based on her need, as described in the previous section (see Figure 65).

To cope with these two issues (*the discovery of problems* and *the generation of changes*), we use two basic measures obtained from the *Semantic Log* (see section 6.2.1):

- *Usage(p, c)* - the number of browsing the link between nodes (concepts) *p* and *c*, where the concept *c* is a subconcept of a concept *p* i.e. $(c,p) \in H_C$.
- *Querying(n)* - the number of querying for the node *n*.

Note that in order to avoid the division with 0, we set up the default values for *Usage(p,c)* and *Querying(n)* to 1.

Additionally, we define four measures for estimating the uniformity (balance) of the usage of a link regarding the link's neighbourhood:

- $SiblingUniformity(p,c) = \frac{Usage(p,c)}{\sum_{\forall x(x,p) \in Hc} Usage(p,x)}$
- $ChildrenUniformity(p) = \frac{\sum_{\forall x(x,p) \in Hc} Usage(p,x)}{\sum_{\forall x(p,x) \in Hc} Usage(x,p)}$

- $ParentUniformity(p,c) = \frac{Usage(p,c)}{\sum_{\forall x(c,x)\in Hc} Usage(x,c)}$
- $UpDownUniformity(p,c) = \frac{Usage(p,c)}{Usage(c,p)}$

$0<SiblingUniformity,\ ChildrenUniformity,\ ParentUniformity\leq 1,\ UpDownUniformity>0$, $(c,p)\in H_C$ and Hc is a set of direct hierarchy relations (see Definition 3).

SiblingUniformity represents the ratio between the usage of a link and the usage of all links, which have the common source node with that link (the so-called sibling links). *ChildrenUniformity* stands for the ratio between the sum of the usage of all the links whose source node is the given node and the sum of the usage of a node through all incoming links into this node[95]. The ratio between the usage of a link and the usage of all links which have the common destination node with that link is called *ParentUniformity*. Finally, *UpDownUniformity* characterises the ratio between the usage of a link in two opposite directions, i.e. in browsing down and browsing up through a hierarchy.

Returning to the example shown in Figure 65, the *SiblingUniformity* between the concepts *x* and the concept *c1* is $SiblingUniformity(x,c_1)=40\%$.

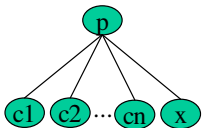
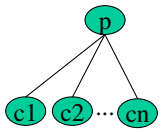
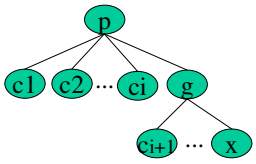
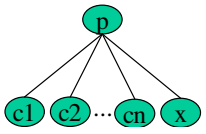
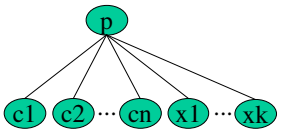
Change Generation

The measures defined in previous section are used to assess balance in the usage of a concept hierarchy. The extreme values of these measures indicate the existence of a problem in the hierarchy, i.e. they are used for *the discovery of problems*. The interpretation of these extremes with respects to the users' needs and according to the intention of an ontology manager (to expand or to reduce the hierarchy) leads to *the generation of changes*. Table 11 shows the summary of typical problems in using a hierarchy, discovered and resolved by our approach. The threshold values for all parameters are set either automatically (i.e. statistically) or manually by the ontology manager.

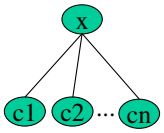
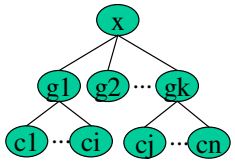
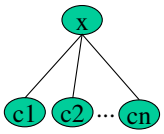
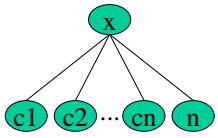
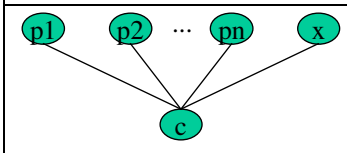
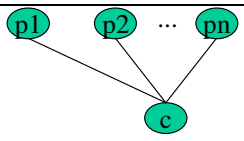
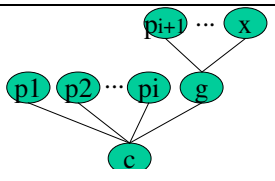
We here give the interpretation of the first case shown in Table 11 only, i.e. *SiblingUniformity*. The reduction is done by deleting a link that is very rarely browsed since we assume that the frequency of usage is related to the relevance of that link for the users. By deleting such a link, we enable the users to focus upon relevant links only. Note that the removal of a link does not necessary cause the removal of the destination node. It is up to the ontology evolution system to decide about it with respect to the consistency of the ontology (see chapter 4). On the other hand, if the ontology manager wants to make this hierarchy more suitable for users by keeping the rarely browsed links, then she has to expand the hierarchy by introducing a new node that groups all less relevant links. In that way, all links are kept, but the users have the focus on the most important ones only. However, this expansion (as all others) requires more efforts of ontology managers since they have to define the meaning of the new node and to select nodes to be grouped. Similar interpretations can be done for all other cases. Note that $0<ChildrenUniformity(p)\leq 1$, where the value 1 corresponds to the "ideal" browsing down a node (i.e. all arrivals in a node are continued down the hierarchy). Therefore, the maximal value of the *ChildrenUniformity* is not considered for the problem discovery.

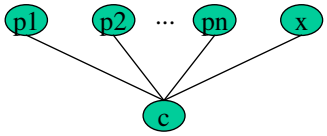
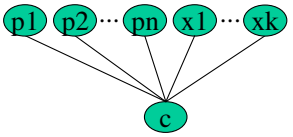
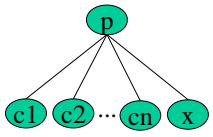
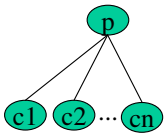
[95] Note that multiple inheritance is allowed. Consequently, a concept can have more than one *parent*.

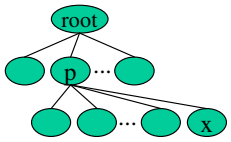
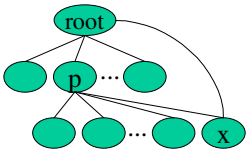
Table 11. The interpretation of the extreme values of the proposed measures

<table>
<tr><td colspan="4">1. SiblingUniformity (SU)</td></tr>
<tr><td colspan="4">Discovery of problems</td></tr>
<tr><td>Problem[96]</td><td colspan="2">Value</td><td>Interpretation</td></tr>
<tr><td>Problem1</td><td colspan="2">below the threshold</td><td>If the SU(p,x) is low then the link p-x might be irrelevant.</td></tr>
<tr><td colspan="4">Generation of changes</td></tr>
<tr><td>Example</td><td colspan="2">Reduction</td><td>Extension</td></tr>
<tr><td> 
</td><td colspan="2"> 
</td><td> 
</td></tr>
<tr><td>Users, who browse down the node p, use the link p–x very rarely.</td><td colspan="2">The link p-x is deleted.</td><td>The set of destination nodes with low SU, c_{i+1}, ...,x, is grouped in newly added node g.</td></tr>
<tr><td colspan="4">2. SiblingUniformity (SU)</td></tr>
<tr><td colspan="4">Discovery of problems</td></tr>
<tr><td>Problem</td><td colspan="2">Value</td><td>Interpretation</td></tr>
<tr><td>Problem2</td><td colspan="2">above the threshold</td><td>If the SU(p,x) is huge then the link p-x might cover many different users' needs.</td></tr>
<tr><td colspan="4">Generation of changes</td></tr>
<tr><td>Example</td><td colspan="2">Reduction</td><td>Extension</td></tr>
<tr><td> 
</td><td colspan="2">-</td><td> 
</td></tr>
<tr><td>Users, who browse down the node p, use the link p–x very often.</td><td colspan="2">Since this link conveys much relevant information, it has to be kept.</td><td>The destination node x is split in several new nodes. The special case is when it is split into so many nodes as it has child-nodes. In that case the node is replaced with its child-nodes.</td></tr>
<tr><td colspan="4">3. ChildrenUniformity (CU)</td></tr>
<tr><td colspan="4">Discovery of problems</td></tr>
<tr><td>Problem</td><td colspan="2">Value</td><td>Interpretation</td></tr>
<tr><td>Problem1</td><td colspan="2">below the threshold</td><td>If the CU(x) is low then all outgoing links</td></tr>
</table>

[96] This column corresponds to the type of problems in using a hierarchy from the user's point of view (i.e. *Problem1* and *Problem2*).

		from the node x might be irrelevant.
Generation of changes		
Example	Reduction	Extension
		
Users, who "visit" the node x, browse down the node x very rarely.	All outgoing links from x are deleted.	The new layer of nodes is introduced by grouping the child-nodes according to a criterion defined by the ontology manager.
4. ChildrenUniformity (CU)		
Discovery of problems		
Problem	Value	Interpretation
Problem2	below the threshold	If the CU(x) is low then it seems that some relevant outgoing links from x are missing.
Generation of changes		
Example	Reduction	Extension
	-	
A lot of users stop the browsing in the node x. Probably they miss some relevant links to browse further.	-	A new node n is introduced and connected to the node x.
5. ParentUniformity (PU)		
Discovery of problems		
Problem	Value	Interpretation
Problem1	below the threshold	If the PU(x,c) is low then the link x-c might be irrelevant.
Generation of changes		
Example	Reduction	Extension
		

The link x-c is used very rarely.	The link x-c is deleted.	The set of source nodes with low PU, p_{i+1}, ..., x, is grouped in a newly added node g.
6. ParentUniformity (PU)		
Discovery of problems		
Problem	Value	Interpretation
Problem2	above the threshold	If the PU(x, c) is huge then the link x-c might cover many different users' needs.
Generation of changes		
Example	Reduction	Extension
	-	
The link x-p is used very often.	Since this link conveys much relevant information, it has to be kept.	The source node is split into several new nodes.
7. UpDownUniformity (UD)		
Discovery of problems		
Problem	Value	Interpretation
Problem1	above the threshold	If the UD(p, x) is huge then the node x might be inappropriate since almost all visits to x finish by browsing back to p.
Generation of changes		
Example	Reduction	Extension
		-
Almost every user who browsed down the link p-x, browsed afterwards back (up) that link.	The link p-x is deleted.	It is possible that the node x is proper, but its child-nodes are inappropriate – that is the reason to browse back. It is up to ontology manager to perform such an analysis.
8. Querying(Q)		
Discovery of problems		
Problem	Value	Interpretation
Problem1	above the threshold	If the Q(x) is huge then the node x might be very relevant as a starting point in browsing.

Generation of changes		
Example	Reduction	Extension
	-	
A lot of users made the query about the content of the node x.	-	A link between the root of the hierarchy and the node x is added.

The process of discovering changes can be performed in two ways: either the system recommends the "problematic" parts of the hierarchy automatically, or the ontology manager selects a part of the ontology, which she wants to update, and interprets the presented parameters on her own. In the automatic discovery, the threshold value can be tuned according to the needs of the ontology manager.

Table 12 shows the dependencies between elementary ontology changes related to a hierarchy with the extreme values of the proposed measures used for the discovery of changes. Note that grouping or the splitting of a node or a link represents a composite ontology change that is realised as a sequence of elementary ontology changes, i.e. (1)-(4). We show only the requested changes. However, the ontology evolution system can induce additional changes in order to keep the consistency of the ontology.

Table 12. Dependency between the discovery of problems (columns) and the generation of changes (rows)

	min SU		**max SU**		**min CU**		**min PU**		**max PU**		**max UD**		**max Q**	
	R[97]	E[98]	R	E	R	E	R	E	R	E	R	E	R	E
AddConcept		x		x		x		x		x				
RemoveConcept											x			
AddSubConcept		x		x		x		x		x				x
RemoveSubConcept	x				x		X				x			

The proposed measures (*SiblingUniformity, ChildrenUniformity, ParentUniformity, UpDownUniformity*) can be seen as an extension of the measures defined in the previous section. These measures consider only the querying activities, i.e. they are based on the frequency of the querying for the ontology entities. They take into account the usage of each ontology entity in isolation, i.e. independently of the usage of other ontology concepts. The information about the ontology structure (e.g. whether a concept is a leaf in a concept hierarchy or has subconcepts) is captured very roughly through the *clarity factor*. Here, we take into account the browsing activities as well. Since the browsing is related to a concept hierarchy, the proposed measures consider the concept in the context that is defined through its neighbourhood (i.e. its parents, children and siblings).

[97] R stands for reduction
[98] E stands for expansion

Evaluation

As a test bed for the presented research, we use the VISION portal (see section 4.4.1), a semantics-driven portal that allows browsing and querying of the state-of-the-art information (people, projects, software etc.) related to the knowledge management. The backbone of the system is the *Vision Ontology*. All users' interactions with the portal are tracked into its *Semantic Log*. We analysed the log data captured in the period from December 2002 to May 2003.

In the *Vision Ontology* there are several *direct hierarchies* (the hierarchy of organisations, projects, etc.), as well as a non-taxonomic hierarchy "*hasTopic*", expressing the hierarchy of research areas. Since this "*hasTopic*" hierarchy is huge (the number of the concepts: 139, the average depth: 5, the average number of direct hierarchy of a concept: 3) and very often browsed, it is very suitable for the evaluation of our approach.

We made some changes in the "*hasTopic*" hierarchy, which corrupt the uniformity of its usage, expecting from users to reverse them by using the proposed approach. Indeed, we induced 35 changes in the hierarchy, as follows:

C1. the addition of five new concepts and their linkage into the hierarchy with a low usage;
C2. the addition of five "*hasTopic*" relations, with a low usage, between existing concepts;
C3. the extension of three leaf concepts (i.e. concepts without children) with three new subconcepts for each leaf concept;
C4. the merging of two children into a concept at three different positions in the concept hierarchy;
C5. the deletion of five "*hasTopic*" relations.

Four subjects who had little experience in ontology editing by using traditional ontology editors participated in the experiment. The task was to improve the "corrupted" ontology by using our system, i.e. to refine the given ontology by balancing its usage.

We measured the correctness of the final hierarchy and the time spent in the modification process. The "gold standard" was the initial ontology, i.e. they should discover all the corruption we made in the ontology. Since there are no available tools for the usage-based ontology management, we evaluate only the usability of our tool, without comparing it to other methods. Moreover, the manual discovery and resolving of the implied changes was impossible, due to a huge number of entries in the *Semantic Log* of the portal and the complexity of the "*hasTopic*" hierarchy. The results of the evaluation are presented in the Table 13. Note that the correctness is defined as the ratio between the number of discovered changes and the number of corruptions in the initial hierarchy.

Table 13. The result of the evaluation

Corruption	Useful parameter for the change discovery	Correctness (total for all four users)	Time (average time for a recovery)
C1	min *SiblingUniformity*	15/20	120 sec
C2	min *ParentUniformity*	18/20	90 sec
C3	min *ChildrenUniformity*	11/12	50 sec
C4	max *SiblingUniformity*	8/12	180 sec
C5	min *ChildrenUniformity*	6/20	420 sec

Discussion:

1. The discovery of irrelevant concepts is well supported in our approach. However, the discovery depends on the "uniformity" of the usage of the neighbourhood concepts. For example, in the case C1 there is a corruption, which was not found by any user. That corruption corresponds to the case that a low-used concept was introduced in the low-used neighbourhood and, therefore, could not be discovered as a problem.

2. Our approach enables a very efficient discovery of the irrelevant links since it compares the usage of a link with three types of neighbours: parents, child-nodes and siblings. It seems to be sufficient information for the discovery of irrelevancies.

3. The irrelevant sub-hierarchies can be found easily by considering the CU parameter.

4. The discovery of links, which can be split into several links, is well supported in our approach. If these links convey much more information than sibling-nodes, then they can be recognised in the very easily. However, just like in the case C1, for C4 there is a corruption, which was not found by any user since it was "hidden" in the huge usage of the sibling links.

5. The discovery of missing links/concepts is the most challenging problem. As we explained in Figure 65, it requires much more effort of an ontology manager to prove possible hypotheses about adding a new concept. However, the suggestions made by our system, based on the parameter *ChildrenUniformity*, seem to be useful.

Since there is no gold standard for the time, we do not discuss these values in detail. The duration of tasks corresponds to the level of the difficulty of the corresponding task. Moreover, all participants denoted them as a huge improvement with respect to the manual change discovery.

6.4 Related Work

Most enterprises invest a great amount of money into establishing mechanisms for detecting the user's behaviour. Many tools, algorithms, and systems have been developed to provide web site administrators with information and knowledge useful to understand users' behaviour and, consequently, to improve web site performance. Most of these approaches are based on using data mining techniques. They track and analyse click-stream data in order to obtain the most frequent paths. The results of the data mining analysis lead to the improvement of the site design and navigation opportunities or to the development of marketing strategies including recommender systems. This knowledge can be also exploited to improve the ontology that underlies the web-based system. None of the existing semantic-based approaches offers these means. Regarding this aspect, our approach can be considered as unique.

Our work presented in this section is related to semantic web usage mining. The web usage mining is the application of data mining techniques to discover usage patterns from Web data, in order to understand and better serve the needs of Web-based applications [118]. The semantic web usage mining is the web usage mining that uses ontologies in order to improve the learning. Our approach is more than the semantic web usage mining since it does not only discover usage patterns of an ontology. It rather analyses these usage patterns in order to make recommendations for ontology changes that would lead to an increase of the usability of this ontology.

Other approaches based on ontologies (e.g. [5]) have been proposed in order to take site semantics into account. In particular, in [22] is presented a framework for web personalisation

that integrates domain ontologies and usage patterns. However, none of them analyses the usage patterns with the goal to improve the ontology.

An approach for discovering of the interesting navigation patterns is proposed in [6]. It is based on the construction of the conceptual hierarchies that reflect query capabilities used in producing dynamic web pages. These hierarchies can be considered as simplified ontologies. Further, the authors define the "quality" of a web site as the conformance of its structure to the users' browsing patterns and propose suggestions for the web site improvement. On the contrary, we define the "quality" of an ontology, which is used as a backbone for an information portal (i.e. web site), based on its usability for end-users and we propose a suggestion for the ontology improvement. However, since the content and the layout of a portal are generated dynamically based on the underlying ontology, the portal itself is improved.

In [97] the authors present a framework for enhancing Web usage records with formal semantics based on an ontology underlying the site. The approach exploits the RDF annotation of static sites to map a URL into ontology entities it deals with. The dynamically generated pages are mapped to semantics by analysing their query strings. The result of a mapping is a semantic log file that contains, for each request, the time stamp, the URL or the query string, and a feature vector, which consists of ontology entities. Our semantic log file is based on a richer knowledge model. Indeed, the log ontology distinguishes between several types of the users' interaction with the portal so that, besides querying, browsing, reading etc., are supported and modelled as well. Further, semantics is not extracted from the content of a web page, but from the ontology this page is based on. Finally, we do not use a feature vector since we aim to deal with large ontologies. Rather, we organise the semantic log as an ontology (and not as an extension of a standard web log) and, consequently, reuse all the functionalities already provided by our ontology management system.

In [128] we made a comprehensive evaluation of most frequently used tools[99] for editing ontologies, Protege, OilEd and OntoEdit, by comparing them regarding several criteria, including their support for the continual ontology improvement. None of them provides support neither for the integration of the usage data into the ontology evolution process nor for the discovery of changes in an ontology, which are crucial facilities of our system. Therefore, these capabilities are novel in comparison with the existing ontology editors.

In [27] the authors present the usefulness of the ontology tools with respect to the knowledge level of the ontology engineers and the stage of development of the ontology. They conclude that for less experienced ontology engineers, the better suited tools are those that (i) require little knowledge of the underlying knowledge representation language and (ii) are easy to learn. However, even though current ontology tools do not require ontology engineers that are experts in knowledge representation and modelling, they are not yet ready for direct use by domain experts. We propose an approach for the usage-driven ontology evolution that supports an inexperienced user, who does not need to be an ontology engineer, to develop/modify an ontology.

Regarding the creation of a hierarchy, there are three approaches: top-down, bottom-up and middle-out, which start the creation of a hierarchy from the top, bottom, or middle of the hierarchy, respectively [146]. These approaches have different strengths and weaknesses. The top-down approach is better at producing crisp top-level distinctions but it can miss important low-level topics. Conversely, the bottom-up approach is better at defining all the significant low-level topics but it can produce obscure high-level topics. The middle-out approach enables the development of the hierarchy in both directions (i.e. top-down and bottom-up).

[99] http://protege.stanford.edu/; http://oiled.man.ac.uk/; http://www.ontoprise.de/com/co_produ_tool3.htm

The problem is how to find the right level of the granularity. Our approach supports the middle-out approach by determining the granularity of the hierarchy based on its usage.

The generic log file analysis process is proposed in [148]. It consists of four steps: (i) cleaning and filtration phase that removes all irrelevant information, (ii) data-cube construction that creates a cube using all relevant information, (iii) on-line analytic processing (OLAP) that is performed on cube data, and (iv) data mining technique that is put to use with the data cube to dig out the desired information. We followed this generic procedure in developing our system. However, we go a step further in two ways. Firstly, the role of our system is to combine the information about usage of ontology entities with the ontology structure itself in order to focus attention of ontology engineers on the part of the ontology that needs to be updated. Secondly, our system suggests ontology managers how to do that. Finally, we propose a formal model for a usage log, which provides explicit semantic information and consequently more scope for analysis.

Observing that in practice the meanings of relationships between concepts evolve over time, in [24] the authors define the ontology-based knowledge sharing system OntoShare that supports a degree of ontology evolution based on usage of the system. When a user shares some information, the system will match the content being shared against each concept (i.e. its keywords and phrases) in the community's ontology. The user is then able to accept the system recommendation or to modify it by suggesting alternative concept(s) to which the document should be assigned. The modification of a set of terms attached to a concept is called the usage-based ontology evolution. In this way the characterisation of a given concept evolves over time based on the inputs from the community of users. We agree that this ability to change as users' own conceptualisation of the given domain changes is a powerful feature, which allows the system to better model the consensual ontology of the community. However, this level of evolution is limited to changing the semantic characterisation of ontology concepts and does not support, for example, the automatic suggestion of new concepts to be added to the ontology.

Moreover, our approach can be used for a comprehensive management of the ontology-based applications, which incorporates the collection, the integration and the analysis of the data needed for the management. In that way, it is a unique tool. However, there are management systems for other types of the applications, which can be related to our work. For example, an approach for managing changes in a knowledge management (KM) system is given in [50]. The authors consider two types of changes: (i) functional changes that are about new KM-systems in the organisation, new versions of a KM-system and new features in one KM-system and (ii) structural changes that deal with new business models, new subsidiaries and new competencies in the organisation. The results of that study have shown that the management of the evolving KM-systems on an ad-hoc basis can lead to unnecessary complexity and failures. Both types of changes can be treated as the explicit changes, which can be very efficiently resolved in our system. However, contrary to our approach, this approach does not consider implicit changes, which can be derived from the usage of the system.

6.5 Conclusion

The possibility to cope with the implicit changes discovered from the users' behaviour seems to be the most important characteristic of an application, which aspires to be useful. Indeed, it enables the continual adaptation of an application to the changes in the users' needs, without demanding the users to provide an explicit feedback about the usability of the application. The

most common attribute for discovering changes is the usage of some structures (buttons, options in the menu, etc.), whose analysis enables their fine-tuning to the users' needs.

In an ontology-based application, the domain ontology is used as a conceptual backbone for structuring the domain information provided in the application. Consequently, the data about the usage of the application can be analysed using the ontology as the background knowledge, which alleviates the process of discovering useful changes in the application. The discovered changes lead to the improvement of the ontology, but in the end effect since the content and layout (structure) of an ontology-based application are based on the underlying ontology, by changing the ontology according to the users' needs, the application itself is tailored to the users' needs.

In this section, we presented an integrated approach for the usage-based management of the ontology-based applications, which covers capturing and structuring the users' activities with the application and the automatic discovery of changes and their systematic resolution by ensuring the consistency of the resulting ontology. The special focus was on the measures to discover some anomalies in modelling an ontology (e.g. the hierarchies of concepts) and the methods to tune this ontology to the real users' needs. The approach is based on the MAPE model. It integrates the results from the analysis of the usage data with the tools that guide the process of modifying the ontology. Since the hierarchy-based organisation of (business-) data is a very efficient solution for the improvement of the searching for information, and is more and more applied in the e-business environment, our approach seems to be a very useful method for tailoring these manually or semi-automatically produced classifications like UNSPSC[100] to the real needs of their end-users.

The evaluation experiments show that our approach can be used in the real-world applications successfully. We find that it represents a very important step in the achievement of a self-adaptive management system [121], which can discover some changes from the user's interactions with the system automatically and evolves its structure correspondingly. The benefits of the proposed approach are manifold: dynamic adaptation of an ontology (and its application) to the changes in the business environment, dynamic analysis of the user's needs and the usefulness of particular ontology to fulfil these needs, to name but a few.

Besides implicit feedback of end-users captured in the usage log, re-occurring extensions of an included ontology in many including ontologies are another force driving usage-driven ontology updates. Therefore, our approach for the usage-driven ontology evolution can be expanded by taking into account the usage of an ontology in other ontologies that reuse it. Since extensions of an ontology are driven by different ontology engineers, dependent, distributed ontologies evolve in different directions. It is necessary to analyse the evolution logs of these ontologies and to infer changes from these logs.

Recommender systems are based on the idea to guide a user through a buying process by recommending to him/her the products that are bought by users with the similar profiles. By making analogy between the user's profile and the evolution log of an ontology, the similar approach can be applied on the dependent distributed ontologies. Given a set of existing changes stored in the evolution logs of dependent ontologies, it is possible to infer rules that (i) prevent changes in the dependent ontologies by incorporating them in the included ontology, (ii) suggest and predict changes in the dependent ontologies and (iii) show up coupling between dependent ontologies that is undetectable by manual analysis.

For example, if many dependent ontologies extend the included ontology in the same or similar way, it would be better to incorporate these changes in the included ontology.

[100] http://eccma.org/unspsc/

Therefore, one possibility for the usage-driven change discovery would be to decide which changes from the dependent ontologies should be introduced into the next version of the included ontology. In this way it would be possible to guide the developer of the included ontology on how to improve its completeness and usefulness. Consequently, all the ontologies that reuse an included one will be automatically informed about changes in the included ontology since these changes have to be undone in the dependent ontologies. In this way, the possibility for the integration between dependent ontologies increase since the level of their overlap is maximised.

Moreover, such a system would be able to suggest further changes in a dependent ontology, as inferred from the other extension of the same included ontology. The re-occurrence of the extensions would increase the usability of the including ontology since the changes with the highest confidence would be suggested.

Finally, this system could compare two different extensions of the same ontology. By detecting the subsumption, the right decision would be to include the subsumed dependent ontology in the subsuming dependent ontology. On the other hand, the equivalence between two ontologies that include the same ontology would signalise that one of them is not necessary.

7 Implementation

The ontology evolution addresses an issue with important theoretical and practical implications. It helps manage the complexity and keep an ontology and dependent artefact up-to-date. However, it is costly and hard to find the skilled ontology engineers to deal with the evolution process in a quick, accurate and reliable way. Moreover, ontologies are growing beyond the human ability to manage them. It would not be possible to change them manually when the ontologies are hundreds of times more complex than the existing ones. Therefore, appropriate organisational and technical means for supporting the ontology evolution are required.

Since the methodological approach for the ontology evolution is elaborated in previous chapters, in this chapter we focus on tools. Firstly, based on the requirements for the tool support, we compare the existing evolution systems. Secondly, we reveal how to build a tool for the ontology evolution so that ontologies are more flexible to a changing environment. Finally, the benefits of our KAON system are elaborated on one case study.

7.1 Existing Support for the Ontology Evolution

Although there are several approaches for the semi-automatic ontology development ([73], [126]), most of the existing ontologies are created manually using ontology editors. Ontology editors are tools that enable inspecting, browsing, codifying, and modifying ontologies and support in this way the ontology development and maintenance task [137]. In this section we first define the requirements for an ontology editor regarding ontology evolution. Then we discuss how existing ontology editors fulfil these requirements.

7.1.1 Requirements for Ontology Editors

Available ontology editors vary in the complexity of the underlying ontology model, usability, scalability, etc. Nevertheless, all of them can be used for building a new ontology from scratch or for extending existing ontologies. Usually ontology editors provide a graphical user interface for building ontologies, which allows ontology engineers to create ontologies without using directly a specific ontology language. A survey on ontology building tools is given in [45]. However, providing support for the initial ontology development is not sufficient since ontology development is necessarily an iterative and a dynamic process [21]. Very seldom is an ontology perfect the first time it is made, and then continues, without change, to be as useful over time as it was when it was first deployed [54].

Since an ontology is usually developed using an ontology editor, many requirements for the ontology evolution have to be part of the ontology editors. An ontology editor must provide

an interface that allows ontology engineers to modify the underlying ontology. The interface has to be based on a set of available ontology changes. Moreover, there are many additional features, which can significantly improve the usability of an ontology editor and enhance its functionality regarding the ontology evolution. In this section, we discuss the most critical requirements for ontology editors in order to be more robust to a changing environment.

Indeed, we specified a number of relevant criteria and applied them to evaluate the different ontology editors. We claim that ontology editors fulfilling these requirements help an inexperienced ontology engineer modify an ontology in the easiest way. The following requirements are derived from the ontology evolution process that is introduced in chapter 3:

- *Functionality requirement* - Functionality requirement comes out from the change representation phase of the ontology evolution process. It specifies which functionality must be provided for the ontology development and evolution. Therefore, it covers a set of changes that are supported by a system. They are split into elementary and composite ontology changes. This functionality heavily depends on the underlying ontology model. A more powerful and expressive model requires a richer set of modelling primitives;
- *Customisation requirement* - The ontology evolution is a process of changing an ontology while maintaining its consistency. The goal of the ontology evolution is thus to evolve an ontology from one consistent state to the next. Since there are several different final consistent states, a mechanism is required for ontology engineers to manage changes resulting not in an arbitrary consistent state, but in a consistent state fulfilling the preferences of an ontology engineer. In order to enable an ontology engineer to obtain the ontology most suitable to her needs, an ontology editor should allow the customisation or control of the ontology evolution;
- *Transparency requirement* - To improve understanding of effects of each change, an ontology editor should provide maximum transparency into details of each change being performed. Transparency should provide a human-computer interaction for evolution by presenting change information in an orderly way, allowing easy spotting of potential problems and alleviating the understanding of the scope of the change;
- *Reversibility requirement* - The reversibility requirement states that an ontology editor has to allow undoing changes at the ontology engineer's request. Consequently, an ontology engineer can control changes and make appropriate decisions;
- *Auditing requirement* - As business applications of ontologies proliferate, so do the needs for auditing ontology evolution. Changes to business information are often accompanied with responsibility for their effects on the business. Auditing is therefore a typical component of business systems, and must be reflected in the ontology editor as well;
- *Refinement requirement* - This requirement states that potential changes improving the ontology may be discovered semi-automatically from the ontology-based data and through the analysis of the user's behaviour. An ontology editor should make recommendations for the ontology improvement;
- *Usability requirement* - An ontology editor addresses the issue of presenting ontologies and allowing the ontology engineers to operate on ontologies in a consistent way. It also addresses how different functions are integrated into the system in a way natural to the user. An ontology editor has to have an interface that enables ontology engineers to create and maintain ontologies, one that is easily understood and allows them to work efficiently with all the complexities inherent in an ontology editor.

The comparison of ontology editors against the evaluation framework is given subsequently.

7.1.2 Evaluation of Existing Ontology Editors

Ontology editors are tools that allow users to visually manipulate ontologies. The number of tools for building ontologies developed in the last years is high[101]. In this section, we evaluate ontology editors in terms of the requirements for the ontology evolution that are given in previous section. We select three ontology editors that are most frequently used in the Semantic Web community:

- Protégé[102] - It is a graphical and interactive ontology-design and knowledge-acquisition environment that is being developed by the Stanford Medical Informatics group (SMI) at Stanford University. Its knowledge model is OKBC compatible. Its component-based architecture enables system builders to add new functionality by creating appropriate plug-ins. The Protege-OWL plugin extends the Protégé platform into an ontology editor for the OWL;
- OntoEdit[103] – It is an ontology engineering environment supporting the development and maintenance of ontologies by using graphical means. OntoEdit is built on top of a powerful internal ontology model. This paradigm supports representation-language neutral modelling as much as possible for concepts, relations and axioms. Several graphical views onto the structures contained in the ontology support modelling the different phases of the ontology engineering cycle. It has an interface to the OntoBroker, which is a F-Logic Inference Engine;
- OilED[104] – It has been developed by the University of Manchester. It is a simple freeware ontology editor, which allows the user to build ontologies using OIL and OWL, and it is not intended as a full ontology development environment. Consistency checking and automatic classification of the ontologies written with it can be performed using the FaCT reasoner.

Table 14 shows the result of comparison of these editors on previously defined dimensions.

The basic *functionality* of each ontology editor is specified as a set of elementary ontology changes. Thus, all editors allow such modifications. Even though composite changes allow an ontology engineer to update an ontology without having to find the right sequence of elementary modifications, most of the existing ontology editors do not include composite changes. Only OntoEdit provides support for some composite changes (e.g. copy).

Most of the existing systems for the ontology development provide only one possibility for realising a change, and this is usually the simplest one. For example, the deletion of a concept always causes the deletion of all its subconcepts. It means that users are not able to control the way the changes are performed. Consequently, the *customisation* is not supported at all.

Moreover, the users do not obtain explanations why a particular change is necessary (*transparency*). In OntoEdit, the user only obtains the information about numbers of induced changes but without providing more details. None of existing editors warns ontology engineers about changes in the included ontologies.

[101] http://www.ontoweb.org/download/deliverables/D13_v1-0.zip
[102] http://protege.stanford.edu/
[103] http://www.ontoprise.de/com/co_produ_tool3.htm
[104] http://oiled.man.ac.uk/

Furthermore, there is no possibility to undo effects of changes (*reversibility*). Protégé and OntoEdit have Edit menu with the Undo/Redo options. However, the performed changes are kept in the memory so that they are lost when the ontology/editor is closed.

Table 14. Evolution support within ontology editors. Description: "-" means that there is no support, "~" states that support is partial and "+" corresponds to the full support.

Editors/ Requirements	Protege	OntoEdit	OilEd
Functionality			
elementary	+	+	+
composite	-	~	-
Customisation	-	-	-
Transparency	-	~	-
Reversibility	~	~	-
Auditing	~	-	~
Refinement	-	-	-
Usability			
user-friendly	+	+	+
verification	~	~	~
validation	-	-	-

Regarding the *auditing* requirement, OilEd has the activity log. However, it records connections to the reasoner, not all ontology modifications. Protégé also has the command history option but in the version we were dealing with it was useless since it was disabled.

As known to the authors, none of the existing systems for ontology development and maintenance offer support for (semi-)automatic ontology improvement, even though it would make the ontology easier to understand and cheaper to modify. Therefore, the *refinement* requirement is not supported at all.

Concerning the *usability* of ontology editors, most of the existing ontology editors have a very similar layout. They are ergonomically correct to minimise human errors. They enable operating "quickly" enough, as this is often considered being one of the most important easy-for-use issues. However, more features are required in existing editors to ensure the successful collaborative building of ontologies. Moreover, all editors can detect logical conflicts (verification) but they do not provide enough information to analyse the sources of conflicts. However, none of the existing editors provide the means for answering questions such as how, why, what if, etc. (validation).

This analysis shows that the current versions of ontology editors offer enough functionalities to allow ontology engineers to build ontologies. However, their support for the ontology evolution can be improved significantly. Consequently, they do not provide a full ontology development environment. Further, they cannot actively support the development of large-scale ontologies, the reuse of ontologies, versioning, and many other activities that are involved in the ontology life cycle process.

7.1.3 Conclusion

In order to enable an ontology engineer to obtain an ontology most suitable to her needs, we investigate the requirements for an ontology editor. We identify several means to do that: (i)

to enrich the list of possible changes; (ii) to enable an ontology engineer to control a way of resolving the changes; (iii) to inform her about all effects of a change; (iv) to allow undoing changes; (v) to allow inspecting the performed changes; (vi) to suggest an ontology engineer to generate a change; (vii) to identify inconsistencies and to provide answers to the questions such as how, why, what if, etc. We believe that an ontology editor that fulfils these requirements will enable maintaining an ontology more easily and according to the user's preferences. In the rest of this section we discuss the KAON system that is designed and built according to these requirements.

7.2 KAON Ontology Evolution

Since the ontology evolution is a very complex task, methods and tools for its support are needed. We developed an ontology evolution system within the KAON framework (see next section) that does not only enable automating the ontology evolution, but also helps ontology engineers realise a request in a most suitable way by providing additional information (e.g. the explanation) as well as by making reasonable suggestions for the continual ontology improvement. From the engineering's point of view, the advantage is in getting a system, which improves the reliability and the decision making process by reducing at the same time the complexity, staffing and other expenses (e.g. redoing modification, performance problem, etc.). Our goal was to minimise the impact of problems and to direct the resolution of the problems by controlling and managing changes. Therefore, our system is also able to modify an ontology in several ways by enabling ontology engineers to customise the modification process. Moreover, the system offers meta-level resolution capabilities by taking into account the time needed to perform changes, the number of additional changes, the structure of the final ontology, etc. Finally, the system helps in adapting the ontology towards needs of end-users that are discovered from the usage of this ontology.

In the rest of this section we elaborate on the KAON ontology evolution support.

7.2.1 KAON

Karlsruhe Ontology and Semantic Web framework (KAON[105]) has been developed at FZI[106] and AIFB[107] at the University of Karlsruhe and used to realise several ontology-enabled projects. Its primary goal is to establish a platform needed to apply Semantic Web technologies ([9], [31]) to E-commerce and B2B scenarios, knowledge management, automatic generation of Web portals, E-Learning, E-Government etc. An important focus of the KAON framework is on integrating traditional technologies for the ontology management and application with those typically used in business applications such as relational databases. In this section we describe how the ontology evolution has been realised within the KAON framework.

The simplified conceptual architecture of the KAON system emphasising points of interest related to the ontology evolution is presented in Figure 66. Roughly, the KAON components can be divided into three layers:

- *Applications and Services Layer* realises UI applications and provides interfaces to non-human agents. Among many applications realised, OI-modeller provides ontology and metadata engineering capabilities. It realises many requirements related to the

[105] http://kaon.semanticweb.org
[106] http://www.fzi.de/wim
[107] http://www.aifb.uni-karlsruhe.de/Forschungsgruppen/WBS/

ontology evolution. The KAON Portal is a tool for building ontology-based Web sites. Regarding the ontology evolution, it provides support for the tracking of the end-users' interactions with the portal;

- KAON API as part of the *Middleware Layer* is the focal point of the KAON architecture since it realises the model of ontology-based applications[108]. It consists of a set of interfaces for access to ontology entities. For example, there are Concept, Property, and Instance interfaces containing methods for accessing ontology concepts, properties, and instances, respectively. Client can access the API through a sublayer providing different types of interfaces (local, remote or Web service interface). The bulk of requirements related to the ontology evolution is realised in this layer and is described in the rest of this section;
- *Data and Remote Services Layer* provides data storage facilities, currently offering file- and J2EE-based storage. This layer also realises security and transactional atomicity of updates. Security deals with access rights to an ontology by ensuring that only trustable transactions are performed. Transactions are used to ensure that a sequence of changes is treated as a unit.

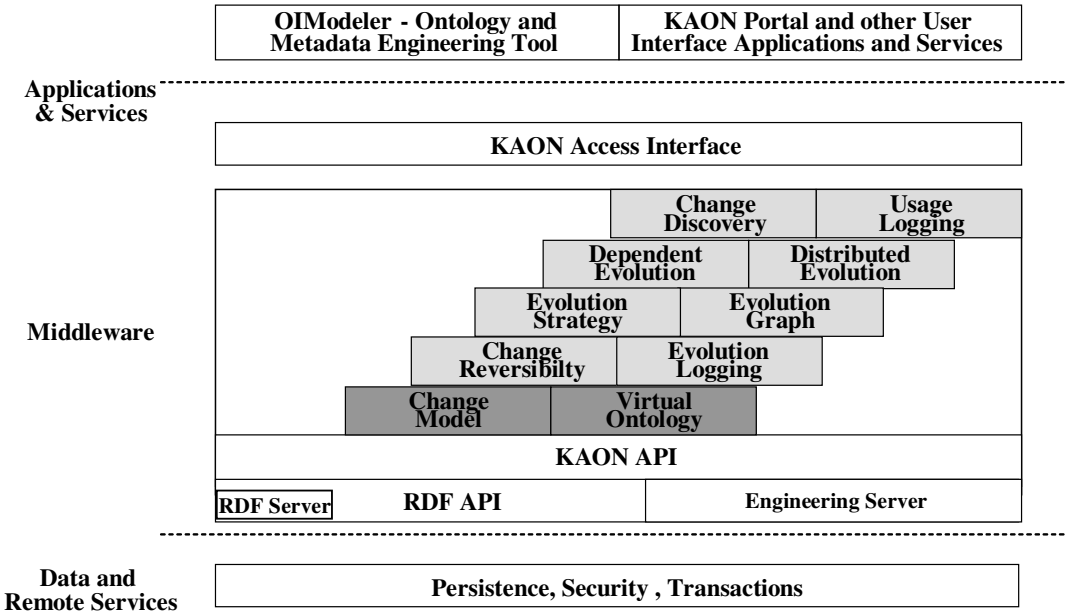


Figure 66. Conceptual KAON Architecture with Respect to Ontology Evolution

KAON API

The KAON API is a set of interfaces offering programmatic access to the KAON ontologies. It contains classes such as Concept, Property and Instance. The API reflects the capabilities of the KAON language (see section 2.2). The API decouples the ontology user from actual ontology persistence mechanisms. There are different implementations for accessing RDF-

[108] The term model refers to the model component of the Model-View-Controller architectural pattern.

based ontologies accessible through the RDF API[109] or ontologies stored in relational databases using the engineering server.

One implementation of the KAON API is based on the RDF API and thus allows access to RDF repositories. Although it offers capabilities for accessing remote RDF repositories, such as RDF Server, it is primarily used for management of local RDF ontologies stored as files.

Engineering Server is an implementation of the API directly based on relational databases and is targeted for cases where concurrent access is needed. The name Engineering Server stems from the fact that the database schema is optimised for the ontology engineering during which adding and deleting concepts are frequent operations that must be done transactionally [80]. Therefore, the engineering server uses a fixed number of tables, rather than a table per concept. The server has been heavily optimised and tested on an ontology consisting of 100,000 concepts, 66,000 properties, and 1,000,000 instances, where loading related information about 20 ontology entities takes under 3 seconds, while deleting a concept in the middle of the concept hierarchy takes under 5 seconds. This informal test has been conducted on a usual single processor desktop computer running Windows XP with 256MB of RAM.

The KAON API incorporates important elements required for the ontology management:

- *Evolution logging* is responsible for keeping track of the ontology changes in an evolution log in order to be able to reverse them at the user's request. Further, the evolution log is also used by the distributed ontology evolution;
- *Change reversibility* enables undoing and redoing changes made in an ontology. Consequently, changes can be executed in reverse order thus forcing the ontology to return to the conditions prior to the change execution;
- *Evolution strategy* is responsible for ensuring that all changes applied to the ontology leave the ontology in a consistent state and for preventing illegal changes. Also, the evolution strategy allows the user to customise the evolution process;
- *Evolution graph* enables ontology engineers to enhance a set of changes with their own changes and to resolve them;
- Ontology inclusion facilities, together with the *dependent evolution*, are responsible for managing multiple ontologies within one node;
- Ontology replication facilities, together with the *distributed evolution*, are responsible for enabling the reuse and the management of distributed ontologies;
- *Change discovery* includes the means for the discovery of problems in an ontology and for making recommendation for their resolution;
- *Usage logging* is responsible for keeping track of the end-users' interactions with ontology-based applications in order to adapt ontologies to the users' needs.

Other aspects such as the *change model* and *virtual ontologies* that are important from the implementation point of view and are not directly visible to the ontology engineers are described in the rest of this section.

Additionally, the KAON API provides support for optimisation and concurrency. The optimised-loading component is responsible for the bulk loading of ontology entities. To improve performance, entities are cached at the client. Concurrency conflict detection is responsible for detecting and resolving conflicts resulting in concurrent updates of different ontology engineers. For example, if one ontology engineer updates the ontology, then other

[109] http://www-db.stanford.edu/~melnik/rdf/api.html

active ontology engineers must be notified of this update. Alternatively, if an ontology engineer attempts to update the ontology using stale information, the conflict must be detected.

Change Model

We realised that the change procedures have to be short and simple and not complicated with the provision for handling all possible consequences of a change. It is better to attend these consequences in separate procedures. The rationale behind this decision was also that a change can have very different consequences depending on the situation and it is impractical to include provisions for handling all of them in every procedure. In addition, shorter procedures are easier for a user to understand and follow.

To achieve that, the KAON API objects do not contain methods for performing changes. Creating a list of change events modifies a model. The list of change events is then applied on the actual model. Furthermore, several changes may be collected in a list, which can then be passed to the model. Lists of changes are passed to the model in a transactional manner. This means, that all changes are either applied together or no changes are applied at all. Further, our system attends a consequence of a change as soon as it occurs. This recursive realisation may produce a high degree of nesting. However, it allows to clear separation between ontology changes.

There are several reasons for using the change event design rather than following the more standard design of having methods that change the model:

- A set of changes to be applied to the model presents a unit of work. Either all changes are applied, or none at all – the list of requested changes is a natural transaction boundary;
- In many cases the KAON API is used to access remote servers. Packing many requests in a list and sending them to the server all at once reduces network communication overhead;
- In order to implement evolution strategies, elementary changes needed to be represented as objects. It would impossible to implement evolution strategies if changes are performed through method calls;
- Event-based design further enables evolution strategies that optimise updates. For example, elements are often moved in the ontology and attached to another place. If implemented strictly sequential, the move operation would first remove an element together with all ancillary data, and add the elements afterwards at the new location. By representing all requested changes as objects, the evolution strategy knows in advance what all changes need to be performed, and can prevent unnecessary deletion and later adding of information.

Virtual Ontology

Ontology evolution covers two main requirements: (i) preventing illegal changes and (ii) keeping consistency. The first requirement states that some users' request for a change cannot be applied since the consistency constraints cannot be satisfied. For example, the root concept must not be deleted. The second requirement is related to the fact that a single change will often leave an ontology in an inconsistent state. Additional changes are necessary to fix other parts of the model. For example, the user may request deleting a concept. To perform this

change, the respective concept must first be detached from its parents and children; children need to be reconnected to some other node, etc. Thus, a single user-initiated change may be rejected or may cause additional changes to be executed.

To fulfil these requirements as optimal as possible we have developed the so-called virtual ontologies. Virtual ontologies as separate implementation of the KAON API are developed to yield better performance. Indeed, virtual ontologies ensure that the KAON API has been heavily optimised for the performance. The idea is to minimise the number of ontology entities being considered. A virtual ontology represents a view on an ontology that is being changed. It contains a copy only of a part of the ontology that is relevant for a change. In this way the largest part of the ontology can be kept intact and only a part related to the current request for a change need to be loaded in the virtual ontology. Therefore, virtual ontologies offer facilities for the partial replication.

In the rest of this section we discuss how previously mentioned requirements are satisfied. Regarding the first requirement, in the case that the request fails, the roll back of the ontology into the initial state is needed. However, since changes are applied on the virtual ontology, the original remains unchanged.

Furthermore, checking the entire ontology is costly, and when changes are small and local, unnecessary for the most part. If an ontology has many concepts, then considering all of them will take a lot of time and will require plenty of memory. Therefore, it will result in horrible performance. To overcome that and, consequently to fulfil the second requirements, our optimisation strategy exploits two principles:

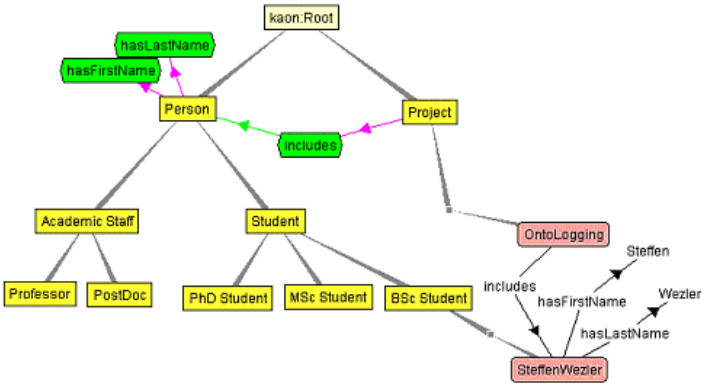
- to minimise the number of ontology entities that are considered for one change request;
- to optimise the loading of relevant entities by applying bulk-loading.

The first principle is based on taking into consideration only a part of the ontology that is pertinent for a request. Since the time taken to perform an ontology change is inconsequential compared to the time taken to check all entities, it is essential to be focused only on a relevant part. Therefore, virtual ontologies are preferable, while they enable incremental checking.

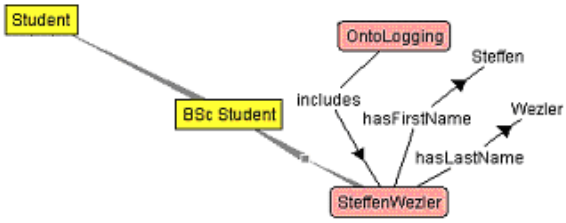
Note that virtual ontologies are created dynamically. Figure 67 shows two virtual ontologies that are created for the same request for the removal of the subconcept relationship between concepts "*Student*" and "*BSc Student*". As shown in Figure 67, the content of the virtual ontology is not defined in advance. It depends on the request for a change, on the original ontology as well as on the selected evolution strategies. The virtual ontology VO1 (cf. b in Figure 67) is the result of the evolution strategy that removes orphaned concepts and all its instances whereas the evolution strategy of the virtual ontology VO2 (cf. c in Figure 67) preserves instances of the orphaned concepts that are deleted. Both virtual ontologies contain the concepts "*Student*" and "*BSc Student*" since both of them occur in the request for a change (i.e. *RemoveSubConcept("BSc Student","Student")*). After the application of this change, the concept "*BSc Student*" does not have parent concepts (i.e. it is an orphaned concept) and, therefore, it has to be deleted as well. This request causes the removal of all its instances. Consequently, both virtual ontologies contain the instance "*SteffenWezler*". Since the first virtual ontology is associated with the evolution strategy that requires the removal of instances, it contains all property instances of this instance. On the contrary, the evolution strategy of the second virtual ontology demands the preservation of instances. Since there is no property defined for the concept "*BSc Student*", this virtual ontology does not require any additional entities from the original ontology. Note that Figure 67b) and Figure 67c) show the entities that occur in virtual ontologies VO1 and VO2 respectively during the change application, and it does not mean that all of them are available at the same time.

Moreover, the time for performing a change is largely determined by the time taken to fetch and then later to flush the entities from the memory. Apart from the usage of only relevant entities from the original ontology, a key to the ontology evolution is to access as many relevant data as possible in one request, rather than accessing data one at time. Our experience showed that the bulk-loading offers significantly better performance.

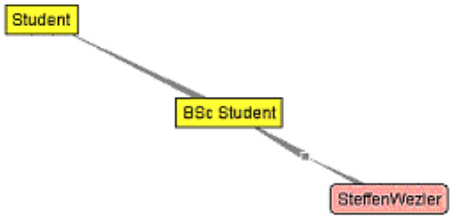
Subsequently we discuss how ontology evolution is realised within the KAON API and within various applications that have been realised on top of the KAON API.



a) Original ontology



b) Virtual ontology VO1



c) Virtual ontology VO2

Figure 67. An example of virtual ontologies

7.2.2 Ontology Evolution in the KAON API

Management of ontology changes is realised through pluggable evolution strategies (see chapter 4) so that an ontology engineer can adapt the behaviour of various change operations according to her needs. For example, when deleting a concept from the ontology, it is possible to direct the API to remove its children as well, or to preserve them. For each change, the ontology engineer is provided with detailed insight into the consequences of a change before the change is actually applied to the ontology. In order to guarantee that updates will leave an ontology in a consistent state, requested changes should always be passed to the evolution strategy first. Therefore, the general pattern for changing an ontology is as follows:

Step 1. Build the list of requested changes;

Step 2. Pass this list to the evolution strategy to compute all necessary changes;

Step 3. Pass the expanded list of changes to the ontology.

The evolution strategies may be configured with different parameters that affect how the strategy calculates pending changes. An example for alternative evolution behaviour is the situation when a concept is to be deleted. Then one may alternatively delete the subconcepts as well, connect them to the parent or connect them to the root concept.

Therefore, before the ontology evolution process is started, a particular evolution strategy must be configured. Changes to the ontology are performed by assembling elementary and composite changes into a sequence (step 1). However, before the ontology is actually updated, this sequence is passed to the present evolution strategy to perform steps described in chapter 4 on the semantics of change phase of the ontology evolution process, resulting in an extended sequence of changes (step 2).

To ensure atomicity of updates, either all or no change from the extended sequence of changes should succeed, so that the validity of the change sequence is checked before any updates are actually performed. If the request for a change will result in an ontology that does not obey the constraints of the ontology model, i.e. if the resulting ontology is not consistent ontology, the request for a change is rejected with an explanation of the violated constraints.

Transparency is realised by presenting the extended sequence of changes to the ontology engineer for the approval. To further aid the understanding of why some changes are performed, the evolution strategy may group related elementary changes and provide explanations why a particular change is necessary, thus greatly increasing the chances that all side effects of changes will be properly understood.

After the ontology engineer reviews the changes, they are passed to the ontology and executed, performing steps from the change implementation phase (step 3).

It is obvious that for each elementary change there is exactly one inverse change that, when applied, reverses the effect of the original change. With such infrastructure in place, it is not hard to realise the reversibility requirement: to reverse the effect of some extended sequence of changes, a new sequence of inverse changes in reverse order needs to be created and applied.

As mentioned in section 3.2.4, the evolution log is needed to track information about performed changes and to associate additional information with each change. Effectively, the log is treated as an instance of the evolution ontology (see section 3.2.4) consisting of concepts for each change, making it is easy to add meta-information to log entries. Structure of the log may be easily customised by editing the evolution ontology. Further, available services for persisting ontology data may be used to persist the log, removing the need to devise yet another type of persistent storage.

Evolution logging and reversibility services are provided as special services of the KAON API, allowing different applications reuse these powerful features. For example, actions performed in one application may be easily reverted in another.

7.2.3 Ontology Evolution in the KAON Applications

Building on top of the KAON API, various tools have been realised within the Application & Services layer, such as OI-modeller for the ontology and meta-data engineering and the KAON Portal for (semi-)automatic generation of Web portals based on the conceptual description. These tools are responsible for providing the user interface. In this section we discuss the support they provide regarding the ontology evolution.

Ontology Evolution in the OI-modeller

As mentioned in the previous section, the ontology evolution is primarily realised through the KAON API. However, UI applications provide human-computer interaction for the evolution, whose primary role is to present the change information in an orderly way, allowing easy spotting of potential problems. Also, any application that changes the ontology must realise the reversibility requirement in its user interface as well.

Within the KAON framework we have developed OI-modeller, the ontology and metadata engineering tool. It is an end-user application that realises a graph-based user interface for single, dependent and distributed ontology development. OI-modeller supports ontology evolution at the user level. Figure 68 shows a modelling session where an ontology engineer attempted to remove the concept "*BSc Student*".

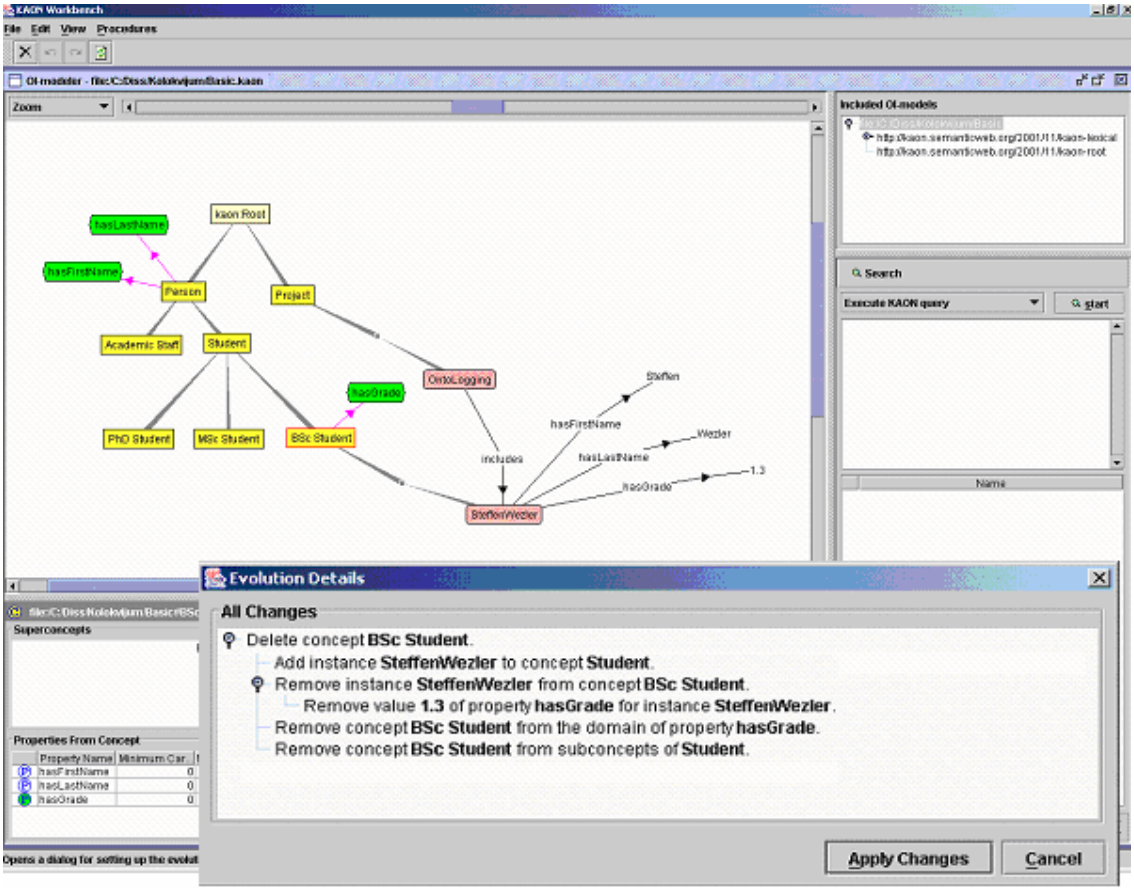


Figure 68. Ontology Evolution in KAON framework: User Interface in OI-modeller

Currently evolution requirements are realised within the OI-modeller, as follows:

- As shown in Figure 69, an ontology engineer may set up the desired evolution strategy. It can be seen that an evolution strategy consisting of several resolution points. For each resolution point the ontology engineer must choose appropriate elementary evolution strategy;
- Before changes are performed, the system computes the set of additional changes that must be applied. The impact of a change is reported to the ontology engineer. Presentation of changes follows the progressive disclosure principle: related changes

are grouped together and organised in a tree-like form. The ontology engineer initially sees only the general description of changes. If she is interested in details, she can expand the tree and view complete information. Only when the ontology engineer agrees will the changes be applied to the ontology. The ontology engineer may cancel the operation before it is actually performed. This is depicted in the bottom-right part of Figure 68 and further elaborated in an example;

- An unlimited undo-redo function is provided. Although this function is by large the responsibility of the KAON API, the user interface is responsible for restoring the visual context after an undo operation. For example, if a concept in hierarchy was selected and then deleted, when operation is undone, the same concept must be selected. If the hierarchy was scrolled in the meanwhile, the original scroll position must be restored. These features are necessary for the ontology engineer to quickly recognise a familiar state and proceed with her work. If not done properly, although an action is undone, the ontology engineer may not realise this and may mistakenly request another undo operation.

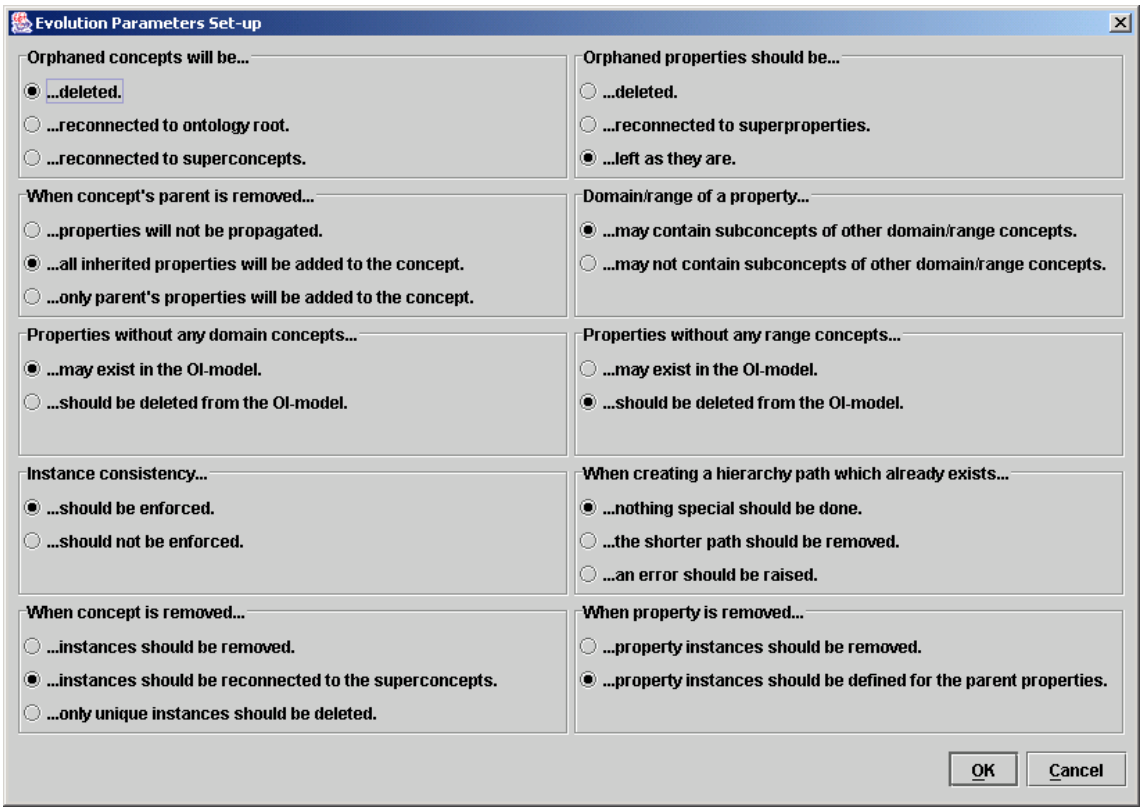


Figure 69. Ontology Evolution in KAON framework: Evolution Strategy Set-up

A sample screenshot of the evolution support in OI-modeller is given in the bottom-right part of Figure 68. In this scenario, the ontology engineer requested to remove concept "*BSc Student*". The evolution strategy decided to push instances of that concept to its parent. By opening a node in the tree, the ontology engineer can see what changes will actually be performed. Hence, the change information can be viewed at different levels of granularity.

We consider reusing existing ontologies as an integral part of ontology engineering. Therefore, OI-modeller also supports working with multiple ontologies at the same time. The

upper right-hand corner of Figure 68 shows the inclusion graph among all open ontologies. By selecting a particular ontology, the ontology engineer signals that a new entity should be created within this ontology. The evolution subsystem takes care of maintaining of consistencies in all open models. The modularization facilities have been implemented within the KAON API.

Ontology Evolution in the KAON-Portal

The KAON Portal is the tool for the building ontology-based Web sites. It is used for presentation and browsing of ontologies in the Web as shown in Figure 70 . In the central part of Figure 70, the initial information about the concept hierarchy is presented. On the left-hand side, there are controls for easy navigation to some part of the ontology (marked as shortcuts), followed by the controls for changing current language and by the control for searching the ontology.

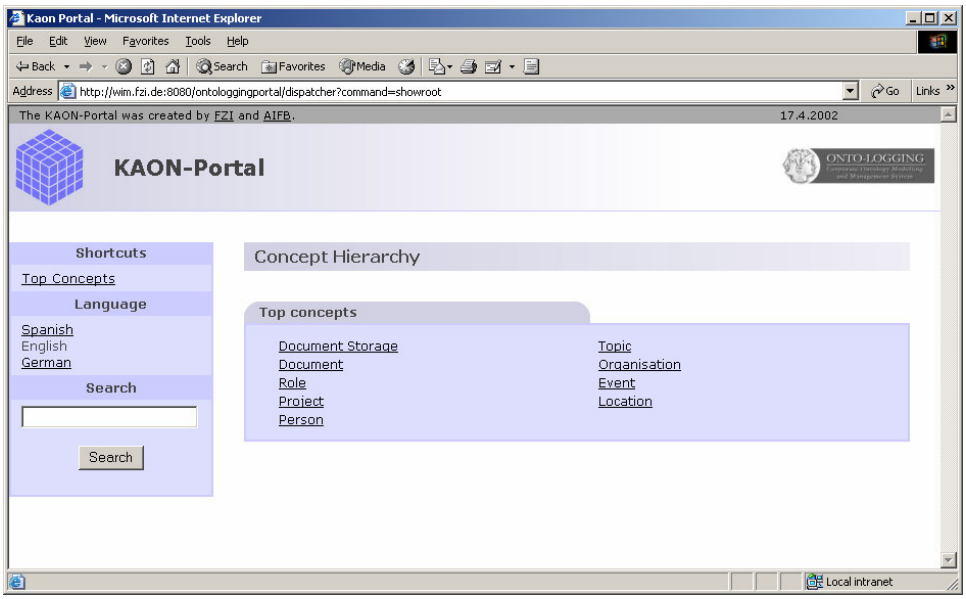


Figure 70. KAON Portal

The KAON Portal keeps track about all users' interactions with the system in the form of the semantic log (see section 6.2.1). On the other hand, the KAON API incorporates mechanisms for assessing the ontology (and by extension an application based on it) with respect to different criteria as well as mechanisms enabling us to take actions to optimise it. To prepare data for the analysis, which is performed in the KAON API, we developed means to aggregate, transform and correlate the usage data, which is produced in the KAON Portal. Here we want to emphasise some implementation details.

Regarding the ontology evolution there are three main functionalities:

(i) to *collect* data from different, possibly distributed logs in case an ontology-based application is deployed on several web servers;

(ii) to *pre-process* data by transforming disparate data into meaningful information. This phase also covers the cleaning and validation of the data for achieving the required quality;

(iii) to *organise* them in a way that enables a fast and efficient access to the data.

In order to integrate data from various servers, we replicate the *Semantic Logs* of all these servers into a "common" log, so called the *OntoLog*. Since all logs are based on the *Log Ontology* (see section 6.2.1) and they reference the same domain ontology, the semantic heterogeneity problem does not occur. Another possibility for the integration was to integrate the logs virtually (on-the-fly) by accessing them at the processing time. Such a solution would enable the immediate visibility (actuality) of log data, but it requires extensive distribute processing and, thus, it is slow and expensive. Since the analyses we want to perform are statistic-based, the actuality of the data is not so critical. However, the update of the *OntoLog* is performed periodically (currently once per week).

Moreover, during this phase, the data is also pre-processed, in order to make it better suited for further analysis. We perform two types of data pre-processing:

1. *Data abstraction* - Since the interaction of the users with the portal is mainly done on the level of ontology instances, the semantic logs (and consequently the *OntoLog*) mainly contain the information about the usage of ontology instances. For example, if a user has seen more details about the project "*OntoLogging*", the log file recorded this information explicitly. However, the goal of our system is to improve the ontology and not its knowledge base. Thus, all log entries regarding ontology instances have to be transformed into corresponding ontology concepts. Regarding the previous example, all the appearances of the instance "*OntoLogging*" in the *OntoLog* have to be replaced with the concept "*Project*";
2. *Extracting links* - the most important information for the analyses we want to perform is the frequency of browsing[110] relations between two concepts (see section 6.3.2). Since the *OntoLog* does not contain explicit information about the source and the target of a browsing event[111], we extract it in the pre-processing phase by analysing successive events. For example, regarding the part of the log presented in Figure 63b, from two successive browsing events ("*Browsing123*" and "*Browsing456*") our system concludes that the link between concepts "*Project*" and "*EUProject*" was browsed since the first event is related to the concept "*Project*", and the second one to the concept "*EUProject*".

Finally, the integrated and pre-processed data has to be analysed in order to enable the ontology engineers to manage the ontology efficiently. However, with increasing frequency of the application usage, the log might contain a large quantity of data. Thus, it has to be reengineered, to enable ontology engineers to perform sophisticated data analysis through a fast access to a variety of possible views of the underlying information. Further, in order to get a fast response, it is useful to pre-calculate at least some of the information that will be needed for analysis. Since OLAP techniques [62] typically handle huge volumes of data that is interrelated in complex ways, and enable the pre-calculation of everything that may be needed, we decided to transfer the log into an *OLAP cube* [124].

The *OLAP cube* consists of four dimensions: time, visitor, event and entity indicating when, by whom, how (querying or browsing) and what is visited. Since the entity dimension, which represents ontology entities, cannot be normalized, the *OLAP cube* has the snowflake structure. The fact table contains the foreign keys to the dimension tables as well as the numeric facts relevant for the usage-driven change discovery. The lowest level of data in the fact table is at a much higher level than in the *OntoLog*, since the log data is aggregated to a level where patterns and trends can emerge and analysis is meaningful. In this way, the

[110] Browsing is treated as a click on the hyperlink between two concepts that are in a direct hierarchy relation.

[111] The *Log Ontology* models the dependency between events through the relation "*previousEvent*" (see Figure 63a).

OntoLog only contains the pre-processed information about the users' interactions, which is needed to improve the ontology whereas the *OLAP cube* enables the analysis of this information at an aggregate level.

Indeed, the *OLAP cube* as a part of the ontology evolution support performs various in-advance analyses in order to speed up the decision making process. The most important data (see sections 6.3.1 and 6.3.2) is the number of browsing the *direct hierarchy relation* between two concepts *c1* and *c2* (denoted as *Usage(c1,c2)*) and the number of querying[112] for a concept *c* (denoted as *Querying(c)*). By processing the *OntoLog,* these values increase. For example, by processing the part of the semantic log presented in Figure 63b, the value of Usage("*Project*", "*EU Project*") and the value of Querying("*Project*") will be incremented. More information about the analyses we perform is given in section 6.3.

In the current implementation, the *OLAP cube* is queried via a web service. An advantage of using web service is that it enables having a thin client that can access the OLAP data in a remote server without threatening the security of the server.

7.3 OntoGov Case Study

In this section we discuss the experiences from applying ontology evolution in one real-world scenario. The ontology evolution is essential part of the ontoGov[113] project which aims at developing a self-managing system in the E-Government domain. The increasing complexity of E-Government services demands a correspondingly larger effort for management. Today, many system management tasks such as service re-configuration due to changes in the law are often performed manually. This can be time consuming and error-prone. Moreover, it requires a growing number of highly skilled personnel, making E-Government systems costly. In this section we show how the usage of semantic technologies for describing E-Government services can improve the management of changes. We have extended our work in ontology evolution in order to take into account the specificities of ontologies that are used for description of semantic web services. Even though we use the E-Government domain as an example, the approach is general enough to be applied in other domains.

7.3.1 Introduction

E-government is a way for governments to use the new technologies to provide people with a more convenient access to government information and services, to improve the quality of the services and to provide greater opportunities to participate in the democratic institutions and processes [120]. In addition to providing new ways of working with citizens, enterprises, or other administrations, E-Government is also concerned with creating an integrated environment for the development, deployment and **maintenance** of online services [141]. In a fast changing world this last requirement is especially important. Moreover, in the current economic situation budgets are reduced and opportunities for gaining efficiency seem to be inevitable: the costs of control and maintenance have become the prime concern of public management. The emphasis in E-Government is thus shifting from implementation to cost efficient operations of service or data centres [68]. This effort includes the development of shared service centres that provide common services to local government organisations without affecting the autonomy of organisations and providing the flexibility to enhance and

[112] The number of queries related to a concept.
[113] OntoGov - Ontology enabled E-Gov Service Configuration, http://www.ontogov.org

include additional functionality [57]. In such a distributed environment, the problem of efficient management of changes in E-Government has become even more critical.

The main focus of the current change management activities is the resolution of the so-called *dynamic* modification. It refers to the problem of managing running processes when unanticipated exceptions arise during a task execution, such as the appearance of some hazards in a system, or obtaining some unexpected results [42]. These approaches ensure the consistent operation of a legacy system under unpredictable problems. However, in a dynamically changing political and economic environment, the regulations themselves have to be continually improved in order to enable the efficient function of a modern society. Taking into account an enormous number of public services and dependencies between them [1] as well as the complexity of interpreting and implementing changes in government regulations, the process of reconfiguring the existing legacy systems (the so-called *static* modification) seems to be quite complex. Indeed, an efficient management system must provide primitives to allow for the progressive refinement without rewriting it from scratch and must guarantee that the new version of the service is syntactically and semantically correct [13]. However, an efficient management system for resolving *static* changes in an E-Government domain does not exist. In this section we present such an approach.

The approach is based on enriching current mechanisms for implementing E-Government processes, i.e. web services, with semantic technologies in order to support a more efficient management of changes. Indeed, the current languages for describing web service, WSDL[114] and their composition on the level of business processes (BPEL4WS[115]) lack semantic expressivity that is crucial for capturing service capabilities at abstract levels. We argue that business process flow specifications should be defined at abstract task levels leaving open the details of specific service bindings and execution flows. This abstract level enables the definition of domain specific constraints that have to be taken into account during the (re)configuration of a process flow. In order to model this abstract representation of web services, we base our work on the OWL-S[116] ontology and on the WSMO[117] ontology, which support the rich description of web services for the Semantic Web. These ontologies lay the foundation for semantic web services [32], [79]. We extend these efforts, in order to support the efficient maintenance of semantic web services.

Since the descriptions of semantic web services are ontologies themselves, we base the web services change management on our work in the distributed and dependent ontology evolution (see chapter 5). It enables us to develop a formal framework for coping with changes which includes the consistency of the service descriptions, possible changes as well as their resolution. Consequently, we can reason about the change management process, making it very flexible and efficient.

Due to our tasks in the ontoGov project, we have applied our approach for the change management in the E-Government domain. However, it is general enough to be applied in an arbitrary application domain that uses (semantic) web services.

7.3.2 Motivating Example

In order to make the description of the approach more understandable, here we define the basic structure of an E-Government system and give a motivating example that will be used

[114] http://www.w3.org/TR/wsdl
[115] http://www-106.ibm.com/developerworks/library/ws-bpel/
[116] http://www.daml.org/services/owl-s/1.0/
[117] http://www.wsmo.org/

through the whole section.

There are four basic roles played by actors in an E-Government system as shown in Figure 71:

- politicians who define a law;
- public administrators who define processes for realizing a law;
- programmers who implement these processes, and,
- end-users (applicants) who use E-Government services.

Public administrators have the key role. They possess a very good knowledge about the E-Government domain. This knowledge is needed for the design of a public service. It includes the legislation that a service is based on, the respective law, related directives, prerequisites etc. Based on the interpretation of a law, a public administrator describes a service as a sequence of activities that have to be done, which represents a business process. For example, the generic schema[118] for the public service for issuing (renewal) a driving licence is realised through the following five activities: (i) application, (ii) verification/qualification, (iii) credential issuance, (iv) record management and (v) revenue collection.

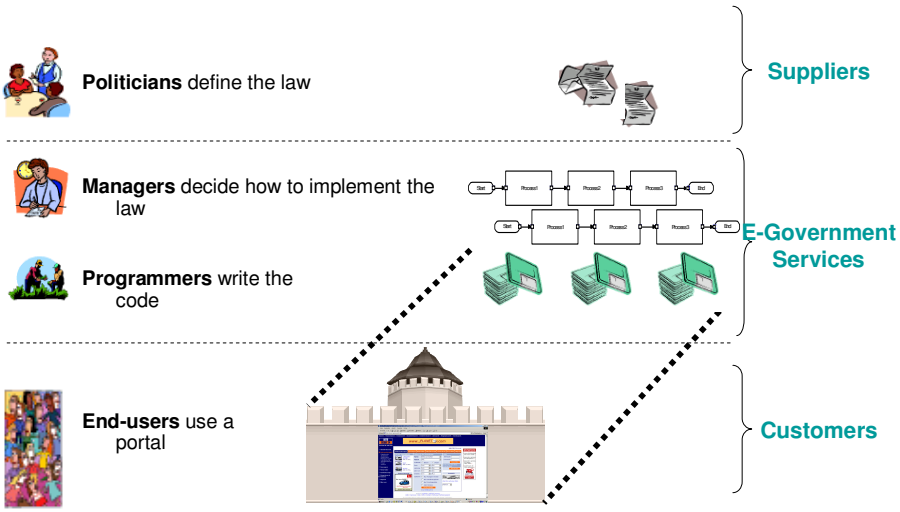


Figure 71. E-Government Framework

In the application activity all the necessary application data/documents are provided by an applicant. In the next activity, the provided information/documents are verified (e.g. validity and liquidity of a credit card) and are qualified by testing whether the applicant meets the qualification requirements. In the credential issuance activity either a permanent or temporary credential document (i.e. driving licence) is issued. The record management activity ensures the ongoing integrity of the driving licensing and control record. Finally, the required fee is charged from the applicant's bank account. Each activity requires some inputs, produces some outputs. It can be executed only when its preconditions are fulfilled and it has postconditions that define the next activity in a conditional manner. In the case of the application activity of the driving licence service, inputs include a birthday certificate, the output is an application

[118] Any process that accepts/enrolls applicants for a fee and then issues some sort of credential has the same generic structure - http://www.aamva.org/Documents/idsSeptember2003StatusReport2Attachment9.pdf

form, the precondition is that the applicant is older than 16 years and the postcondition is that all fields in the application form are filled. Further, each activity can also be decomposed into several subactivities or can be specialised.

The crucial activity is the verification/qualification since it reflects the constraints contained in the law. For example, it implements a rule that a person younger than 16 cannot apply for issuing the driving licence whereas for motor cars (category B) the minimal age is 18. From the business process management point of view, the law can be treated as the business rule required to achieve goals of an organisation (defined by its business policy).

Due to the changes in the political goals of a government, changes in the environment, and changes in the needs of the people or due to the possibility to organise regulations in a better way, the politicians might (i) make the revision of a law by accepting an amendment, (ii) enact a new law or (iii) even repeal a law. In the case of a new amendment, the public administrator must understand the changes in the law caused by the amendment; locate activities/services in which this law has to be implemented and translate changes into the corresponding reconfiguration of the business process. Let us continue the example with the driving licence. Recently the German law that regulates issuing driving licences has been changed so that foreigners from non-EU countries must have the German driving licence, although they have a domestic licence. Let us analyse which changes in the existing business process for the issuing driving licence will be caused by this legal change. For each change we discuss the role that an efficient change management system should play.

First of all, the public administrator should locate a business process and the corresponding activities that should be modified due to this change in a law. This is a time-consuming action if it is performed in a non-systematic way. Therefore, an efficient change management approach should inform the public administrator on these activities automatically. It means that each business activity must contain a reference to a chapter/section/paragraph/article/amendment of a law it implements. For example, the activity verification/qualification of the driving licence service is based on the Section 2, Paragraph "Mindestalter"[119] in the Law "Bundesgesetz ueber den Fuehrerschein".

After finding the service that has to be modified, the public administrator has to decide how to do that. She can specialise this service in the new one or she can adapt it to include new requirements. Let's assume that the public administrator made a decision to generate a specific driving licence service for foreigners. This service should not be generated from scratch. Rather, it should be a specialisation of an already existing driving licence service. The public administrator has to change the preconditions of this new service since it is only for foreigners from non-EU countries. This automatically causes a change in the preconditions of the original service[120] since the preconditions of two different services that provide the same functionality must be disjoint. Only in this way will the run-time system know which service to execute. It is clear that when the preconditions are semantically defined, the judgement about the inclusion relation among them can be done automatically.

Further, the verification/qualification activity of the new service requires checking whether a foreigner already has a domestic licence. Therefore, a new input for that activity is necessary. Since each input has to be supplied, this change is propagated to the previous activity, i.e. the application activity, which is responsible for the interaction with an applicant. It means that that activity has to deliver (as its output) the information about the domestic licence, the

[119] Since in the scope of the project the German laws were considered as examples, we mention here the original titles in German.

[120] The precondition of the original service was that a person is older than 18 years. Now, this precondition is extended by the condition that a person must be from EU.

validity of which should be tested in the verification activity. Consequently, the application activity of the new service needs an additional input compared to the original service.

Obviously, different changes in a law have different consequences in the existing services. We briefly discuss one more example. Recently the German law that regulates issuing driving licences has been changed, so that teenagers older than 17 years can obtain a (temporary) licence for motor cars if they pass the exams and if they drive with a person who is older than 25, has the driving licence for more than five years and has scored less than 20 negative points[121] in the last five years. In that case, the older person must have a licence for co-driving. This change in the law requires changes in the postconditions of the verification/qualification activity: instead of approval and non-approval of the licence, it can be temporarily approved. Further, the credential issuance activity has to generate an additional output since the new co-driving licence should be printable as well. An efficient change management system should enable the public administrator to perform all these changes efficiently (e.g. to make a minimal set of additional changes) and to ensure the overall consistency of the reconfigured service automatically (e.g. to prohibit that an activity has two contradictory preconditions).

In the rest of this section we present a change management system that fulfils the above mentioned requirements.

7.3.3 Our Approach

Given the requirements described in the section 7.3.2, we have developed an approach for the change management of semantic web services. Note that even though we use the E-Government domain as an example, the approach is general enough to be applied in other domains. In order to emphasise this generality in this section we substitute the E-Government vocabulary, used in the previous section, with the commonly used business process management terminology [116]. Therefore, instead of the term *law* we use a *business rule*, a *public E-Government service* is treated as a *business process* and a *manager* plays the role of a *public administrator*.

Since we assume that services are described using ontologies, the management of changes requires the management of these semantic descriptions. Therefore, our approach can be based on our work in ontology evolution. Moreover, we have extended the work in order to take into account the specificity of semantic web services. We firstly define these extensions of our ontology evolution approach. Then we discuss the way of bridging the gap between business rules[122] and semantic web services implementing these rules. Finally, we define the procedures for the change propagation in semantic web services by defining the semantics of the required changes.

Evolution of the Semantic Web Service Ontology

In this section, we extend our approach for the ontology evolution to the handling the evolution of semantic web service ontologies. Since the evolution is driven by the set of changes that have to preserve the consistency, the approach requires (i) the explicit specification of changes that can be applied and (ii) the consistency definition. Both of them heavily depend on the underlying model and thus they vary from application to application. Therefore, we firstly introduce an ontology that is used for describing semantic web services.

[121] Negative points are collected by participating in some traffic accidents.

[122] Note that in the E-Government domain business rules represent the laws since the laws define how to realize the E-Government services.

Secondly, we define more complex changes that can be applied to these descriptions. Finally, we specify the consistency constraints that are derived from the semantics of this ontology.

Ontologies used for modelling semantic web services

The first step that has to be clarified is the description of web services. We distinguish among the following ontologies:

- *Meta Ontology* that contains entities needed to describe services;
- *Domain Ontology* that contains domain specific knowledge;
- *Service Ontologies* that describe concrete services.

For each service, a *Service Ontology* that includes the *Meta Ontology* and the *Domain Ontology* is defined, and it might include (reuse) other *Service Ontologies*. For example, the service ontology for the driving licence issuance E-Government service (see section 7.3.2) describes that it is a composite service that is realized through the application, verification/qualification etc., which can be considered as atomic services (i.e. an activity). Therefore, it includes *Meta Ontology*, since the *Meta Ontology* defines the building blocks for the service description. Each of these services (application, verification/qualification etc.) is related to the *Domain Ontology*. For example, the application service requires the birth certificate that is the domain knowledge.

We do not consider dynamic web services whose process flow can be composed on the fly. However, we allow the dynamic binding of web services during the execution. Therefore, we focus on the static web services, whose composition is explicitly predefined by the business rules (i.e. a law). In order to model the dependency between a business rule and the service implementing it and to take into account the other specificities of the E-Government services we introduce the *Meta Ontology*. We note that it is not possible to reuse OWL-S or WSMO that are the most salient initiatives to describe semantic web services. Whereas the WSMO ontology does not contain the process model, the OWL-S ontology does not allow[123] using the domain ontology entities as inputs/outputs of an activity in the process model. Moreover, the formalism for expressing conditions is not defined.

Similarly to the OWL-S ontology, the *Meta Ontology* consists of two parts: the profile that is used for the service discovery and the process model that is used to describe the process flow. To define the *profile* we extend the *OWL-S service profile ontology* in several ways. First, we define the property "*hasReferencedBusinessRule*" that establishes a reference between the service description and the business knowledge that is represented in the form of an ontology. This ontology is called *Business Rule* ontology and depends on the application domain. In the E-Government domain, this ontology contains the knowledge about laws, and is called the *Legal Ontology*. It is important mentioning that this ontology may be used as a well-defined vocabulary (semantics) for describing (annotating) both the content and the structure of legal documents [40]. However, for the problem we are aiming to resolve, it is necessary to model only the structure of legal documents, not their content. The description of the *Legal Ontology* is given in [123].

The second extension of the service profile ontology comes from the business process modelling point of view. Indeed, in order to model the resources involved in a business process, we introduce additional entities such as the property "*requires*" and the concept "*Resource*" which can be either a person who is involved in the executing a service or an equipment (i.e. hardware or software) that performs a service automatically. In that way, we

[123] In OWL Lite and OWL DL classes and individuals form disjoint domains. Even though OWL Full allows the freedom of RDF Schema: a class may act as an instance of another (meta)class, it cannot be used, since it is not decidable.

establish a bridge between the common language used by business people – in order to describe the business processes (i.e. web services) - and the ontology language used for describing web services.

Finally, the last extension of the *OWL-S service profile ontology* is achieved by taking into the consideration the standard metadata defined for the particular domain, since ontologies may advance metadata solutions. Our goal was to model all information that exists in the standard including the implicit knowledge. Even though we use the *CEN Application Profile v.1.0 metadata standard*[124], which is used as a standard in the E-Government domain, we note that similar strategies can be applied for other standards as well. The approach can be summarized as follows: (i) the metadata standard is transformed into a set of the ontology properties that are explicitly included in the *Meta Ontology*; (ii) the *Meta Ontology* is extended with several concepts (e.g. the concept "*Topic*") representing ranges of these properties with the goal to improve service discovery; (iii) "hidden" (hard-coded) knowledge embedded in the standard is translated into a set of rules in the *Meta Ontology* and is used in typical inferencing tasks.

To describe the process flow we combine the results of the *OWL-S process ontology* with the experiences from the business process modelling by taking into the consideration the specificities of the E-Government domain. Similarly to the *OWL-S process ontology*, we distinguish between the services and the control constructs. Services can be either atomic or composite services. For each service we define the standard set of attributes such a name, a description, etc. However, there are specific requirements concerning retraceability, realisation, security, cost etc. Therefore, we introduce the E-Government specific properties:

- each service is associated to the law it is based upon ("*hasReferencedBusinessRule*"). We note that it is very important to document the laws and regulations not only for the whole process but also for specific activities;
- each service is associated to the software component that implements it ("*hasReferencedSoftware*"). However, it is possible that the same description of the service is related to the different implementations. To inform the workflow engine about the software component that has to be invoked, it is necessary to model the decision attribute ("*hasDecisionAttribute*");
- it is necessary to assign security levels to each service ("*hasSecurityLevel*");
- information about cost and time restrictions can be also specified ("*hasCost*", "*hasTimeRestriction*", etc.).

Similarly to the *OWL-S process ontology*, services have the inputs ("*hasInput*") and output ("*hasOutput*"). The concepts "*Input*" and "*Output*" are defined as subconcepts of the concept "*Parameter*". Since some inputs have to be provided by the end-user the concept "*User-defined Input*" is defined as a specialisation of the concept "*Input*". To establish the equality between two parameters we introduce the symmetric property "*isEqualTo*".

Since it is required that inputs/outputs are defined in the domain ontology, we introduce the additional concept "*Reference*" due to two reasons: (i) a property may be attached to several domain concepts; (ii) a concept defined in the domain ontology may have many properties and only a subset of them is used as an input. In order to specify the context of the usage of a property and to select a subset of them, we introduce the properties "*hasConcept*" and "*hasProperty*" respectively. The range of these properties is the *Root* concept that is included in each KAON ontology. By using the meta-modelling facilities provided by the KAON management system, it is possible to reference any entity (i.e. a concept, a property or an

[124] http://www.cenorm.be/sh/mmi-dc

instance) defined in the domain ontology. Furthermore, to name a parameter we define the attribute "*hasName*".

The description of all entities of the *Meta Ontology* (and their semantics) is given in [123].

Changes

For the ontology evolution we defined the set of ontology changes that includes all elementary changes (e.g. *AddConcept*, see Table 1) and some more complex changes, the so-called composite changes (e.g. *MoveConcept*, see Table 4). However, this granularity level should be extended in order to enable a better management of changes in a service description. For example, to make the service *s1* a predecessor of the service *s2*, the manager needs to apply a *list* of ontology changes that connects outputs of *s1* to the corresponding inputs of *s2*. We cannot expect that she spends time finding, grouping and ordering the ontology changes to perform the desired update. In order to do that, she should be aware of the way of resolving a change; she should find out the right changes and order them in a right way. This activity is time consuming and error prone.

Therefore, managers require a method for expressing their needs in an easier, more exact and declarative manner. For them it would be more useful to know that they can connect two services rather than to know how it is realised. To resolve the above-mentioned problem, the intent of the changes has to be expressed on a more coarse level, with the intent of the change directly visible. Only in this way can managers focus on what has to be done, and not on how to do that.

To identify this new level of changes, we start from the *Meta Ontology*. The abstract, simplified model of this ontology is shown in Figure 72. For each service one can specify inputs, outputs, preconditions, postconditions, resources and business rules, other services that it either specialises or is connected with. Each of these entities can be updated by one of the meta-change transformations: add and remove (see section 2.4.1). A full set of changes can thus be defined by the cross product of the set of entities of the *Meta Ontology* and the set of meta-changes. A part of them[125] is shown in Table 15.

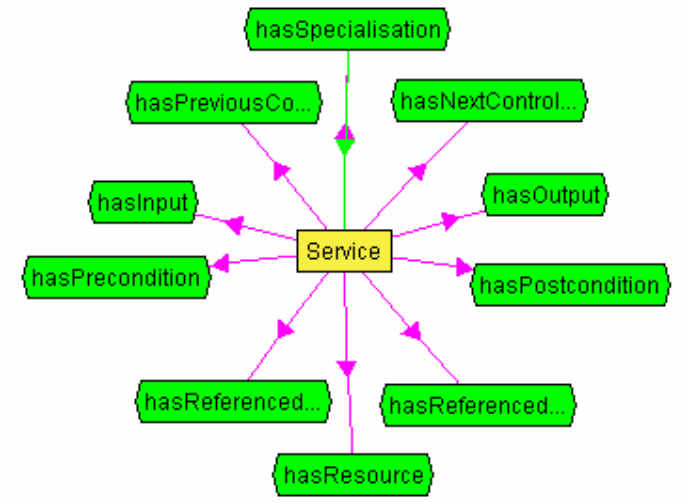


Figure 72. Abstract model of the Meta Ontology

[125] We focus here only on the entities that are important for the management. Other aspects, such as properties defined in the service profile, are ignored. Due to this abstraction of the *Meta Ontology*, only the most typical and most frequently occurring changes are shown, since they are relevant from the management point of view

Changes shown in Table 15 build the backbone of a semantic web service management system. They make the evolution of the semantic description of web services much easier, faster, more efficient, since they correspond to the "conceptual" operation that someone wants to apply without understanding the details (i.e. a set of ontology changes) that the management system has to perform.

Table 15. The taxonomy of changes of the semantic web service ontology

	Additive Changes	**Subtractive Changes**
Service	AddService	RemoveService
Input	AddServiceInput	RemoveServiceInput
Output	AddServiceOutput	RemoveServiceOutput
Precondition	AddServicePrecondition	RemoveServicePrecondition
Postcondition	AddServicePostcondition	RemoveServicePostcondition
Service Specialisation	AddServiceSpecialisation	RemoveServiceSpecialisation
Next Connection	AddServiceNextService	RemoveServiceNextService
Previous Connection	AddServicePreviousService	RemoveServicePreviousService
Business Rule	AddServiceBusinessRule	RemoveServiceBusinessRule
Software Component	AddServiceSoftware	RemoveServiceSoftware
Resource	AddServiceResource	RemoveServiceResource

Consistency

To define the consistency of the semantic web service ontologies we start from the ontology consistency definition that is given in section 2.3. Since ontologies that are used to describe semantic web services include other ontologies, we have to use the dependent and distributed ontology consistency definition (see Definition 41).

Furthermore, the *Meta Ontology* can be considered as the meta-level for the semantic web service description. Since the set of consistency constraints heavily depends on the underlying model, the semantics of the *Meta Ontology* defines a set of constraints that all service ontologies have to fulfil. In this section we discuss how the existing dependent ontology consistency definition has to be enriched in order to take into account the specificities of the *Meta Ontology*.

We introduce the following additional constraints:

- Service profile specific constraints:

- *Business knowledge specific constraints*
  C1: Each service has to have a reference to at least one business rule.
- *Tracebility*
  C2: Each service has to have at least one resource that controls its execution.
- *Applicability*
  C3: Each service has to have at least one software component attached to it that implements it.

*- **Service process specific constraints***:

- *Completeness*

  C4: Each service has to have at least one input.

  C5: Each service has to have at least one output.

  C6: Each service input has to be either output of some other service or is specified by the end-user.

- *Satisfyability*

  C7: If the input of a service is the output of another service, then it has to be subsumed by this output.

  C8: If the input of a service subsumes the input of the next service, then its preconditions have to subsume the preconditions of the next one.

  C9: If two services are subsumed by the same service, then their preconditions have to be disjoint.

- *Uniqueness*

  C10: If a service specialises another service, one of its parameters (i.e. inputs, outputs, preconditions or postconditions) has to be different. The difference can be achieved either through the subsumption relation with the corresponding counterpart or by introducing a new one.

- *Well-formedness*

  C11: Inputs, outputs, preconditions and postconditions have to be from the domain ontology.

- ***Domain specific constraints***:

  - *Structural dependency*

    C12: Any specialisation of the service *S1* must always be a predecessor of any specialisation of the service *S2*, where *S1* and *S2* are two activities defined in the *Meta Ontology* and their order is given in advance (i.e. *S1* precedes *S2*).

It is worth mentioning that only consistency constraints *C1* and *C12* are domain-dependent. Whereas *C1* has a reference to the *Business Rules Ontology*, *C12* is related to the generic schema for the services and specifies the obligatory sequence among activities. In the E-Government domain, *C1* requires that each service is related to a law and each law is referenced, respectively. *C12* states that the structure of *Service Ontologies* must follow predefined rules, so that a service specialising an application service has to precede specialisation of a verification service.

We give short interpretations of some constraints from the change management point of view:

- *C1* enables to find the corresponding service if a law is changed;
- *C6* ensures that a change in an output of an activity is propagated to the inputs of successor activities and vice versa;
- *C8* prohibits the changes, which lead to non-optimal service reconfiguration. For example, if the preconditions for an activity include a constraint that a person has to be older than 18, the preconditions of the next activity cannot be that a person has to be older than 16.

Note that each of these constraints is formally defined and is automatically verified against service descriptions. For example, the first constraint *C1* is formalised as:

$$\forall x \exists y(x,"Service") \in H_c^* \wedge (y,"BusinessRule") \in H_c^* \wedge$$
$$x \in domain("hasReferencedBusinessRule") \wedge$$
$$y \in range("hasReferencedBusinessRule")$$

where *Hc**, *P*, *domain*, *range* are defined in Definition 3. The property "*hasReferencedBusinessRule*", the concept "*Service*" and the concept "*Business Rule*" are entities of the *Meta Ontology*.

Finally, we define the consistency of the semantic web services in the following way:

Definition 43 ***Semantic Web Service Consistency***: A semantic web service is a consistent service if its description is dependent ontology consistent and the additional constraints (*C1-C12*) are fulfilled.

Note that a change in the business logic does not cause any ontology inconsistency. Regarding the E-Government domain, after the removal of a single input of an activity, the ontology consistency is still fulfilled. However, this change provokes the semantic web service inconsistency, since the consistency constraint *C4* is not satisfied. Therefore, the extension of the consistency definition is a prerequisite for the management of the semantic web services.

Since semantic web services must be compliant with the set of semantic web service consistency constraints, in the rest of this section we discuss how to preserve the consistency. We firstly define a procedure that informs managers about changes in the business rules that provoke some inconsistencies. Thereafter we introduce the procedures for ensuring the semantic web service consistency.

Propagating Changes from Business Rules to Services

The basic requirement for a management system is that it has to be simple, correct and usable for managers. Note that they are responsible for keeping semantic web services up-to-date and do not need to be experienced ontology engineers. Thus, a management system must provide capabilities for the automatic identification of problems in the (description of the) semantic web services. Moreover, it must assist the managers in defining solutions for resolving them.

In this section we define the procedure for finding the "weak places" in the description of the semantic web services by considering the changes in the business rules and their impact on the consistency. The procedure is focused on discovering inconsistencies in a semantic web service description whose repairing improves the agreement of this ontology with the business rules.

When we designed this support, we assumed that the update would be only a partially automated process rather than a fully automated process. We do not want to update web services automatically, but rather to notify the managers about problems. For example, the manager should be informed about a new amendment. However, the realisation of this amendment must not be automated since it requires a lot of domain knowledge that cannot be formally represented in the *Legal Ontology* and is a result of experiences. Therefore, our system only makes recommendations about a potential resolution of a problem. For example, a new amendment might be realised through the specialisation of a web service that implements the law for which this amendment is defined.

Obviously, the information about the business rule that is implemented by a service is very important for the change management. It means that the consistency can be achieved only by referring to this knowledge. This was one of the reasons for defining the *Meta Ontology*.

The procedure for propagating changes from business rules to web services is based on the evolution between dependent and distributed ontologies since we assume that the *Business Rule Ontology* is reused in the *Meta Ontology* through the replication (see section 5.3.2). In the E-Government domain the physical distribution is very important since E-Government services must follow federal, state and local laws that are defined externally. Note that a *Service Ontology* might reuse the *Meta Ontology* either through inclusion or replication, which depends whether they are within the same system or not.

The procedure consists of four steps:

1. ***Checking actuality of the Business Rules Ontology*** – Since each ontology has a version number associated with it that is incremented each time when the ontology is changed, checking the equivalence of the original of the *Business Rules Ontology* and the replica can be done by a simple comparison of the number.
2. ***Extracting Deltas*** – After determining that the included *Business Rules Ontology* needs to be updated, the evolution log (see section 3.2.4) of this ontology is accessed and deltas are extracted (see section 5.3.2). For example, after the addition of the new amendment *A7* in the *Legal Ontology* as the adaptation of the paragraph *P2*, the delta will contain changes shown in Figure 73.

```
<a:AddInstanceOf rdf:ID="i-1079962974979-1202219624"
   a:has_referenceConcept="Legal#Amendment"
   a:has_referenceInstance="Legal#A7"
   a:inOIModel="file:/C:/ontoGov/Legal"
   a:version="10">
...
</a:AddInstanceOf>
<a:AddPropertyInstance rdf:ID="i-1079962991620-1904187797"
   a:has_referenceProperty="Legal#modifies"
   a:has_referenceSourceInstance="Legal#A7"
   a:has_referenceTargetInstance="Legal#P2"
...
<a:has_previousChange rdf:resource="#i-1079962974979-1202219624"/>
</a:AddPropertyInstance>
```

Figure 73. A part of a log of the Legal Ontology

3. ***Analysis of changes*** – Each performed change is analysed, in order to find semantic web services that have to be updated. We distinguish between the addition and the deletion of an entity from the *Business Rule Ontology*. Removals can be resolved directly by applying the existing ontology evolution system since it ensures the consistency by generating addition changes (see chapter 4). However, the addition requires an additional effort that depends on the structure of the *Business Rules Ontology*. Here we describe how this problem is resolved in the E-Government domain by considering the *Legal Ontology*. We analyse the addition of a new amendment. The goal is to find services that realise the law related to this amendment and to order them in an appropriate way. Since each service is referred to a law/chapter/paragraph/article, the corresponding service can be easily found. In case there are several services referring to the given law (e.g. through a paragraph or an amendment), they are ranked according to the semantic similarity that is based on calculating the distance between two entities in the hierarchy, we proposed in [133]. Currently, we are developing a search module that uses NLP-methods to calculate similarity between two texts, no matter the text is the description of an amendment or paragraph.
4. ***Making recommendation***: In order to make recommendations how to adapt the up-to-date semantic web services we define the *Lifecycle Ontology* [123]. It describes design

decisions and their relationship to affected parts of the service as well as to the requirements that motivate the decisions [67]. Since the *Lifecycle Ontology* is a description of the service design process, which clarifies which design decisions were taken for which reasons, it proves to be valuable for further development and maintenance. During ongoing development, it helps the managers to avoid pursuing unpromising design alternatives repeatedly, but it also facilitates maintenance by improving the understandability of the service design.

Propagating Changes Within Services

The key process in the change management is the resolution of the changes triggered by the procedure described in the previous section. It has to guarantee that a change is correctly propagated and that no inconsistency is left in the system. If this was left to the managers, the management process would be too error-prone and time consuming – it is unrealistic to expect that humans will be able to comprehend all the existing services and interdependencies between them. For example, in the E-Government domain an unforeseen and uncorrected inconsistency is one of the most common problems.

Therefore, the change management has to be supported by a tool that improves the efficiency and the quality of this process. In order to develop such a tool, the problem has to be formulated in terms of a formal model. Since our approach is based on the semantic description of services, the formal model requires the specification of the semantics of changes that can be applied to the semantic web services.

For each change shown in Table 15 it is required to specify (see section 4.2):

(i) necessary *preconditions*;
(ii) sufficient *postconditions*;
(iii) possible *actions*.

The *preconditions* of a change are a set of assertions that must be true to be able to apply the change. For example, the preconditions for the change *AddServiceSpecialisation(S1,S2)*, which results in the specialisation of the service *S1* in the service *S2*, are: (i) *S1* and *S2* are different services; (ii) *S2* is not an indirect parent (through the inheritance hierarchy) of *S1*; (iii) *S2* is not already defined as a specialisation of *S1*; (iv) for each input/output/preconditions/postconditions of *S1* there is a corresponding element in *S2* that is subsumed by the original[126].

The *postconditions* of a change are a set of assertions that must be true after applying the change and it describes the result of the change. For example, the removal of a service results in the fact that this service is not in this service ontology anymore.

The *actions* are additional changes that have to be generated in order to resolve the side effects of a change on other related entities. It means that each inconsistency problem is treated as a request for a new change, which can induce new problems that cause new changes and so on. An inconsistency arises when one of the semantic web service consistency constraints (see Definition 43) is corrupted. For example, the addition of a service will trigger the addition of an input for this service (i.e. *AddServiceInput* change) since the consistency constraint *C4* requires that each service has to have at least one input. To define the necessary actions for each change, we reuse the approach described in section 4.2.

Here we define the procedure for the *AddServiceInput(service, input)* change:

[126] Note that the first three preconditions are inherited from the *AddSubConcept* ontology change whereas the last one is specific for this particular change.

- *Preconditions* $-input \notin Inputs(service)$, where *Inputs* is a set of all inputs already defined for a service. This is in agreement with the single ontology consistency constraints that ensure the uniqueness of the definition.
- *Postconditions* $-input \in Inputs(service)$, which means that this input is defined for this service.
- Actions – *AddEqualsTo(input, x)*, where:

  $\exists s1 \quad service \in hasNext(s1) \wedge x \in Outputs(s1) \wedge (input, x) \in H_C^*$, where *hasNext*[127] is a property defined in the *Meta Ontology* for connecting services, *Inputs/Outputs* represent a set of inputs/outputs defined for a service and H_c^* is already defined in section 2.2.

A new input might corrupt the *C6* consistency constraint, since the inputs provided by the end-users are usually defined for the first service in the process flow. To resolve this problem, one has to specify that this input is provided by the output of the previous service. This can be realized as a request for a new change *AddEqualsTo*, which establishes the "*IsEqualsTo*" property between corresponding input/output parameters.

For example, according to the changes in a law, the driving licence verification activity requires fingerprint. This change causes the inconsistency since the new input hangs. The problem can be resolved by generating the additional change *AddEqualsTo* between the verification activity and its predecessor. This further induces a new output of the predecessor, i.e. application activity, which can potentially trigger other changes and so on.

Finally, it is important to note that any change in the domain ontology is resolved automatically by using the existing ontology evolution system. For example, let's consider that the domain ontology contains the concept "*Person*" and two of its specialisations: "*Child*" and "*Adult*". Since there is a special procedure for the passport issuance for the children (see section 7.3.2), this service is a specialisation of the standard service passport issuance service. The application service of the service for children requires an additional input (i.e. parent authorisation). The precondition for this application service is that it is required for a child. Let's now consider that the concept "*Child*" needs to be removed. The ontology evolution system will propagate this change to all ontologies that included the changing ontology. Therefore, the ontology describing the passport issuance procedure for children will also be informed about changes. Since, according to the ontology consistency definition (see section 2.3), undefined entities are not allowed, the request for the removal of the corresponding application service will be generated.

This example shows that the management of semantic web service descriptions heavily depends on the management support for the domain ontologies. We showed how our ontology evolution approach can be reused and extended to take into account the specificities of semantic web service description.

7.3.4 Related Work

Although the research related to Web Services has drastically increased recently, there are very few approaches that cope with the changes in the process flow of a web service. The management approaches are mainly focused on the composition of a web service from scratch and neglect the problem of the continual improvement of the service. The change

[127] s2∈ hasNext(s1) means that *s2* is one of the services that *s1* precedes through one control construct.

management approaches are mainly focused on re-implementing some software modules [57]. We found two reasons for such behaviour:

1. Since the technology is rather new, the real challenges for the change management are still to come. Indeed, in the workflow community, from which web services are transferring a lot of experiences, the workflow maintenance is a well-researched topic;
2. The description of web services lacks a conceptual level on which the reasoning about a compositional model, including the reasons and the methods for its reconfiguration, will be possible. As we have already mentioned, the emerging semantic web services approaches introduce such a level and we give here a short overview of their achievement in the (re)composition.

Workflow

The workflow community has recently paid attention to configurable or extensible workflow systems, which present some overlaps with our ideas. For example, the work on flexible workflows has focused on the dynamic process modification [58]. In this publication, workflow changes are specified by transformation rules composed of a source schema, a destination schema and of conditions. The workflow system checks for parts of the process that are isomorphic with the source schema and replaces them with the destination schema for all instances for which the conditions are satisfied. However, the workflow schema contains fewer primitives than an ontology so that this approach is much less comprehensive then ours. Moreover, the change in the business policy is not treated at all.

The most similar to our approach is the work related to the workflow evolution [13]. This paper defines a minimal, complete and consistent set of modification primitives that allow modifications of workflow schemata. The authors introduce the taxonomy of policies to manage the evolution of running instances when the corresponding workflow schema is modified. However, the authors are focused on the dynamic workflow evolution, which is not the focus of our work, as we have mentioned in the section 7.3.1.

Semantic Web Services

Recently, the approaches for the composition of semantic web services have emerged drastically. We discuss only the most relevant to our approach.

In [61] a framework for the interactive service composition is presented, where the system assists users in constructing a computational pathway by exploiting the semantic description of services. Given the computational pathway and the user's task description (i.e. a set of initial inputs and expected results), the system performs a set of checks (e.g. are all the expected results produced, are all the needed input data provided) in order to ensure the consistency of the resulted model. The checks used in this approach can be seen as a subset of the constraints we defined for ensuring the consistency. Moreover, since we derive the constraints from the ontology model behind the semantic web services, we can guarantee the completeness and the consistent propagation of the changes.

In [149] the authors present a prototype for dynamic binding of Web Services for the abstract specification of business integration flows using a constraint-based semantic-discovery mechanism. They provide a way of modelling and accommodating scoped constraints and inter-service dependencies within a process flow while dynamically binding services. The result is a system that allows people to focus on creating appropriate high-level flows, while providing a robust and adaptable runtime. Similarly to our approach they contend that the selection of Web services for a step in a process flow is, often, not a stand-alone operation, as there may be dependencies on the previously chosen services for the process. They introduce

two types of dependencies: description-based and domain constraints whereas both of them can be easily mapped into our business-knowledge specific constraints that ensure the meaningful order between services in a flow. Additionally we provide process specific constraints that ensure the consistency of the process flow.

Next, there are several approaches for the automatic composition of semantic web services [44], [88] that drive the design at a conceptual level in order to guarantee its correctness and to avoid inconsistencies among its internal components. In that context, our approach can be seen as an automatic re-composition of a service driven by the constraints derived from the business environment, domain knowledge and internal structure of a service.

Finally, the main difference between our approach and all the related researches is that we base our management framework on the systematic evolution of the model that underlines semantic web services (i.e. several dependent and distributed ontologies). It enables us to be predictive in the management (i.e. we can reason about the consequences of changes in the system) and to expand the framework whereas the consistency of the managed system is ensured, easily.

7.3.5 Conclusion

In this section we presented an approach for the management of changes in semantic web services. The approach is based on our work on the ontology evolution that is elaborated in this thesis. As a case study we considered the E-Government domain since E-Government services are under the continual adaptation to the political goals of a government and to the needs of the people. Up to now, the changes have been initiated and propagated manually, which causes a lot of errors and redundant steps in the change management process. Our approach enables the automation of the change propagation process and ensures its consistent execution since it is based on a formal framework for coping with changes. Consequently, we can reason about the change management process, making it very flexible and efficient.

The proposed approach can be extended by suggesting the changes that can improve services. This can be done (i) by monitoring the execution of E-Government services (e.g. the activity that causes the delay is a candidate for optimisation) and/or (ii) by taking into account the end-users' complaints (e.g. end-users might not be satisfied with the quality of services since they have to supply the same information several times).

.

8 Conclusion

Due to the ever increasing complexity, heterogeneity and physical distribution of the business, the importance of ontologies for the conceptualisation of the business applications becomes inevitable. It is especially important for the recently increased research in the Semantic Web and Web Services that enable publishing business processes on the Web.

However, the frequently changing business context implies the need to cope with changes in ontology-based business applications in a more systematic way. Firstly, different causes of changes (e.g. changes in the business environment, user's preferences, internal processes, etc.) have to be uniformly represented, in order to enable their efficient processing. Secondly, the changes have to be consistently resolved in the application, and their effects have to be propagated to all dependent business systems. Moreover, in order to control the resolution of the changes (e.g. the identification and overcoming of undesired changes), the responsible persons have to be able to make appropriate decisions. Finally, the continual business reengineering requires an automatic discovery of new changes by analysing the manner in which the application is used (e.g. the detection of trends in the users' behaviour). In order to fulfil these requirements efficiently, the managing of the changes in the ontology-based application has to be performed on the level of ontologies themselves. Therefore, the need for an efficient approach to the management of the changes in an ontology (e.g. ontology evolution) is obvious. In this thesis, we presented such an approach.

By analysing typical problems that arise during the ontology development, we formulated the following set of design requirements for an ontology evolution system:

3. Ontology evolution has to (i) enable the handling of the given ontology changes and (ii) ensure the consistency of the underlying ontology and all dependent artefacts;

4. Ontology evolution should be supervised allowing the user to manage changes more easily;

5. Ontology evolution should offer advice to the user for a continual ontology refinement.

Based on the analysis of these requirements, we defined a process-oriented ontology evolution approach that manages changes in six steps:

(i) The process starts with capturing changes either from explicit requirements or from the result of change discovery methods;

(ii) Next, the changes are represented formally and explicitly;

(iii) The semantics of the change phase prevents inconsistencies by computing the additional changes that guarantee the transition of the ontology into a consistent state;

(iv) In the change propagation phase, all dependent artefacts (ontology instances on the Web, dependent ontologies and application programmes using the changed ontology) are updated;

(v) During the change implementation phase, the required and induced changes are applied to the ontology in a transactional manner;

(vi) In the change validation phase, the user evaluates the results and restarts the cycle if necessary.

A special attention in developing and implementing the proposed process-oriented ontology evolution approach was paid to the usability issue, by taking into account the users with the different background regarding the ontology management. Indeed, the recent, widespread expansion of the ontology-based research leads to the involvement of the users with various experiences, skills and requirements in the usage/development of ontologies. Since getting up-to-date ontologies is one of the crucial bottlenecks in the management of the ontology-based applications, our primary intention was and remains to enable the customisation of the ontology-evolution process to the current need (i.e. knowledge, preferences) of the user. In that way, not only do we enable an inexperienced user to understand and apply the main concepts of the approach easily, but we also allow an advanced user to satisfy a very complicated request for a change as much comfortable as possible. For example, we categorized ontology changes into elementary, composite and complex changes, in order to enable the users to process their different requests efficiently. A novice will be focused only on elementary and composite changes that are built in the system. However, an expert can benefit from the possibility to define a complex request for a change declaratively, without defining how it has to be realized. Another example are the evolution strategies that enable an expert to control the effects of ontology changes. Moreover, the advanced evolution strategies are suitable for the novices, since they can customise the change resolution at a higher level of abstraction (e.g. the minimal number of induced changes) without considering each particular resolution point.

Another aspect we considered is the applicability of the proposed approach on the Semantic Web, which, as a semantic extension of the current Web, opens the possibility to develop large and distributed ontology-based applications. The vision of the Semantic Web can only be realized through the proliferation of well-known ontologies describing different domains. To enable the interoperability in the Semantic Web, it will be necessary to break these ontologies down into smaller, well-focused units that may be reused. In doing so, the ontology engineer should reuse as many definitions as possible from the existing ontologies to speed up the engineering and to enable the interoperability. Since ontologies are rarely static, an infrastructure for the management of ontology changes, taking into account dependencies between ontologies, is needed. As an answer to this challenge, in this thesis, we developed a multi-dimensional approach for the ontology evolution that takes into account the number of evolving ontologies and their physical distribution.

Another issue we found important for the widespread acceptance of the approach is its cyclic nature that provides a platform for the continual learning. In today's business, an enormous quantity of data is produced, partially consumed and forgotten in corporate databases. In the ontology-based applications, all this data can be integrated on the semantic (ontology) level and applied for the improvement of the ontology. In that way, an ontology is not a passive model that structures a domain, but an active component that automatically reflects changes in the domain it models. We defined a comprehensive framework for the change discovery centred around the Log Ontology, which tracks the users' behaviour, and we proposed several

heuristics for the discovery of changes. In that way, our approach goes beyond a standard change management process; rather it is a continual improvement process.

A substantial part of the results from this thesis is a system for the ontology evolution, implemented in the well known[128] KAON ontology engineering framework. Although the system reflects our intention to highly increase the usability, its primary design decision was related to the possibility to work with large datasets. Indeed, moving ontologies into a large real-world context requires the scalability of the platforms they are dealing with. This is probably the most critical issue in the whole research related to ontologies. Can the approaches scale when their application data increases drastically? In the Semantic Web environment, such a data explosion is inevitable. We did our best in tackling the complexity problem in the presented research. The evaluation studies showed that we moved in the right direction. We performed several evaluations on the large datasets, including the 600MB large MEDLINE datasets.

Although a lot of the research has been successfully performed in the thesis, there are still open issues that can be resolved in the area of the ontology evolution. We would like to mention the most important topics from our perspective:

- *Language-independent ontology evolution*: Our ontology evolution approach is developed as much as possible independently of the underlying ontology language. However, the set of consistency constraints and the set of elementary ontology changes heavily depend on the KAON ontology language. Since the OWL ontology language is going to be a standard language for representing ontologies on the Semantic Web, the adaptation of the ontology evolution approach to the semantics of the OWL ontology language would be useful. It requires the adjustment of the ontology consistency definition to the formal semantics of the OWL ontology model, new ontology changes, since the OWL ontology language is richer than the KAON ontology language, as well as the explicit specification of semantics of ontology changes.

  Moreover, in order to achieve the interoperability (i.e. the reuse) between the evolving ontologies developed in different ontology languages, the ontology evolution should be specified with enough generality that the other ontology evolution systems working with different ontology languages can benefit from our work. One way to accomplish this is to model all aspects of the ontology evolution declaratively. This abstraction may assist in the design of a language independent ontology evolution system;

- *Request specification*: In the future, our approach for the specification of a request for a change can be extended by defining a declarative language for this specification. It will allow expressing the ontology changes and constraints in a single framework, and, thus, will allow to reason about interactions between the two. This language will differ from the existing ontology query languages, which are only used for the retrieval of the data from an ontology. It will extend these languages by incorporating the modifications, as well;

- *Partial reuse*: Our approach for the ontology reuse is currently limited to including entire models rather than including subsets. By allowing the inclusion of a part of a model, it will be much more difficult to ensure the consistency of the including ontology, since it is not clear which additional elements from the included ontology must be included. Moreover, lifting these constraints will have a significant impact on the evolution of dependent ontologies, since it will be difficult to conclude which changes have to be broadcast to which dependent ontology;

[128] Till June 2004, there was 11.649 downloads.

- *Ontology dependency*: The future work can be directed towards providing more ways for working with multiple ontologies. Currently, we consider only the ontology reuse. However, there are also other dependency forms, such as ontology mapping, ontology merging, ontology alignment and ontology integration. For example, the ontology mapping relates similar (according to some metric) concepts and relations from different sources to each other; the ontology merging creates a new ontology from two or more existing ontologies with overlapping parts. Each of these dependency forms puts different requirements on the evolution between dependent ontologies. Some of them can be resolved by introducing a special meta-ontology that captures relationships between entities from different ontologies. For example, to set up the mapping between ontologies, the mapping ontology might be defined. This ontology should contain the "equal" property that can be used for establishing equivalence between concepts from different ontologies. To support the evolution of the instantiation of this ontology, the ontology evolution approach has to be extended in two ways. Firstly, the set of consistency constraints has to be extended by taking into account the semantics of the mapping ontology. Secondly, the set of changes has to be extended with the more complex changes that can be applied to these mappings. Finally, the evolution support has to take into account that concepts and properties from the ontologies between which the mapping is established are considered as instances in the ontology that describes these mappings;
- *Change discovery from dependent ontologies*: Besides the implicit feedback of end-users captured in the usage log, re-occurring extensions of an included ontology in many including ontologies are another force driving the usage-driven ontology updates. Therefore, our approach for the usage-driven ontology evolution can be extended by taking into account the usage of an ontology in other ontologies that reuse it. Since the extensions of an ontology are driven by different ontology engineers, dependent, distributed ontologies evolve in different directions. It is necessary to analyse the evolution logs of these ontologies, and to discover changes that can be applied to the included ontology.

Ontology evolution is a promising research area since evolution over time is an essential requirement for successful application of ontologies. New methods and tools to support this complex task can help for easy and consistent modification and thus can enable the widespread use of ontologies in industrial and academic applications. This thesis is a step towards achieving aforementioned goal.

9 References

[1] N. Adam, F. Artigas, V. Atluri, S. Chun, S. Colbert, M. Degeratu, A. Ebeid, V. Hatzivassiloglou, R. Holowczak, O. Marcopolus, P. Mazzoleni, W. Rayner, *E-government: Human centered systems for business services*, In Proceedings of the 1st National Conference on Digital Government, Los Angeles, CA, pp. 48–55, 2001.

[2] E. Allen, T. Khoshgoftaar, Y. Chen, *Measuring coupling and cohesion of software modules: an information - theory approach*, In Proceedings of the Seventh International Software Metrics Symposium (METRICS 2001), London, pp. 124-134, 2001.

[3] J. Banerjee, W. Kim, H.J. Kim, H. Korth, *Semantics and implementation of schema evolution in object-oriented databases*, In Proceedings of the Annual Conference on Management of Data (ACM SIGMOD 16(3)), San Francisco, pp. 311-322, 1987.

[4] V. Barker, D. O'Connor, J. Bachant, E. Soloway, *Expert Systems for Configuration at Digital: XCON and Beyond*, Communications of the ACM, Volume 32, Number 3, pp. 298-312, 1989.

[5] B. Berendt, A. Hotho, G. Stumme, *Towards semantic web mining*, In Proceedings of the 1st International Semantic Web Conference (ISWC 2002), Sardinia, Italia, LNCS 2342, pp. 264-278, 2002.

[6] B. Berendt, M. Spiliopoulou, *Analysis of navigation behaviour in web sites integrating multiple information systems*, The VLDB Journal, Volume 9, pp. 56-75, 2000.

[7] M. Bhide, P. Deoasee, A. Katkar, A. Panchbudhe, and K. Ramamritham, *Adaptive push-pull: disseminating dynamic Web data*, IEEE Transaction on Computers, Volume 51, Number 6, pp. 652-668, June 2002.

[8] R. Botafogo, E. Rivlin, B. Shneiderman, *Structural analysis of hypertexts: identifying hierarchies and useful metrics*, ACM Transactions on Office Information Systems, Volume 10, Number 2, pp. 142-180, 1992.

[9] T. Berners-Lee, *XML 2000 – Semantic Web talk*, http//www.w3.org/2000/Talks/1206-xml2k-tbl/slide10-0.html, 2000.

[10] P. Breche, *Advanced principles for changing schemas of object databases*, In Proceedings of the 8th Conference on Advanced Information Systems Engineering (CAiSE'96), Heraklion, Crete, Greece, LNCS 1080, pp. 476-495, 1996.

[11] P. Breche, M. Woerner, *How to remove a class in an ODBS*, In Proceedings of the 2nd International Conference on Applications of Databases (ADBS'95), San Jose, California, pp. 235-246, 1995.

[12] A. Bultmann, J. Kuipers, F. van Harmelem, *Maintenance of KBS's by Domain Experts: The Holy Grail in Practice*, In Proceedings of the 13th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE 2000), New Orleans, Louisiana, USA, LNCS 1821, pp. 139-148, 2000.

[13] F. Casati, S. Ceri, B. Pernici, G. Pozzi, *Workflow evolution*, In Proceedings of 15 th International Conference on Conceptual Modelling (ER'96), Cottbus, Germany, pp. 438-455, 1996.

[14] W. J.Clancey, *Heuristic classification*, Artificial Intelligence, Volume 27, Number 3, pp. 289-50, 1985.

[15] K. Claypool, *Managing schema change in a heterogeneous environment*, PhD Thesis, URN ETD-0617102-213436, Science Department, Worcester Polytechnic Institute, 2002.

[16] K. Claypool, E. Rundensteiner, *SERF: transforming your database*, IEEE Bulletin - Special Issue on Database Transformation Technology, pp. 19-24, Mart 1999.

[17] G. Cobena, *Change management of semi-structured data on the Web*, Inria, TU-0789, PhD Thesis, 2003.

[18] F. Coenen, T. Bench-Capon, *Maintenance of knowledge-based systems*, Academic Press, the A.P.I.C. Series, Number 40, ISBN: 0-12-178120-8, 1993.

[19] D. Connolly, F. van Harmelen, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein, *DAML+OIL reference description*, http://www.w3.org/TR/daml+oil-reference, 2001.

[20] N. Cullot, C. Parent, S. Spaccapietra, C. Vangenot, *Ontologies: A contribution to the DL/DB debate*, In Proceedings of the 1st International Workshop on the Semantic Web and Databases (SWDB 2003), 29th International Conference on Very Large Data Bases, Berlin Germany, pp. 109-129, 2003.

[21] A. Das, W. Wu, D. McGuinness, *Industrial strength ontology management*, The Emerging Semantic Web, Selected papers from the 1st Semantic web working symposium, Stanford University, California, USA, Frontiers in Artificial Intelligence and Applications, Volume 75, IOS press, ISBN 1-58603-255-0, 2002.

[22] H. Dai , B. Mobasher, *A road map to more effective web personalization: Integrating domain knowledge with web usage mining*, In Proceedings of the International Conference on Internet Computing 2003 (IC'03), Las Vegas, Nevada, pp. 58-64, 2003.

[23] DAML + OIL - http://www.daml.org/2001/03/reference.html

[24] J. Davies, A. Duke, Y. Sure, *OntoShare – A knowledge management environment for virtual communities of practice*, In Proceedings of the 2nd International Conference on Knowledge Capture (K-CAP2003), Florida, USA, pp. 20-27, 2003.

[25] C. Delcourt, R. Zicari, *The design of an integrity consistency checker (ICC) for an object-oriented database system*, In Proceedings of European Conference for Object-Oriented Programming (ECOOP'91), Geneva, Switzerland, LNCS 512, pp. 97-117, 1991.

[26] S. Decker, M. Erdmann, D. Fensel, R. Studer, *Ontobroker: Ontology based access to distributed and semi-structured information*, Meersman, R. et al. (Eds.), Database Semantics: Semantic Issues in Multimedia Systems, Kluwer Academic Publisher, pp. 351-369, 1999.

[27] A. J. Duineveld, R. Stoter, M.R. Weiden, B. Kenepa, V.R. Benjamins, *Wondertools? A comparative study of ontological engineering tools*, International Journal of Human-Computer Studies, Volume 52, Number 6, pp. 1111-1133, 2000.

[28] M. Ehrig, A. Maedche, *Ontology-focused crawling of Web documents*, In Proceedings of the Symposium on Applied Computing 2003 (SAC 2003), Melbourne, Florida, USA, ACM, pp. 1174-1178, 2003.

[29] M. Erdmann, *Ontologien zur konzeptuellen modellierung der semantik von XML*, PhD Thesis, University of Karlsruhe, Hamburg: Books on Demand, ISBN 3-8311-2635-6, 2001.

[30] A. Farquhar, R. Fikes, and J. Rice, *The ontolingua server: Tools for collaborative ontology construction*, Technical Report, Stanford KSL 96-26, September 1996.

[31] D. Fensel, J.A. Hendler, H. Lieberman, W. Wahlster (Eds.), *Spinning the Semantic Web: Bringing the World Wide Web to its full potential*, MIT Press 2003, ISBN 0-262-06232-1, 2003.

[32] D Fensel, C. Bussler, *The Web Service Modelling Framework WSMF*, In Electronic Commerce Research and Application, Volume 1, Number 2, pp. 113-137, 2002.

[33] D. Fensel, I. Horrocks, F. van Harmelen, S. Decker, M. Erdmann, M. Klein, *OIL in a nutshell*, In Proceedings of 12th International Conference on Knowledge Engineering and Knowledge Management (EKAW 2000), Juan-les-Pins, France, LNCS 1937, pp. 1-16, 2000.

[34] D. Fensel, *Ontologies: dynamics networks of meaning*, In Proceedings of the 1st Semantic web working symposium, Stanford, CA, USA, 2001.

[35] M. Fernandez-Lopez, A. Gomez-Perez, J.P. Sierra, A.P. Sierra, *Building a chemical ontology using methontology and the ontology design environment*, IEEE Intelligent Systems, Volume 14, Number 1, pp.37-46, 1999.

[36] F. Ferrandina, S.E. Lautemann*, An integrated approach to schema evolution for object databases*, In Proceedings of the Internation Conference on Object Oriented Information Systems (OOIS 1996), London, UK, pp. 280-294, 1996.

[37] E. Franconi, F. Grandi, and F. Mandreoli, *A semantic approach for schema evolution and versioning in object-oriented databases*, Computational Logic 2000, pp. 1048-1062, 2000.

[38] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, *Refactoring: improving the design of existing code*, Addison-Wesley, ISBN: 0201485672, 1999.

[39] N. Foo, *Ontology revision*, In Proceedings of the 3rd International Conference on Conceptual Structures (ICCS'95), Santa Cruz, CA, USA , LNCS 54, pp. 1-14, 1995.

[40] A. Gangemi, A. Prisco, M.T. Sagri, G. Steve, D. Tiscornia, *Some ontological tools to support legal regulatory compliance, with a case study*, Workshop on Regulatory Ontologies and the Modelling of Complaint Regulations, Part of the International Federated Conferences (OTM'03), Catania, Sicily, Italy, LNCS, Volume 2889, pp. 607-620, 2003.

[41] B. Ganter, R. Wille, *Formal concept analysis - mathematical foundations*, Springer Verlag, ISBN: 3540627715, 1999.

[42] D. Georgakopoulos, H. Schuster, D. Baker, A. Cichocki, *Managing escalation of collaboration processes in crisis mitigation situations*, In Proceedings of 16th International Conference on Data Engineering (ICDE 2000), San Diego, CA, USA, pp. 45-56, 2000.

[43] Y. Gil, M. Tallis, *A Script-Based Approach to Modifying Knowledge Bases*, In Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97), Providence, RI, pp. 377-383, 1997.

[44] A. Gomez-Perez, R. Gonzalez-Cabero, M. Lama, *A framework for design and composition of Semantic Web services*, First International Semantic Web Services Symposium, 2004 AAAI Spring Symposium Series, ISBN 1-57735-198-3, pp. 113-120, 2004.

[45] A. Gomez-Perez, M. Fernandez-Lopez, O. Corcho, *Ontological engineering: with examples from the areas of knowledge management, e-commerce and the Semantic Web*, Springer Verlag London Ltd., ISBN: 1-85233-551-3, 2003.

[46] A. Gomez-Perez, *Ontological engineering: A state of the art*, Expert Update, Volume 2, Number 3, pp. 33-43, 1999.

[47] A.Gomez-Perez, V. Richard Benjamins, *Applications of ontologies and poblem-solving methods*, AI Magazine, Volume 20, Number 1, pp. 119-122, 1999.

[48] T. Gruber, *A translation approach to portable ontology specifications*, Knowledge Acquisition, An International Journal of Knowledge Acquisition for Knowledge-Based Systems, Volume 5, Number 2, pp.199-220, 1993.

[49] N. Guarino, C. Welty, *Identity, unity and individuality: Towards a formal toolkit for ontological analysis*, In Proceedings of the 14th European Conference on Artificial Intelligence (ECAI 2000), Amsterdam, IOS Press, pp. 219-223, 2000.

[50] C. Hardless, R. Lindgren, U. Nulden, K., Pessi, *The evolution of knowledge management system need to be managed*, Journal of Knowledge Management Practice, Volume 3, 2000.

[51] F. van Harmelen, I. Horrocks, *FAQs on OIL: the Ontology Inference Layer*, IEEE Intelligent Systems, Trends and Controversies, Volume 15, Number 6, pp. 69-72, November/December 2000.

[52] F. Hayes-Roth, D.A. Waterman, D. B. Lenat, *Building expert systems*, Addison-Wesley, ISBN: 0-201-10686-8, 1983.

[53] J. Heflin, *Towards the Semantic Web: Knowledge representation in a dynamic, distributed environment*, Ph.D. Thesis, University of Maryland, College Park, http://www.cse.lehigh.edu/~heflin/pubs/heflin-thesis-orig.ps.gz, 2001.

[54] J. Heflin, J. Hendler, *Dynamic ontologies on the Web*, In Proceedings of Seventeenth National Conference on Artificial Intelligence (AAAI-2000), Menlo Park, CA, AAAI/MIT Press, Cambridge, MA, pp. 443-449, 2000.

[55] W. Huersch, *Maintaining consistency and behaviour of object-oriented systems during evolution*, In Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '97), ACM SIGPLAN Notices, Volume 32, Number 10, pp. 1-21, 1997.

[56] IEEE 90, Institute of Electrical and Electronics Engineers, IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. New York, NY, 1990.

[57] M. Janssen, R. Wagenaar, *An analysis of a shared services centre in e-government, system sciences*, In Proceedings of the 37th Annual Hawaii International Conference, Big Island, HI, USA, pp.124-133, 2004.

[58] G. Joeris, O. Herzog, *Managing evolving workflow specifications*, In Proceedings of the 3rd IFCIS International Conference on Cooperative Information Systems (CoopIS'98), New York, pp. 310-319, 1998.

[59] J. Kephart, D. Chess, *The Vision of Autonomic Computing*, IEEE Computer, pp. 41-50, January 2003.

[60] J. I. Kiger, *The Depth/Breadth Trade-Off in the Design of Menu-Driven User Interfaces*, International Journal of Man-Machine Studies, Volume 20, Number 2, pp. 201-213, 1984.

[61] J. Kim, Y. Gil, *Towards Interactive Composition of Semantic Web Services*, First International Semantic Web Services Symposium, 2004 AAAI Spring Symposium Series, ISBN 1-57735-198-3, pp. 100-107, 2004.

[62] R. Kimball, R. Merz, *The data webhouse toolkit: building the web-enabled data warehouse*, John Wiley & Sons, ISBN: 0471376809, 2000.

[63] M. Klein, *Versioning of distributed ontologies*, available as Deliverable D20 V1.1, EU/IST Project WonderWeb, http://wonderweb.semanticweb.org/deliverables/documents/D20-1.1.pdf, 2002.

[64] M. Klein, N.F. Noy, *A component-based framework for ontology evolution*, In Proceedings of Workshop on Ontologies and Distributed Systems at 18th International Joint Conference on Artificial Intelligence (IJCAI-03), Acapulco, Mexico, CEUR-WS Volume 71, available as Technical Report IR-504, Vrije Universiteit Amsterdam, 2003.

[65] M. Klein, A. Kiryakov, D. Ognyanov, D. Fensel, *Ontology versioning and change detection on the Web*, In Proceedings of the 13th Europian Conference on knowledge engineering and knowledge management (EKAW 2002), Siguenza, Spain, LNCS 2473, pp. 197-212, 2002.

[66] M. Klein, D. Fensel, *Ontology versioning for the Semantic Web*, In Proceedings of the 1st International Semantic Web Working Symposium (SWWS), Stanford University, California, USA, pp. 75-91, 2001.

[67] D. Landes, *Design KARL – A language for the design of knowledge-based systems*, In Proceedings of the 6th International conference on Software Engineering and Knowledge Engineering (SEKE'94), Jurmala, Lettland, pp. 78-85, 1994.

[68] G. Leganza, IT Trends 2003, Midyear Update: Enterprise Architecture, Report Giga Group, 2003.

[69] J.A. Leite, *Evolving knowledge bases: specification and semantics*, IOS Press, ISSN: 0922-6389, 2003.

[70] A. Maedche, B. Motik, L. Stojanovic, *Managing multiple and distributed ontologies on the Semantic Web*, the VLDB Journal - Special Issue on Semantic Web, Volume 12, pp. 286-302, 2003.

[71] A. Maedche, B. Motik, L. Stojanovic, R. Studer, R. Volz, *An infrastructure for searching, reusing and evolving distributed ontologies*, In Proceedings of the Twelfth International World Wide Web Conference (WWW 2003), Budapest, Hungary, ACM, pp. 439-448, 2003.

[72] A. Maedche, B. Motik, L. Stojanovic, R. Studer, R. Volz, *Ontologies for enterprise knowledge management*, IEEE Intelligent System, Volume 18, Number 2, pp. 26-34, March/April 2003.

[73] A. Maedche, *Ontology learning for the Semantic Web*, Kluwer, ISBN: 0792376560, 2002.

[74] A. Maedche, L. Stojanovic, R. Studer, R. Volz, *Managing multiple ontologies and ontology evolution in OntoLogging*, In Proceedings of the Conference on Intelligent Information Processing (IIP-2002), Montreal, Canada, pp. 51-63, 2002.

[75] A. Maedche, S. Staab, *Measuring similarity between ontologies*, In Proceedings of the 13th European Conference on Knowledge Engineering and Knowledge Management (EKAW 2002), Siguenza, Spain, LNCS 2473, pp. 251-263, 2002.

[76] A. Maedche, V. Zacharias, *Clustering ontology-based metadata in the Semantic Web*, In Proceeding of the 6th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD 2002), Helsinki, Finland, pp. 348-360, 2002.

[77] A. Maedche, S. Staab, *Ontology learning for the Semantic Web*, IEEE Intelligent Systems, Special Issue on Semantic Web, Volume 16, Number 2, pp. 72-79, March/April 2001.

[78] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, J. Widom, *Lore: A database management system for semistructured data*, In SIGMOD Record, Volume 26, Number 3, pp. 54–66, September 1997.

[79] S. McIlraith, T. Son, H. Zeng, *Semantic Web Services*, IEEE Intelligent Systems, Special Issue on the Semantic Web, Special Issue on Semantic Web, Volume 16, Number 2, pp. 46-53, March/April 2001.

[80] B. Motik, A. Maedche, R. Volz, *A conceptual modelling approach for building semantics-driven enterprise applications*, In Proceedings of the First International Conference on Ontologies, Databases and Application of Semantics (ODBASE-2002), Springer, California, USA, LNCS 2519, pp. 1082-1099, 2002.

[81] T. Menzies, J. Debenham, *Expert system maintenance*, Encyclopaedia of Computer Science and Technology, editor A. Kent and J.G. Williams, Volume 47, Number 27, pp. 35-54, 2000.

[82] T. Menzies, *Knowledge maintenance: The state of the art*, The Knowledge Engineering Review, Volume 14, Number 1, pp. 1-46, 1999.

[83] T. Menzies, *Object-oriented patterns: lessons from expert systems*, Software – Practice and Experience (SPE), Volume 27, Number 12, pp. 1457-1478, 1997.

[84] B. Meyer, *Object-oriented software construction*, 2nd edition, Prentice-Hall, Upper Saddle River, NJ, ISBN: 0136291554, 2000.

[85] D. McGuinness, F. van Harmelen, *OWL Web Ontology Language overview*, W3C Recommendation, http://www.w3.org/TR/2004/REC-owl-features-20040210, 2004.

[86] D. McGuinness, *Conceptual modeling for distributed ontology environments*, In Proceedings of the International Conference on Conceptual Structures (ICCS 2000), Darmstadt, Germany, pp. 100-112, 2000.

[87] D. McGuinness, R. Fikes, J. Rice, S. Wilder, *An environment for merging and testing large ontologies*, In Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR-2000), Breckenridge, Colorado, USA, Morgan-Kaufman, pp. 483-493, 2000.

[88] S. Narayanan, S. McIlraith, Simulation, *Verification and automated composition of web services*, In Proceedings of the Eleventh International World Wide Web Conference (WWW-2002), Hawaii, USA, ACM, pp. 77-88, 2002.

[89] S. Nelson, W. Johnston Douglas, B. Humphreys, *Relationships in Medical Subject Headings*, Relationships in the organization of knowledge, edited by C.Bean and R. Green, Kluwer Academic Publishers, ISBN 0-7923-6813-4, pp.171-184, 2001.

[90] S. Nelson, *MeSH, UMLS, and the Semantic Web*, Presentations at the Medical Information Society of Taiwan (MIST), Taoyuan, Taiwan, http://www.nlm.nih.gov/mesh/presentations/taiwan2001/semanticweb/index.htm, 2001.

[91] F. Nickols, *Change management 101: A primer*, http://home.att.net/~nickols/change.htm

[92] K. Norman, *The psychology of menu selection: designing cognitive control of the human/computer interface*, Ablex Publishing Corporation, ISBN: 089391553X, 1991.

[93] N. F. Noy, M. Klein, *Visualizing changes during ontology evolution*, In Proceedings of the International Conference on Intelligent User Interfaces (IUI 2004), Madeira, Portugal, 2004.

[94] N. F. Noy, M. Klein, *Ontology evolution: not the same as schema evolution,* to apper in Knowledge and Information Systems, Volume 6, Number 4, July 2004, available as SMI technical report SMI-2002-0926, http://smi-web.stanford.edu/ pubs/SMI_Abstracts/SMI-2002-0926.html, 2002.

[95] N. F. Noy, D. McGuinness, *Ontology development 101: a guide to creating your first ontology*, Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report SMI-2001-0880, 2001.

[96] N. F. Noy, R. W. Fergerson, M. A. Musen, *The knowledge model of Protege-2000: combining interoperability and flexibility*, In Proceedings of the 12th International Conference On Knowledge Engineering and Knowledge Management (EKAW 2000), Juan-les-Pins, France, pp. 17-32, 2000.

[97] D. Oberle, B. Berendt, A. Hotho, J. Gonzales, *Conceptual user tracking*, In Proceedings of the 1st International Atlantic Web Intelligence Conference (AWIC 2003), Madrid, Spain, LNAI 2663, pp. 155-164, 2003.

[98] D. Ognyanov, A. Kiryakov, *Tracking changes in rdf(s) repositories*, in Proceedings of 13th International Conference on Knowledge Engineering and Knowledge Management (EKAW 2002), Siguenza, Spain, LNCS 2473, pp. 373-378, 2002.

[99] OIL - http://www.ontoknowledge.org/oil/

[100] D. E. Oliver, *Change management and synchronization of local and shared versions of a controlled vocabulary*, PhD thesis, Stanford University, available as SMI Report Number: SMI-2000-0849, 2000.

[101] D. E. Oliver, Y. Shahar, M. A. Musen, E. H. Shortliffe, *Representation of change in controlled medical terminologies*, Artificial Intelligence in Medicine, Volume 15, Number 1, pp. 53–76, 1999.

[102] M. T. Oezsu, P. Valduriez, *Principles of distributed database systems*, Prentice Hall International, Inc., ISBN: 0136597076, 1999.

[103] OWL - http://www.w3.org/2001/sw/WebOnt/

[104] R. J. Peters, M. Oezsu, *An axiomatic model of dynamic schema evolution in object-base management systems*, ACM Transactions on Database Systems, Volume 22, Number 1, pp. 75-114, 1997.

[105] G. Pierre, M. van Steen, *Dynamically selecting optimal distribution strategies on web documents*, IEEE Transaction on Computers, Volume 51, Number 6, pp. 637-651, June 2002.

[106] H.S. Pinto, S. Staab, Y. Sure, C. Tempich, *OntoEdit empowering SWAP: A case study in supporting DIstributed, Loosely-controlled and evolvInG Engineering of oNTologies (DILIGENT)*, In Proceedings of the 1st European Semantic Web Symposium, Heraklion, Greece, Springer, LNCS 3053, pp. 16-30, 2004.

[107] H.S. Pinto, J. Martins, *A methodology for ontology integration*, In Proceedings of the international conference on Knowledge capture (K-CAP 2001), Victoria, British Columbia, Canada, pp. 131-138, 2001.

[108] A. Pons, R. K. Keller, *Schema evolution in object databases by catalogs*, In Proceedings of International Database Engineering and Applications Symposium (IDEAS'97), Montreal, Canada, pp. 368-376, 1997.

[109] A. Pons, R. Keller, *Evolving object database schema by a catalog of primitive modifications*, In Proceedings of the Eight International Conference on Software Engineering and its Applications, Paris, France, pp. 439-452, 1995.

[110] D. Poole, A. Mackworth, R. Gobel, *Computational intelligence: A logical approach*, Oxford University Press, New York, ISBN 0-19-510270-3, 1998.

[111] F. Puppe, *Systematic introduction to expert systems - knowledge representations and problem solving methods*, Springer, ISBN 0387562559, 1993.

[112] V. Ramana, *The importance of hierarchy building in managing unstructured data*, Special Supplement to KM World, March 2002.

[113] J.F. Roddick, A *survey of schema versioning issues for database systems*, Information and Software Technology, Volume 37, Number 7, pp. 383-393, 1996.

[114] E. Rundensteiner, A. Leem, Y. Ra, *Capacity-augmenting schema changes on object-oriented databases: towards increased interoperability*, In Proceedings of International Conference on Object-Oriented Information Systems (OOIS'98), Paris, France, pp. 349-368, 1998.

[115] G. Salton, C. Buckley, *Improving retrieval performance by relevance feedback*, Journal of the American Society for Information Science, Volume 41, Number 4, pp. 288-297, 1990.

[116] W. Scacchi, A. Valente, *Developing a knowledge web for business process redesign*, In Proceedings of the Twelfth Workshop on Knowledge Acquisition, Modeling and Management (KAW'99), Voyager Inn, Banff, Alberta, Canada, 1999.

[117] D. Sjoberg, *Quantifying schema evolution,* Information and Software Technology Journal, Volume 35, Number 1, pp. 35–54, 1993.

[118] J. Srivastava, R. Cooley, M. Deshpande, P.N. Tan, *Web usage mining: discovery and application of usage patterns from web data*, in SIGKDD Explorations, Volume 1, Number 2, pp.12-23, 2000.

[119] S. Staab, H.-P. Schnurr, R. Studer, Y. Sure, *Knowledge processes and ontologies*, IEEE Intelligent Systems, Special Issue on Knowledge Management, Volume 16, Number 1, January/February 2001.

[120] J.E. Stiglitz, P.R. Orszag, J.M. Orszag, *The role of government in a digital age*, http://www.ccianet.org/digital_age/report.pdf, 2000.

[121] L. Stojanovic, *An approach for continual ontology improvement*, to appear in Proceedings of the First International Conference on Knowledge Engineering and Decision Support (ICKEDS'2004), Porto, Portugal, 2004.

[122] L. Stojanovic, J. Schneider, A. Maedche, S. Libischer, R. Studer, T. Lumpp, A. Abecker, G. Breiter, J. Dinger *The role of ontologies in autonomic computing systems*, To appear in IBM Systems Journal, Volume 43, Number 3, 2004.

[123] L. Stojanovic, G. Kavadias, D. Apostolou, F. Probst, K. Hinkelmann, *E-Gov Lifecycle Ontology*, available as Deliverable D2, EU/IST Project OntoGov, http://www.ontogov.org, 2004.

[124] L. Stojanovic, N. Stojanovic, J. Gonzalez, R. Studer, *OntoManager - a system for the usage-based ontology managemen*t, In Proceedings of the 2st International Conference on Ontologies, Databases and Application of Semantics (ODBASE 2003), Catania, Sicily, Italy, LNCS 2888, pp. 858-875, 2003.

[125] L. Stojanovic, A. Maedche, N. Stojanovic, R. Studer, *Ontology evolution as reconfiguration-design problem solving*, In Proceedings of the international conference on Knowledge capture (K-CAP'03), Sanibel Island, FL, USA, pp. 162-171, 2003.

[126] L. Stojanovic, N. Stojanovic, A. Maedche, *Change discovery in ontology-based knowledge management systems*, In Proceedings of 21st International Conference on Conceptual Modelling (ER'2002), Workshop on Evolution and Change in Data Management (ECDM'02), Tampere, Finland, 2002, Revised Papers, LNCS 2784, ISBN 3-540-20255-2, pp. 51-62, 2003.

[127] L. Stojanovic, N. Stojanovic, S. Handschuh, *Evolution of the metadata in the ontology-based knowledge management systems*, In Proceedings of the 1st German Workshop on Experience Management, Berlin, Germany, LNI 10 GI 2002, pp. 65-77, 2002.

[128] L. Stojanovic, B. Motik, *Ontology evolution within ontology editors*, In Proceedings of the OntoWeb-SIG3 Workshop Evaluation of Ontology-based Tools (EON2002) at the 13th International Conference on Knowledge Engineering and Knowledge Management (EKAW 2002), Siguenza, Spain, CEUR-WS Volume 62, pp. 53-62, 2002.

[129] L. Stojanovic, A. Maedche, B. Motik, N. Stojanovic, *User-driven ontology evolution management*, In Proceedings of the 13th European Conference on Knowledge Engineering and Knowledge Management (EKAW'02), Siguenza, Spain, LNCS 2473, pp. 285-300, 2002.

[130] N. Stojanovic, L. Stojanovic, *Evolution in the ontology-based knowledge management system*, In Proceedings of the Xth European Conference on Information Systems - ECIS 2002, Gdañsk, Poland, 2002.

[131] N. Stojanovic, L. Stojanovic, *Usage-oriented evolution of ontology-based knowledge management systems*, In Proceedings of the 1st International Conference on Ontologies, Databases and Application of Semantics (ODBASE 2002), Irvine, CA, pp. 1186-1204, 2002.

[132] N. Stojanovic, L. Stojanovic, J. Gonzalez, *On enhancing searching for information in an information portal by tracking users' activities*, In Proceedings of the First International Workshop on Mining for Enhanced Web Search (MEWS 2002), held in conjunction with 3rd International Conference on Web Information Systems Engineering (WISE 2002), Singapore, IEEE Computer Society 2002, pp. 246-256, 2002.

[133] N. Stojanovic, A. Maedche, S. Staab, R. Studer, Y. Sure, *SEAL - a framework for developing SEmantic portALs*, In Proceedings of the international Conference on Knowledge Capture (K-CAP'01), Victoria, British Columbia, Canada , pp. 155-162, 2001.

[134] H. Stuckenschmidt, M. Klein, *Integrity and change in modular ontologies*, In Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'03), Acapulco, Mexico, pp. 900-908, 2003.

[135] M. Stumptner, F. Wotawa, *Model-based reconfiguration*, In Proceedings of 5th International Conference on Artificial Intelligence in Design (AID 98), Lisbon, Portugal, 1998.

[136] H. Su, D. Kramer, E. Rundensteiner, *XEM: XML Evolutin Manager*, Computer Science Technical Report Series, WORCESTER POLYTECHNIC INSTITUTE, WPI-CS-TR-02-09, 2002.

[137] Y. Sure, *On-To-Knowledge - ontology based knowledge management tools and their application*, In German Journal Kuenstliche Intelligenz, Special Issue on Knowledge Management (1/02), pp. 35-37, 2002.

[138] E. Sunagawa, K. Kozaki, Y. Kitamura, R. Mizoguchi, *An environment for distributed ontology development based on dependency management*, In Proceedings of the Second International Semantic Web Conference (ISWC2003), Sanibel Island, FL, USA, pp.453-468, 2003.

[139] E. Sunagawa, K. Kozaki, Y. Kitamura, R. Mizoguchi, *Management of dependency between two or more ontologies in an environment for distributed development*, In Proceedings of the International Workshop on Semantic Web Foundations and Application Technologies (SWAFT), Nara, Japan, http://www-kasm.nii.ac.jp/ SWFAT/PAPERS/SWFAT17R.PDF, 2003.

[140] M. Tallis, Y. Gil, *Designing scripts to guide users in modifying knowledge-based systems*, In Proceedings of the 14th National Conference on Artificial Intelligence (AAAI/IAAI 1999), Orlando, Florida, USA, pp. 242-249, 1999.

[141] E. Tambouris, *An integrated platform for realising online one-stop government: the eGOV project*, In Proceedings of the DEXA International Workshop "On th Way to Electronic Government", IEEE Computer Society Press, Los Alamitos, CA, ISBN 0-7695-1230-5, pp. 359-363, 2001.

[142] V.A.M. Tamma, T.J.M Bench-Capon, *A conceptual model to facilitate knowledge sharing in multi-agent systems*, In Proceedings of the Autonomous Agents Workshop on Ontologies in Agent Systems (OAS 2001), Montreal, Canada, pp. 69-76, 2001.

[143] A. ten Teije, F. van Harmelen, A. Th. Schreiber, B. J. Wielinga, *Construction of problem-solving methods as parametric design*, International Journal of Human-Computer Studies, Special issue on problem-solving methods, Vol. 49, Number 4, pp. 363-389, 1998.

[144] M.S. Tuttle, D. Sherertz D, M. Erlbaum, *Adding your terms and relationships to the UMLS Metathesaurus*, In Proceedings of the Fifteenth Annual Symposium on Computer Applications in Medical Care, New York, USA, pp. 219-223, 1991.

[145] L. Tokuda, D. Batory, *Automating three modes of evolution for object-oriented software architecture*, In Proceedings of the 5th Conference on Object Oriented Technologies and Systems (COOTS'99), San Diego, CA, 1999.

[146] M. Uschold, M. Gruninger, *Ontologies: principles, methods, and applications*, Knowledge Engineering Review, Volume 11, Number 2, pp. 93-155, 1996.

[147] M. Ushold, M. Healy, K. Williamson, P. Clark, S. Woods, *Ontology reuse and application*, In Proceedings of the International Conference on Formal Ontology and Information Systems (FOIS'98), IOS Press, pp. 179-192, 1998.

[148] J. Valdman, *Log file analysis*, available as Technical Report DCSE/TR-2001-04, Department of Computer Science and Engineering (FAV UWB), http://www.kiv.zcu.cz/publications/2001/tr-2001-04.pdf, 2001.

[149] K. Verma, R. Akkiraju, R. Goodwin, P. Doshi, J. Lee, *On accommodating inter service dependencies in web process flow composition*, First International Semantic Web Services Symposium, 2004 AAAI Spring Symposium Series, ISBN 1-57735-198-3, pp. 37-43, 2004.

[150] R. Volz, *Web Ontology Reasoning With Logic Databases*, PhD Thesis, University at Fridericiana zu Karlsruhe (TH), Germany, 2004.

[151] J.R. Wen, J.Y. Nie, H.J. Zhang, *Clustering user queries of a search engine*, In Proceedings of the 10th International World Wide Web Conference (WWW10), Hong Kong, pp. 162-168, 2001.

[152] B. Wielinga, G. Schreiber, *Configuration design problem solving*, IEEE Intelligent Systems, Volume 12, Number 1, pp. 49-56, March-April 1997.

[153] I. Witten, E. Frank, *Data mining: practical machine learning tools and techniques with java implementations*, Morgan Kaufmann, ISBN: 1558605525, 1999.

[154] R. Zicari, *A framework for schema updates in an object-oriented database system*, In Proceedings of the Seventh International Conference on Data Engineering (ICDE'91), Kobe, Japan, pp. 2-13, 1991.