Vrije Universiteit Brussel

FACULTEIT VAN DE WETENSCHAPPEN
Departement Computer Wetenschappen
Web & Information Systems Engineering

An Approach to Web-based Ontology Evolution

Proefschrift ingediend met het oog op het behalen van de graad van Doctor in de wetenschappen

Peter Plessers

Academiejaar: 2005 – 2006

Promotor: Prof. Dr. Olga De Troyer

"It is an error to imagine that evolution signifies a constant tendency to increased perfection. That process undoubtedly involves a constant remodelling of the organism in adaptation to new conditions; but it depends on the nature of those conditions whether the directions of the modifications effected shall be upward or downward."

— Thomas H. Huxley (1825 - 1895)

Abstract

The World Wide Web has become one of the biggest success stories in recent history. It has allowed global exchange of information on a scale unprecedented in human history, and has an impact on almost all facets of our daily lives. Notwithstanding its enormous success, one of the shortcomings of the Web is that most information is represented in a form only usable for human interpretation i.e., the current Web can be considered to be *machine-readable*, but unfortunately not *machine-understandable*. The *Semantic Web* has been proposed as a solution to overcome this shortcoming by making the semantics of Web content explicit by means of *ontologies*.

As is the case with everything in the world that surrounds us, ontologies are not indifferent to changes. Ontologies evolve as a consequence of changes of domains they describe as well as changes in business and user requirements. Because ontologies are intended to be used and extended by other ontologies, and because they are deployed in a highly decentralized environment as the Web, the problem of ontology evolution is a far from trivial problem. The fact that ontologies depend on other ontologies means that the consequences of changes don't remain local to the ontology itself, but affect depending ontologies as well. Furthermore, the decentralized nature of the Web makes it impossible to simply propagate changes to depending artifacts.

In this dissertation, we propose an ontology evolution approach that (1) allows ontology engineers to request changes for the ontologies they manage; (2) ensures that the ontology evolves from one consistent state into another consistent state; (3) guarantees that the depending artifacts of an ontology remain consistent after changes have been applied; (4) provides a detailed overview of the changes that occur. The cornerstones of our approach are the notion of a *version log* and the *Change Definition Language*. A version log stores for each concept ever defined in an ontology the different versions it passes through during its life cycle. The Change Definition Language is a temporal logic based language that allows ontology engineers to formally define changes. The purpose of the change definitions expressed in this Change Definition Language are twofold: they are used for both requesting and implementing changes to an ontology, as well as detecting occurrences of change definitions in the evolution of an ontology.

Samenvatting

Het World Wide Web is één van de grootste successverhalen in de recente geschiedenis. Het heeft de uitwisseling van informatie mogelijk gemaakt op een schaal die ongezien was in de gehele menselijke geschiedenis, en heeft een impact op bijna alle facetten van ons dagelijks leven. Niettegenstaande dit enorm succes is één van de tekortkomingen van het Web dat de meeste informatie aangeboden wordt in een vorm die enkel voor mensen begrijpbaar is. Het huidige Web kan beschouwd worden als zijnde *machine-leesbaar*, maar helaas niet *machine-begrijpbaar*. Het Semantisch Web werd voorgesteld als een oplossing voor deze tekortkoming door de betekenis van informatie expliciet te maken door gebruik te maken van ontologieën.

Zoals alles in de wereld rondom ons, zijn ontologieën ook niet onverschillig aan veranderingen. Ontologieën evolueren niet enkel als gevolg van veranderingen in domeinen die zij beschrijven, maar ook door wijzigingen in business en user requirements. Doordat ontologieën bedoelt zijn om gebruikt en uitgebreid te worden door andere ontologieën, en omdat zij ontplooid worden in een erg gedecentraliseerde omgeving zoals het Web, is het probleem van ontologie evolutie verre van een triviaal probleem. Het feit dat ontologieën afhankelijk zijn van andere ontologieën betekent dat de gevolgen van veranderingen niet beperkt blijven tot de ontologie zelf, maar eveneens mogelijke gevolgen kunnen hebben voor afhankelijke artefacten. Bovendien zorgt het gedecentraliseerde karakter van het Web ervoor dat het onmogelijk wordt om veranderingen simpelweg te propageren naar afhankelijke artefacten.

In deze thesis stellen we een ontologie evolutie benadering voor die (1) ontologie ingenieurs toelaat om veranderingen aan te vragen voor de ontologieën die zij beheren; (2) er voor zorgt dat een ontologie steeds evolueert van een consistente versie naar een andere consistente versie; (3) er voor zorgt dat de afhankelijke artifacten van een ontologie eveneens consistent blijven nadat veranderen zijn toegebracht; (4) een gedetaileerd overzicht geeft van alle aangebrachte veranderingen. De hoekstenen van onze benadering zijn de notie van een *version log* en de *Change Definition Language*. Een version log bewaart voor ieder concept dat ooit gedefinieerd is in een ontologie de verschillende versies die dat concept doormaakt gedurende zijn hele levenscyclus. De Change Definition Language is een taal gebaseerd op een temporele

logica die het ontologie ingenieurs mogelijk maakt om veranderingen formeel te definiëren. Het doel van de definities van veranderingen uitgedrukt in deze Change Definition Language is tweeledig: zij worden gebruikt voor zowel het aanvragen en implementeren van veranderingen in een ontologie, als het detecteren van gebeurtenissen van definities van veranderingen in de evolutie van een ontologie.

Acknowledgement

My greatest gratitude goes to my promoter, Prof. Dr. Olga De Troyer. The start of my PhD, now four years ago, was not what one would call a plain sailing, but she always kept believing in me and has put in a lot of effort to keep me going. She deserves all the thanks for giving me the opportunity to join the WISE research group and for guiding and supporting me during these past years. Furthermore, she was always willing to give me her much appreciated advice whenever I needed it.

I would also like to thank the members of my jury, Jeen Broekstra, York Sure, Geert-Jan Houben, Robert Meersman and Dirk Vermeir for providing me with interesting comments and pointers to further improve the quality of my work. I especially would like to thank Dirk Vermeir for taking the time to proofread an early version of a chapter of my dissertation.

I would like to address a word of thank you to everyone involved in the Advanced Media project. The project gave me the opportunity to explore various research problems and denotes the beginning of my research on ontology evolution. The work done during this project has greatly contributed to the end result of my dissertation. In this context, I especially would like to thank Johan Brichau, Thomas Cleenewerck and Dirk Deridder for making my time in building X, the farthest corner of the VUB campus, more enjoyable.

A BIG thank you goes to my colleague and friend Sven Casteleyn with whom I have been working together in the past four years. His proof readings have certainly improved the quality of my work significantly. Furthermore, the many discussions I had with him have contributed a lot to my work, but even more importantly also to me, as a person.

I would also like to thank my thesis student Johan Van den Broeck for his excellent help on the implementation on some of the tools needed to validate my work and for playing the role of guinea pig for my Change Definition Language.

Furthermore, I would also like to thank my colleagues at the WISE research group for all the valuable discussions and the (maybe less productive, but certainly enjoyable) chit-chat. Thanks go to Wesley Bille, Frederic Kleinermann, Abed Mushtaha, Bram Pellens and Wael Al Sarraj.

A lot of gratitude goes to my parents, Henri Plessers and Madeleine

Aerts. They are the ones who have given me the opportunity to study and who have always supported me in every decision I made. They taught me to never give in, which turned out to be a valuable lesson during the last couple of months :-).

Last but not least, thanks to my girlfriend Veerle Van Aken for her enormous support and for enduring me for the last couple of months when I was primarily concerned with writing my dissertation. She has always been there for me, especially in times when I was at a complete loss.

Finally, I would like to thank all the people I unwittingly forgot to mention here. Many people contributed in some way to this PhD and I thank all of them.

Glossary

Base Change Definition A base change definition is a definition of a change expressed in terms of the Change Definition Language.

Change Definition Set A change definition set is a collection of conceptual change definitions an ontology engineer or a maintainer of a depending artifact is interested in.

Compatibility Requirement A compatibility requirement is a requirement that a version of an ontology should fulfill to be considered backward compatible with its previous version for a certain depending artifact.

Conceptual Change Definition A conceptual change definition defines a change on a conceptual level by grouping a collection of base change definitions.

Deduced Change A deduced change is a change that complements a requested change of the ontology engineer in order to resolve a detected inconsistency.

Detected Change A detected change is an occurrence of a conceptual change definition that was not explicitly requested by an ontology engineer nor deduced by the ontology evolution framework in order to resolve inconsistencies, but has that been detected by the ontology evolution framework.

Evolution Log An evolution log is a particular interpretation of the evolution of an ontology in terms of occurrences of change definitions. Different interpretations of the same ontology evolution may exist by means of different evolution logs based on different change definitions.

Requested Change A requested change is a change, which is expressed in terms of a conceptual change definition, that is explicitly requested by an ontology engineer to be applied to an ontology.

Version Log A version log is a log that stores for each concept ever defined in an ontology (this includes Classes, Properties and Individuals), the

different versions it passes through during its life cycle: from its creation, over its modifications, until its eventual retirement.

Virtual Version Log A virtual version log is a version log that describes a virtual evolution of an ontology i.e., a possible evolution that however has not occurred (yet) in reality.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Most people likely associate the term evolution immediately with the domain of biology and the evolution theory of natural selection that was popularized by Charles Darwin in his book *The Origin of Species* [14]. However, the subject of evolution has also played an important role in a variety of other domains. Evolution has been – and still is – the topic of extensive research in domains such as economics, psychology, natural languages and computer science. While the exact meaning of the term evolution varies from domain to domain, it can in general be best circumscribed as a process in which something changes into an improved form that better suits its environment's needs.

In the domain of computer science, the topic of evolution has been investigated in different settings such as software evolution, knowledge base maintenance and database evolution. The introduction of the World Wide Web, and particularly the Semantic Web [8] where ontologies are used to make the semantics of information explicit, has brought the topic of evolution into a new perspective. The topic of this dissertation therefore concerns the problem of evolving ontologies, a problem that affects the cornerstones of the Semantic Web.

1.1 Research Context

The World Wide Web (also referred to as WWW or simply the Web) began as a networked information project at CERN, initiated by Tim-Berners Lee and Robert Cailliau. Although it was originally intended to be used to exchange information between members of the physics community, it quickly spread to other research disciplines and rapidly evolved into the success story we know today. Although the Web started off with merely a handful of users and Web pages, its usage has swiftly reached more than one billion

users world wide[1], and the number of Web sites has surpassed 80 million[2]. Furthermore, the number of Web sites is still growing at a dazzling speed, illustrated by the fact that the Web has doubled in size in the past three years.

Notwithstanding the enormous success of the Web, its current form has a number of severe shortcomings. One of these shortcomings is that most information on the Web is represented in a form (mostly HTML) solely usable for human interpretation — i.e., the current Web can be considered to be *machine-readable*, but unfortunately not *machine-understandable*. This shortcoming is for example clearly illustrated by all major search engines that are mainly restricted to syntax-based queries, as they are not able to grasp the semantics of the information found on the Web. The *Semantic Web* has been proposed as a solution to overcome this shortcoming by making the semantics of Web content explicit.

The Semantic Web is defined as an additional layer on top of the current Web. Tim Berners-Lee described the Semantic Web as *an extension of the current Web in which information is given well-defined meaning, better enabling computers and people to work in cooperation* [8]. To give information a well-defined meaning, the Semantic Web relies on the use of ontologies. The term ontology, originating from the domain of philosophy, is in computer science defined as a formal, explicit specification of a shared conceptualization of a domain of interest [30]. In other words, ontologies form a formal representation of concepts and relations between these concepts that can be distinguished in a particular domain (e.g., medicine). Its explicit and formal specification allows machines to reason about the information captured by the ontologies. Therefore, the additional layer of ontologies on top of the current Web transforms the Web into a machine-understandable Web.

As follows from the definition of an ontology, an ontology is a *shared* conceptualization i.e., an ontology is intended to be used and extended by other ontologies. Furthermore, several artifacts (e.g., Web sites, applications, ...) may depend on the same ontology. As a consequence, the Semantic Web extends the Web with what can be considered to be a true *web of ontologies*, where ontologies are interlinked and used by Web sites and Web applications. The distributed and decentralized characteristics of the Web also hold for the Semantic Web. This means that ontologies can be managed independently from each other and ontologies can be used and extended without the permission or knowledge of the owner. Moreover, the owner of an ontology may even be unaware of who uses or extends his ontology.

As is the case with everything in the world that surrounds us, ontologies are not indifferent to changes. Several reasons for ontology changes have been identified in literature by various authors [32] [51] [73] [85]. We list the

[1] See `http://www.internetworldstats.com` (figures of March 31, 2006)
[2] See `http://www.netcraft.com` (figures of April 2006)

most important reasons below:

- When a particular domain changes, the ontologies that describe this domain have to change accordingly, in order to reflect the changed real world situation.
- While the domain itself may not change, the view on the domain described by an ontology may evolve as consequence of a shift of focus i.e., particular aspects of the domain may gain or lose importance over time.
- The initial versions of an ontology may still contain design flaws, and new design flaws may possibly get introduced as a consequence of further changes to an ontology. A possible cause of ontology evolution is therefore the correction of design flaws.
- Finally, the requirements of the users of an ontology may change (e.g., a request for new or modified functionality), demanding a change to the ontology to support the changed requirements.

Because ontologies are intended to be used and extended by other ontologies, and because they are deployed on the Web, which is a highly decentralized environment, the problem of ontology evolution is far from trivial. The fact that ontologies depend on other ontologies means that the consequences of changes don't remain local to the ontology itself, but affect depending ontologies as well. Furthermore, the decentralized nature of the Web makes it impossible to simply propagate changes to depending artifacts. A manual and ad-hoc handling of an ontology evolution process is not feasible nor desirable as it is a too laborious, time intensive and complex process [86]. A structured approach is therefore essential to support the ontology engineer in this evolution process. In this dissertation, we propose such an approach.

1.2 Problem Statement

At this moment, no generally accepted single definition of ontology evolution exists in the research community. Because the term 'ontology evolution' is often used with (slightly) different meanings, we first formulate a definition of this term that will be used throughout this dissertation. This definition reflects a similar viewpoint as stated by [23] and [83]. We define ontology evolution as *the process of adaptation of an ontology to the arisen changes in a domain while maintaining both the consistency of the ontology itself as well as the consistency of depending artifacts*, where depending artifacts are other ontologies, Web sites, Web applications, etc. depending on an ontology.

The work presented in this dissertation does not claim to be a complete solution to all the various aspects that are associated with the problem of

ontology evolution. Nevertheless, this work makes a number of contributions to the research field. In this dissertation, we propose an answer to the following problems:

Problem 1: Comprehending ontology evolution An essential prerequisite to cope with ontology evolution is that maintainers of depending artifacts need to be able to obtain a clear understanding of the changes that have occurred within an ontology. This is an important requirement because these maintainers are ultimately responsible for keeping their depending artifacts consistent with a changing ontology. To support this requirement, an ontology evolution approach must be able to give a complete overview of the changes that have occurred. As ontologies can become exceedingly complex and large-scale, it is important that this overview of changes supports (1) different levels of abstraction (i.e., fine-grained changes that represent a lot of detail vs. coarse-grained changes that rather reveal a pattern of evolution), (2) can be approached from different points of view (i.e., a view that focuses on the actual changes vs. a view that rather focuses on the implications of changes), and (3) even allows for different interpretations of the same changes.

While the requirement as described above is valid for all systems considering evolution, this requirement becomes even more crucial when we consider ontologies that are deployed on the Web. In a setting as the Web, problems arise when maintainers of depending artifacts have to rely on the overview of changes placed at their disposal by the ontology engineer of an ontology they depend on. Consequently, the quality of the overview of changes depends largely on the goodwill of the ontology engineer. Because ontology engineers – due to the decentralized nature of the Web – are often unaware of the existence of depending artifacts, and the overview of changes is in the first place of interest for maintainers of depending artifacts, a high quality overview can not be taken for granted. This is especially true when we take into account that in most existing approaches the creation of such an overview is mostly a manual task. Even when the overview of changes is of high quality, this does not guarantee that it suits the needs of the depending artifacts as it eventually is just one particular view on the changes. Again – due to the decentralized nature of the Web – the overview of changes cannot be adapted to the specific requirements of depending artifacts as these are unknown.

Finally, just as a particular domain described by several ontology engineers results in ontologies that most likely are different from each other (because different ontology engineers may have different views on the same domain), an overview of the evolution of an ontology may

also lead to different interpretations of that same evolution. Different interpretations of the same changes may trouble the understanding of an ontology evolution for maintainers of depending artifacts. This is certainly the case when a depending artifact depends on more than one ontology which all use different interpretations for the same changes. As a consequence, the understanding of the changes becomes troublesome as the meaning of the changes differs from ontology to ontology. Note that it is impossible to create a standardized set of change definitions to resolve this problem for the following two reasons: (1) there exists an infinite number of possible changes and (2) differences in interpretation happen to be an intrinsic aspect of the Web.

Problem 2: Ontology consistency As ontologies are often used to reason about and to infer implicit knowledge from, it is essential for an approach supporting ontology evolution to ensure that ontologies evolve from one consistent state into another consistent state. The problem is that the changes which an ontology engineer wants to apply to an ontology may introduce inconsistencies. To maintain ontology consistency, it doesn't only suffice to verify whether an ontology remains consistent after changes are applied. In addition, when an inconsistency is detected, possible solutions should be proposed to the ontology engineer to resolve the detected inconsistency. However, the problem of inconsistency resolving is far from trivial, as it implies an approach that understands *why* an ontology is inconsistent.

Problem 3: Decentralized authority The decentralized architecture of the Web certainly has contributed a lot to its enormous success. Due to the decentralization of authority, it has become possible to create such a large and complex structure as the Web. Regarding ontologies, this means that ontologies on the Web can be used and extended without the permission of the owner. Moreover, an ontology engineer doesn't even have to be aware of possible depending artifacts that depend on his ontology. Although the decentralized architecture of the Web clearly has a number of advantages, it also imposes a number of problems w.r.t. ontology evolution:

- an ontology engineer cannot force maintainers of depending artifacts to update to the latest version of an ontology because he doesn't have the necessary permissions and depending artifacts may be unknown;
- maintainers of depending artifacts are in control of their own depending artifacts and therefore may update at their own pace (which includes the decision not to update at all);
- the other way round, maintainers of depending artifacts cannot

prevent ontologies they depend on from changing because they don't have the necessary permissions to control the ontologies.

In the field of distributed databases, the problem of consistency maintenance between databases was solved by propagating the changes applied to a database to all depending databases. Because ontology engineers cannot force maintainers of depending artifacts to update, a similar technique of change propagation cannot be used to keep depending artifacts consistent with a changing ontology. Furthermore, because maintainers of depending artifacts cannot prevent ontologies they depend on from changing and because maintainers of depending artifacts may decide to update to the latest version of an ontology at their own pace, it is likely that multiple versions of an ontology need to be supported. Finally, the refusal of a maintainer of a depending artifact to update may prevent other depending artifacts from updating too. Consider as an example an ontology O_3 that imports an ontology O_2 and O_2 imports an ontology O_1. When O_1 changes but O_2 doesn't update to the latest version of O_1, ontology O_3 is blocked as it cannot be updated to the latest version of O_1.

Problem 4: Depending artifact consistency As mentioned in problem 3, maintainers of depending artifacts are eventually responsible to decide whether or not to update to the latest version of an ontology. In order for these maintainers to make a well-considered decision, it is important that they are aware of the consequences of updating for their depending artifacts. When considering ontologies as depending artifacts, the following questions do arise. Which inconsistencies does an update introduce? Which are the concept definitions that need to be changed to resolve an inconsistency? Is essential information missing since the previous used version?

1.3 Goal

In this dissertation, we aim to develop an ontology evolution approach that:

- allows ontology engineers to request and apply changes for the ontologies they manage;
- ensures, when an ontology engineer decides to change an ontology, that the ontology evolves from one consistent state into another consistent state;
- guarantees that the depending artifacts of an ontology remain consistent after changes have been applied;

- provides a detailed overview of the changes that occur, supporting different levels of abstraction, different view points and different interpretations.

As we deal in this dissertation with the evolution of ontologies on the Web, we focus our approach on the OWL Web ontology language which has been a recommendation of the W3C as the standard ontology language for the Web since 2004. In particular, we focus on the DL variant of OWL due to its characteristics of computational completeness and decidability.

1.4 Approach

In the ontology evolution framework presented in this dissertation, we represent the evolution of an ontology by means of a *version log*. A version log stores for each concept ever defined in the ontology (this includes Classes, Properties and Individuals), the different versions it passes through during its life cycle: from its creation, over its modifications, until its eventual retirement. Note that the version log doesn't represent an interpretation of the evolution in terms of changes (e.g., addClass, changeSubClassOf, ...), but rather lists the state of each concept at the different moments in time.

When an ontology engineer intends to modify an ontology, he specifies a *change request* in terms of *changes* that he wants to apply. Changes are in our approach formally defined in terms of a temporal logic based language, called the *Change Definition Language*. This Change Definition Language allows us to define changes in terms of conditions that must hold before and after the appliance of the change (respectively, pre- and post-conditions). For this language, we have adopted a *hybrid-logic approach* that is midway between modal-logic and predicate-logic approaches. The advantage is that, besides expressing temporal relations, it is also possible to refer to explicit moments in time.

As requested changes of an ontology engineer may turn the ontology into an inconsistent state, the ontology evolution framework checks for ontology consistency after each requested change and suggests possible solutions to resolve inconsistencies, if needed. As we focus on OWL DL, we can use existing OWL reasoners (e.g., Fact [37], Racer [62], Pellet [81], ...) to detect inconsistencies. However, a major drawback of current reasoners is that they provide little or no information to resolve inconsistencies. We therefore extended the tableau reasoning algorithm, used by most state-of-the-art reasoners, in order to determine which concept definitions are causing an inconsistency.

Once inconsistencies have been found, they are resolved by adding additional changes, called *deduced changes*, to the change request. In our ontology evolution framework, we provide the ontology engineers with a set of rules they can rely on to add deduced changes to a change request. The

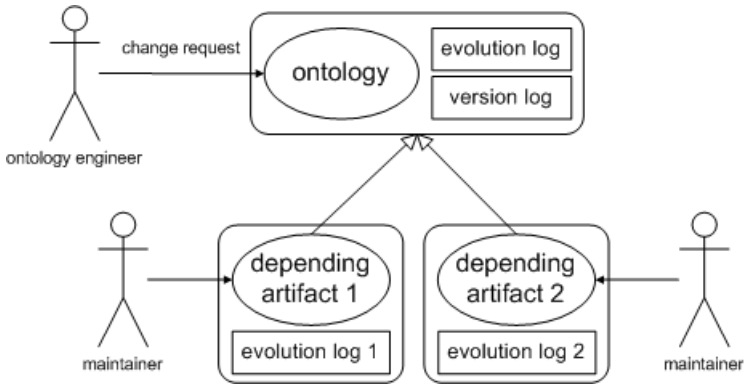

Figure 1.1: The approach allows different interpretations for the same ontology evolution

set of rules ensures that correct deduced changes are added in order to resolve a detected inconsistency. This means that, after all inconsistencies are resolved, the change request contains the necessary changes to transform an ontology from one consistent state into another consistent state.

Furthermore, changes to an ontology may also cause inconsistencies in the depending artifacts that depend on this changed ontology. To prevent depending artifacts from becoming inconsistent, we redefine dependencies between depending artifacts and ontologies to dependencies between depending artifacts and ontologies *at a given moment in time* i.e., a depending artifact depends on a specific version of an ontology. When an ontology changes, its depending artifacts remain consistent as they still dependent on the old version, and its maintainers can update at their own pace, or decide not to update to the new version.

To better understand the evolution of an ontology, our approach allows to create an *evolution log* for an ontology. Such an evolution log is, in contrast with a version log, a particular interpretation of the evolution of an ontology in terms of change definitions. Remember that change definitions are defined in terms of a temporal logic based language, which makes it possible to automatically detect changes that have occurred by evaluating the change definitions against a version log. In fact, the change definitions can be seen as temporal queries. We use the term *detected changes* to refer to the occurrence of change definitions that have been detected by the framework. An evolution log eventually may contain occurrences of requested changes, deduced changes and detected changes. The clear distinction between the *representation* of an evolution (using a version log) and the *interpretation* of an evolution (using an evolution log), makes it possible to look at an ontology evolution at different levels of abstraction, from different view points, and to associate different interpretations with the same evolution. Figure 1.1 illustrates the association of different interpretations to the same ontology

evolution. Both maintainers of the depending artifacts can have an own interpretation of the evolution of the ontology they depend on that differs from the interpretation of the ontology engineer himself. The differences in interpretation are represented by the different evolution logs.

1.5 Advantages

The approach outlined in the previous section provides several advantages compared to the current situation:

- The approach of automatic change detection makes it possible to generate more semantically rich evolution logs without additional effort from the ontology engineer then is generally achieved with manual approaches.

- Due to the change detection mechanism, the quality of evolution logs is in our approach far less dependent on the tools that are used to edit an ontology. When the evolution log is only populated with requested changes by the ontology engineer, the resulting evolution log depends a great deal on the change operations supported by the tool. Some tools may offer the ontology engineer a comprehensive set of complex change operations, while with other tools this set is rather limited. The result would be that the tool supporting a rich set of complex change operations would create a rich evolution log, while the evolution log of the tool supporting a limited set of complex change operations would be rather poor. Our change detection mechanism is able to smoothen out differences in tool support as it enables us to enrich an evolution log with detected changes.

- The clear separation between the representation of an evolution (by means of a version log) and the interpretation of this evolution in terms of changes (by means of an evolution log), allows maintainers of depending artifacts to have different interpretations on the same evolution.

- Maintainers of depending artifacts can update to a new version of an ontology they depend on at their own pace, without preventing other depending artifacts from updating.

1.6 Contributions

With the new ontology evolution approach presented in this dissertation, we have made the following contributions to the research community:

- We present a formal language based on a temporal hybrid logic approach, called the *Change Definition Language*, that allows ontology engineers to define both primitive and complex changes in a declarative way. The Change Definition Language allows to unambiguously define the semantics of changes.

- Making use of the formal definitions of changes, the proposed approach of change detection makes it possible to automatically detect the occurrence of changes. This approach of change detection gives rise to a semantically richer evolution log and allows to have different views on and interpretations of the same evolution of an ontology.

- While existing OWL DL reasoners are able to identify ontology inconsistencies, they don't reveal the cause of the detected inconsistency. Therefore, we propose an extension to current OWL DL reasoners in order to determine the exact cause of a detected inconsistency. The proposed extension selects those axioms from the ontology that together form the cause of an inconsistency. Removing one of these axioms causing the inconsistency guarantees to resolve the detected inconsistency.

- As removing a complete axiom as a solution to resolve an inconsistency seems in most cases as throwing away the baby with the bathwater, we present a set of rules to weaken axioms instead. This set of rules can be applied by an ontology engineer to the selection of axioms causing the inconsistency gathered from the extended reasoner in order to resolve contradictions.

- We propose an approach to verify whether a new version of an ontology remains backward compatible with an older version for a particular depending artifact. The approach is based on a set of compatibility requirements that a maintainer of a depending artifact can specify, and that are automatically checked by the proposed framework.

- We also present an approach to keep multiple, distributed ontologies consistent after changes are applied to one of the ontologies. The approach redefines dependencies between ontologies as dependencies between version logs at a specific moment in time. Changes to an ontology won't affect the consistency of depending ontologies as they depend on a specific version of the ontology. Past versions of an ontology can be reconstructed by making use of a version log.

- Because the refusal of maintainers of depending artifacts to update may prevent other depending artifacts from updating as well, an approach is proposed based on the use of a virtual version log to circumvent this problem. A virtual version log allows a maintainer of a

depending artifact to simulate a new version of an ontology it depends on without actually changing that ontology.

- Finally, we developed a set of prototype tools that form a proof-of-concept of the main ideas of the proposed ontology evolution framework. We developed two plug-ins for the Protégé ontology editor: one to automatically generate a version log for the changes applied to an ontology, and another to support the detection of changes. Furthermore, we have extended the FaCT++ reasoner in order to automatically reveal the axioms of an ontology causing a detected consistency.

1.7 Outline

The remainder of this dissertation is structured as follows:

Chapter 2 describes relevant background and related work for this dissertation. It gives an overview of ontologies and existing ontology languages. It thereby focuses in particular on the Semantic Web languages. Furthermore, it provides a short introduction to Description Logics, as they form the foundation of OWL DL, the ontology language of choice in this dissertation. The syntax and semantics of $\mathcal{SHOIN}(\mathbf{D})$, the Description Logic variant OWL DL is based upon, are described, an overview of the different reasoning tasks is provided and the exact correspondence with OWL DL is discussed. As related work, an overview of the state-of-the-art in related evolutionary domains and other ontology evolution approaches is given. Furthermore, a comparison is made between our approach and existing approaches.

Chapter 3 gives an informal overview of the ontology evolution framework that we propose in this dissertation. Attention is paid to the specific characteristics and design principles of the proposed approach. Furthermore, the different phases that together form the framework are introduced.

Chapter 4 discusses the foundations of our ontology evolution framework. A first foundation is the notion of a version log that is used to represent the evolution of an ontology. A second foundation is the Change Definition Language that makes it possible to formally define changes. The chapter also introduces the temporal logic upon which the Change Definition Language is based. Finally, the evolution log that represents an interpretation of the evolution of an ontology in terms of primitive and complex changes is presented.

Chapter 5 discusses the evaluation of conceptual change definitions for the purpose of both change requests and change detection. It also discusses the recovery of unnecessary made changes. Furthermore, it illustrates how conceptual change definitions can be defined by means of the Change Definition Language introduced in the previous chapter.

Chapter 6 focuses on maintaining ontology consistency after changes are applied and verifying backward compatibility of an ontology w.r.t. a

given depending artifact. The first part of the chapter presents a method to determine which axioms of an ontology are causing an inconsistency. Furthermore, it also presents a set of rules that ontology engineers can use to resolve a detected inconsistency. The second part of the chapter deals with verifying backward compatibility of an ontology. It introduces the notion of compatibility requirements to express the conditions an ontology should meet to be considered backward compatible w.r.t. a given depending artifact.

Chapter 7 deals specifically with ontology evolution in a decentralized environment such as the Web. We illustrate how consistency can be maintained between multiple ontologies that depend on each other, taking the decentralized characteristic of the Web into account. The chapter also deals with the problem of blocked ontologies caused by a refusal of depending artifacts to update.

Chapter 8 gives an overview of the different prototype tools that were developed as proof of concept of the main ideas presented in this dissertation.

Chapter 9 reflects on the results presented in this dissertation. A summary is provided, limitations and boundaries are stated and contributions and achievements are outlined. Finally, possible extensions and future work are discussed.

Chapter 2

Background and Related Work

The previous chapter introduced this dissertation. We presented the research context, discussed a number of problems we have identified and formulated the goal of the dissertation. Furthermore, we briefly introduced the approach that we take to achieve this goal, and discussed advantages and contributions of this approach. Finally, we gave an outline of the remainder of this dissertation.

In this chapter, we present necessary background knowledge and discuss related work, both required to place this dissertation into a broader perspective. This chapter is structured as follows. In Section 2.1, we first give a clear definition of the term *ontology*. In Section 2.2, we provide an overview of the current state-of-the-art in ontology languages, more in particular Web ontology languages. Section 2.3 shortly discusses description logics as they form the foundation of OWL DL, the language of choice in this dissertation. In Section 2.4, we discuss the related domains of temporal databases, database evolution and ontology evolution itself. In Section 2.5, we focus on specific aspects of the ontology evolution problem and make a comparison with the work presented in this dissertation. Finally, Section 2.6 concludes the chapter with a short summary.

2.1 Ontology

The term ontology comes from the Greek word 'ὄvτoç', which literally means *the study of being*. In philosophy, the term is used to refer to the branch of metaphysics occupied with the study of existence i.e., the study of what entities and what types of entities exist. In the field of computer science, the term ontology is used in the context of knowledge representation. An ontology is defined as a formal, explicit specification of a shared conceptualization of a domain of interest [30]. We further analyze this definition:

Shared conceptualization A conceptualization is an abstract, simplified representation of the world in terms of objects, concepts, and other entities that are assumed to exist and the relationships that hold among them [27]. The term shared implies that the conceptualization is shared among different stakeholders i.e., they all must agree on a specific representation of the world.

Formal and explicit specification As it is the purpose of ontologies to represent knowledge in a machine-readable and machine-interpretable manner, the conceptualization should be specified formally and explicitly. The semantics of concepts and relationships between these concepts should therefore be expressed in a formal language (e.g., description logics).

Domain of interest As it is impossible to conceptualize the whole universe of discourse (including imaginary worlds), ontologies represent only a certain part or aspect of the world. In literature, one commonly distinguishes between *domain ontologies* and *upper ontologies* (the latter is also often referred to as *core ontologies* or *foundation ontologies*). A domain ontology conceptualizes a certain subject or domain (e.g., human social relations, medical science, jurisdiction, ...), while an upper ontology specifies concepts and relations between these concepts that are generally applicable across a broad range of domains (e.g., spatial and temporal relations).

Ontologies have a strong basis in artificial intelligence and knowledge representation. Ontologies provide means for deductive reasoning and classification of knowledge, and allow systems to infer new knowledge from the knowledge explicitly stored. Furthermore, they can be used to support the sharing and communication of information between different systems. We therefore witness a grow in the appliance of ontologies in content and document management, information integration and knowledge management systems.

Moreover, ontologies play a key role in the vision of the Semantic Web. The Semantic Web is an extension of the current World Wide Web in which information is given well-defined meaning, better enabling computers and people to work in cooperation [8]. Although the Web was initially designed to be an information space for both humans and machines, it turned out that most information is only designed for human consumption. Ontologies provide the facility to turn this information into machine-understandable information. Several languages have been developed to realize the Semantic Web.

Several ontology languages have been developed in the past (e.g., OIL [20], SHOE [36], ...). However, in this dissertation we focus primarily on the Semantic Web languages and OWL in particular as it is the standard W3C

ontology language. A classification of different types of ontology languages and an overview of the Semantic Web languages is part of the discussion in the next section.

2.2 Ontology Languages

In the first subsection, we classify the different types of ontology languages, similar to the classification made by [19], based on their formal language: first-order predicate logic-based languages, frame-based languages and description logic-based languages. In the second subsection, we discuss the languages for the Semantic Web (including OWL DL).

2.2.1 Classification

We distinguish three types of ontology languages: first-order predicate logic-based languages, frame-based languages and description logic-based languages. We don't intend to provide a complete overview of existing ontology languages, but rather mention the representative languages of each of these categories.

First-order predicate logic-based languages As the name already suggests, this category of ontology languages has its foundation in first-order predicate logic. Both Cycl [57] and KIF [25, 26] are examples of first-order predicate logic-based languages. Cycl has been developed within the Cyc project[1]. The goal of the project was to build an ontology of fundamental human knowledge (called Cyc ontology). Cycl is used as representation language for this Cyc ontology. The second ontology language, KIF (*Knowledge Interchange Format*), is a language designed for use in the interchange of knowledge among disparate computer systems (created by different programmers, at different times, in different languages, ...). Being a language for knowledge interchange, KIF can also be used as a language for expressing and exchanging ontologies. An important feature of both languages is that they allow to express *knowledge about knowledge* (i.e. meta-knowledge).

Frame-based languages Frame-based languages are based on *frames* or *classes* as modeling primitives to represent a collection of instances. *Slots* or *attributes* can be associated with classes to store either primitive values, or to relate to other classes. Classes can in general be subclassed by making use of a special type of slot. Frame-based languages have a long history in Artificial Intelligence, and have been successfully applied in software engineering in the object-oriented paradigm. Examples of frame-based ontology languages are Ontolingua [18] and

[1] See `htt://www.cyc.com`

FrameLogic [50]. Ontolingua is an extension of KIF to a frame-based language; FrameLogic is a language for specifying object-oriented databases, frame systems, and logical programs [39]. A drawback of these frame-based languages is that they are in general not formally defined and the associated reasoning tools strongly depend on the implementation strategies.

Description logic-based languages *Description Logics*, or also known as *terminological logics*, describe knowledge in terms of concepts and roles. In the Description Logics research community, a lot of emphasis has been put on reasoning algorithms and their complexity. Ontologies based on Description Logics are therefore able to perform a number of reasoning tasks (e.g., instance checking, concept satisfiability, ...). Several reasoners for description logics have been implemented. Both CLASSIC [9] and LOOM [58] are examples of implemented Description Logics systems. Also OWL has its foundation in Description Logics. We discuss in more detail the OWL language in Section 2.2.2 and Description Logics in Section 2.3.

2.2.2 Semantic Web Languages

To realize the Semantic Web, a number of languages have been proposed by the W3C[2]. The Semantic Web consists of a number of layers shown in Figure 2.1 (taken from [45]). The layers of the lower part of the stack *Unicode*, *URI*, *XML + namespaces (NS) + XML schema*, *RDF + RDF Schema*, and the *Ontology vocabulary layer* (which consists of OWL) are largely in place. OWL has been a W3C Recommendation since February 2004. Research on the remaining top layers *Logic*, *Proof* and *Trust* haven't yet resulted in W3C Recommendations.

The Unicode layer ensures that all languages build on top make use of international characters sets, while the URI layer provide means for identifying resources on the Semantic Web (i.e., any resource can be referred to using a URI). All languages (including RDF(S) and OWL adopt XML (*eXtensible Markup Language*)[3] as syntax as they are built on top of the XML + NS + XML Schema layer. XML allows users to add (a tree-) structure to their documents by using a self-defined set of tags. XML Schema[4] is used to define the allowed structure and vocabulary of an XML document. Note that such an XML schema doesn't define the semantics of the tags introduced.

In the following two subsections, we discuss the languages RDF(S) and OWL.

[2] See `http://w3c.org`

[3] See `http://www.w3.org/XML/`

[4] See `http://www.w3.org/XML/Schema`

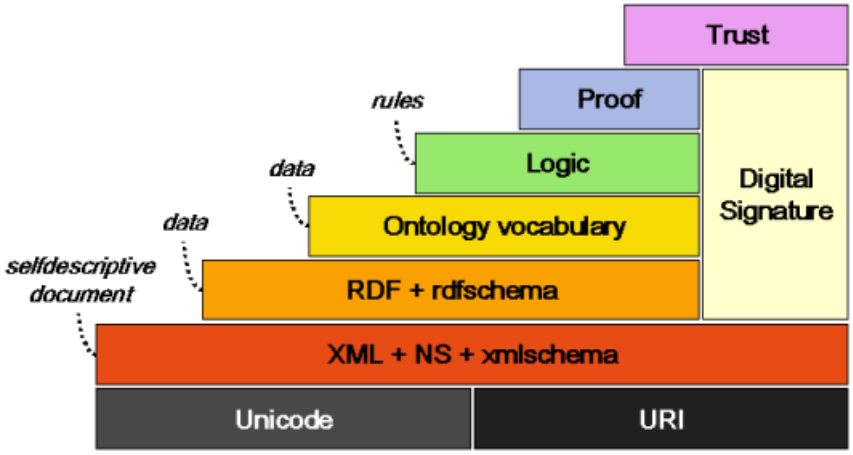

Figure 2.1: The Semantic Web layers

RDF(S)

RDF[5] stands for *Resource Description Framework* and allows users to add structured metadata to the Web. The main modeling primitives of RDF are *Resources*, *Properties* and *Statements*. A Resource can be anything one can refer to by means of a URI, and Properties describe either relations between Resources or characteristics of Resources (attributes). An RDF description consists of a number of Statements or triples of the form: subject (a Resource), predicate (a Property) and object (a Resource or Literal (e.g., string, number, date, ...)). Furthermore, RDF also introduces *Containers* and *Collections* which provide a way to group Resources. Although both Containers and Collections describe a group of Resources, their semantics are somewhat different. The semantics of a Collection specify that the listed Resources are all the members and the only members of a particular Collection (i.e., no other members exist), while Containers don't impose this restriction. Three types of Containers exist: a Bag (an unordered Container), a Sequence (an ordered Container) and an Alternative (represents a list of alternative options). Noteworthy is that RDF allows the reification of Statements i.e. treating Statements as if it were real data (intertwine the meta-data and data level). This shows great resemblance with the meta-knowledge feature in first-order predicate logic-based ontology languages.

An example of an RDF description is shown in Figure 2.2. It specifies that the editor of the Web page with given URI has as full name 'Dave Becket' and has a homepage with given URI, and that the title of the Web page is 'RDF/XML Syntax Specification (Revised)'. Note that, as is the case in XML, RDF itself doesn't allow to define the vocabulary (e.g., fullName, editor, ...) used in an RDF description. For this purpose, RDFS or RDF

[5]See `http://www.w3.org/RDF/`

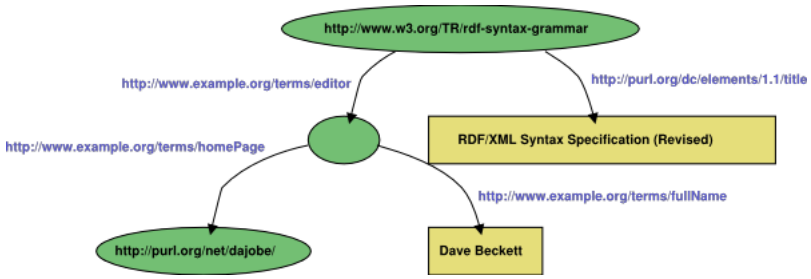

Figure 2.2: Graphical representation of an RDF example

Schema was developed.

RDFS[6] is a vocabulary description language for RDF as it allows to define a domain-specific vocabulary. RDFS provides modeling primitives to describe Classes and Properties. Classes correspond to the generic concept of Type or Category, and are similar to classes in object-oriented programming languages. RDFS also allows to define subclass relations between two classes. To describe Properties, RDFS provides means to describe both the domain and range of a property and to define property hierarchies (subproperty statements). Furthermore, resources can be defined as being an instance of a particular Class.

Note that the semantics that can be expressed by RDFS are rather limited. For example, no cardinality constraints can be expressed. In the situation where one desires to express more complex semantics, ontologies come into play. In the next subsection, we discuss the OWL Web Ontology Language, which is based on RDF and RDFS.

OWL

The direct roots of OWL[7] go back to the OIL language. OIL [20] unified three important aspects provided by different communities: formal semantics and efficient reasoning as provided by Description Logics, epistemological rich modeling primitives as provided by the Frame community, and a standard proposal for syntactical exchange notations as provided by the Web community (i.e., syntax based on XML and RDF(S)). As a result of the cooperation between the DAML project and the OIL language group, the DAML+OIL [38] language was proposed, which eventually led to the W3C Recommendation language OWL.

OWL further extends RDF(S) to provide additional machine-processable semantics for resources on the Web. It provides the following added capabilities compared to RDF(S):

[6] See `http://www.w3.org/TR/rdf-schema/`
[7] See `http://www.w3.org/2004/OWL/`

- Cardinality constraints on Properties (e.g., a person has *exactly one* name);

- Value constraints on Properties using *all-values* and *some-values* constructs (e.g., for a `SoccerTeam`, all values of the `hasPlayer` Property must be an instance of `SoccerPlayer`);

- Transitive, symmetric and inverse Properties;

- Equivalence between Classes, Properties or Instances (e.g., the Classes `Aircraft` and `Plane` are equivalent);

- Constructs to combine Classes (i.e., define Classes as union, intersection or complement of other Classes; define Classes to be disjoint with other Classes);

- Constraints on domain and range of specific Class-Property combinations (e.g., a Property `hasPlayer` has for a Class `SoccerTeam` 11 players, while it has only 5 values for a Class `BasketballTeam`).

OWL provides three increasingly expressive sublanguages (i.e., OWL Lite, OWL DL and OWL Full), targeted at different types of users:

- **OWL Lite** supports those users primarily needing a classification hierarchy and simple constraints (e.g., OWL Lite permits only cardinality constraints of values 0 or 1).

- **OWL DL** supports users who want the maximum expressiveness while retaining computational completeness and decidability. The abbreviation DL refers to its formal foundation on Description Logics. The characteristics of computational completeness and decidability of course also apply to OWL Lite as it is a subset of OWL DL. Note that OWL DL (and OWL Lite) add a number of restrictions to the syntax of RDF(S). Most importantly, individuals and classes are clearly separated, just as individuals and concepts are also separated in Description Logics.

- **OWL Full** is meant for users who want maximum expressiveness and the syntactic freedom of RDF(S). The drawback is that no computational guarantees can be assured. Note that OWL Full allows for example Classes to be simultaneously a collection of individuals and an individual in its own right. This means that, in contrast to OWL DL and OWL Lite, OWL Full facilitates meta-modeling as found in RDFS (e.g., attaching property instantiations to Classes).

2.3 Description Logics

Description Logics (DLs) are a family of knowledge representation languages that can be used to represent the knowledge of an application domain in a structured and formally well-understood way [5]. DLs describe application domains by means of concept descriptions i.e., expressions that are built from atomic concepts and atomic roles using the concept and role constructors provided by the particular DL. DLs differ from their predecessors in that they are equipped with a formal, logic-based semantics. For a detailed overview of DLs, we refer the interested reader to [4].

The DL variant of OWL conforms to the $\mathcal{SHOIN}(\mathbf{D})$ Description Logic, while OWL Lite conforms to less expressive $\mathcal{SHIF}(\mathbf{D})$ variant [40]. As the former variant includes the latter, we mainly concentrate on the former in this dissertation. In Section 2.3.1, we briefly introduce syntax and semantics of the $\mathcal{SHOIN}(\mathbf{D})$ variant, the correspondence with OWL DL is clarified in Section 2.3.2. Finally, Section 2.3.3 discusses the common reasoning tasks of DLs.

2.3.1 Syntax and Semantics

In this dissertation, we adopt the following convention: A is an atomic concept, C is a complex concept, R is an abstract role and S is an abstract simple role[8], U is a concrete role, D is a datatype, o is a nominal or individual, v is a data value and n is a non-negative integer. The semantics of $\mathcal{SHOIN}(\mathbf{D})$ is given by an interpretation $\mathcal{I} = \langle \Delta^{\mathcal{I}}, \Delta^{\mathbf{D}}, \cdot^{\mathcal{I}}, \cdot^{\mathbf{D}} \rangle$ with a non-empty abstract domain $\Delta^{\mathcal{I}}$, disjoint from the concrete domain $\Delta^{\mathbf{D}}$. Furthermore, it consists of the functions $\cdot^{\mathcal{I}}$ and $\cdot^{\mathbf{D}}$ so that $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, $U^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathbf{D}}$, $o^{\mathcal{I}} \in \Delta^{\mathcal{I}}$. The syntax and semantics of the concept descriptions are given in Table 2.1.

Based on the aforementioned syntax, different types of axioms can be formed: concept definition axioms $C_1 \equiv C_2$, concept inclusion axioms $C_1 \sqsubseteq C_2$, role definition axioms $R_1 \equiv R_2$, role inclusion axioms $R_1 \sqsubseteq R_2$, transitivity axioms $Trans(R)$, concept assertions $C(a)$, role assertions $R(a, b)$, individual equalities $o_1 = o_2$ and individual inequalities $o_1 \neq o_2$. The syntax and the semantics of the different types of axioms are given in Table 2.2.

2.3.2 Correspondence with OWL

An ontology corresponds to the notion of a knowledge base in Description Logics. A knowledge base compromises two components: the TBox and the ABox. The TBox introduces the *terminology*, i.e., the vocabulary of an

[8] A role is simple if it is neither transitive nor has any transitive subroles. Note that it is required to restrict number restrictions to simple roles in order to yield a decidable logic [41].

Description	**Syntax**	**Semantics**
Conjunction	$C_1 \sqcap C_2$	$(C_1 \sqcap C_2)^{\mathcal{I}} = C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$
Disjunction	$C_1 \sqcup C_2$	$(C_1 \sqcup C_2)^{\mathcal{I}} = C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}}$
Negation	$\neg C$	$(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \backslash C^{\mathcal{I}}$
Exists restriction	$\exists R.C$	$(\exists R.C)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} \mid \exists b.(a,b) \in R^{\mathcal{I}}$ and $y \in C^{\mathcal{I}}\}$
Value restriction	$\forall R.C$	$(\forall R.C)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} \mid \forall b.(a,b) \in R^{\mathcal{I}} \rightarrow b \in C^{\mathcal{I}}\}$
Atleast restriction	$\geq nS$	$(\geq nS)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} \mid \#(\{b \mid (a,b) \in S^{\mathcal{I}}\}) \geq n\}$
Atmost restriction	$\leq nS$	$(\leq nS)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} \mid \#(\{b \mid (a,b) \in S^{\mathcal{I}}\}) \leq n\}$
One of	$\{o_1, \ldots, o_n\}$	$\{o_1, \ldots, o_n\}^{\mathcal{I}} = \{o_1^{\mathcal{I}}, \ldots, o_n^{\mathcal{I}}\}$
Datatype exists	$\exists U.D$	$(\exists U.D)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} \mid \exists b.(a,b) \in U^{\mathcal{I}}$ and $y \in D^{\mathbf{D}}\}$
Datatype value	$\forall U.D$	$(\forall U.D)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} \mid \forall b.(a,b) \in U^{\mathcal{I}} \rightarrow b \in D^{\mathbf{D}}\}$
Datatype atleast	$\geq nU$	$(\geq nU)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} \mid \#(\{b \mid (a,b) \in U^{\mathcal{I}}\}) \geq n\}$
Datatype atmost	$\leq nU$	$(\leq nU)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} \mid \#(\{b \mid (a,b) \in U^{\mathcal{I}}\}) \leq n\}$
Datatype one of	$\{v_1, \ldots, v_n\}$	$\{v_1, \ldots, v_n\}^{\mathcal{I}} = \{v_1^{\mathcal{I}}, \ldots, v_n^{\mathcal{I}}\}$
Inverse role	R^-	$(R^-)^{\mathcal{I}} = \{(b,a) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid (a,b) \in R^{\mathcal{I}}\}$

Table 2.1: Syntax and semantics of concept descriptions

Description	**Syntax**	**Semantics**
Concept definition	$C_1 \sqsubseteq C_2$	$C_1^{\mathcal{I}} = C_2^{\mathcal{I}}$
Concept inclusion	$C_1 \sqsubseteq C_2$	$C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$
Role definition	$R \sqsubseteq S$	$R^{\mathcal{I}} = S^{\mathcal{I}}$
Role inclusion	$R \sqsubseteq S$	$R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$
Transitivity	$Trans(R)$	$R^{\mathcal{I}} = (R^{\mathcal{I}})^+$
Concept assertion	$C(a)$	$a^{\mathcal{I}} \in C^{\mathcal{I}}$
Role assertion	$R(a,b)$	$(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$
Individual equality	$a = b$	$a^{\mathcal{I}} = b^{\mathcal{I}}$
Individual inequality	$a \neq b$	$a^{\mathcal{I}} \neq b^{\mathcal{I}}$

Table 2.2: Syntax and semantics of axioms

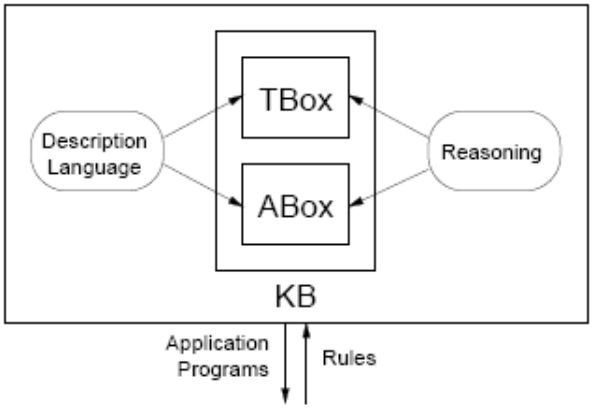

Figure 2.3: Architecture of a knowledge representation system based on Description Logics

application domain, while the ABox contains *assertions* about individuals in terms of this vocabulary. The descriptions of concepts, roles and individuals allowed in a knowledge base depends on the particular description logic used. For OWL DL ontologies, the descriptions must follow the syntax of $\mathcal{SHOIN}(\mathbf{D})$ as discussed in the previous section. Furthermore, Description Logic systems provide services to reason about the descriptions contained in the knowledge base. We give an overview of the supported reasoning tasks in the next subsection (see Section 2.3.3). Figure 2.3 (taken from [4]) shows the architecture of a knowledge representation system based on Description Logics.

In the remainder of this section, we give an overview of the correspondence between the OWL DL syntax and the $\mathcal{SHOIN}(\mathbf{D})$ syntax (see Table 2.3). Note that the concept descriptions and axioms involving concrete roles are not explicitly listed in the table as they don't require a different syntax then is used for abstract roles.

2.3.3 Inferencing

The purpose of a DL knowledge base goes beyond just storing concept definitions and assertions (see Figure 2.3). The formal semantics of Description Logics make it possible to turn implicit knowledge into explicit knowledge though inferencing. We first discuss the key *reasoning tasks* for a DL knowledge base (focusing on both the TBox and the ABox), after which we end this section explaining the difference between *closed- and open-world assumption*, thereby highlighting its effect on the semantics of a DL knowledge base.

OWL	DL
intersectionOf	$C_1 \sqcap \ldots \sqcap C_2$
unionOf	$C_1 \sqcup \ldots \sqcup C_2$
complementOf	$\neg C$
oneOf	$\{o_1, \ldots, o_n\}$
allValuesFrom	$\forall R.C$
someValuesFrom	$\exists R.C$
hasValue	$\exists R.\{o\}$
minCardinality	$\leq nR$
maxCardinality	$\geq nR$
cardinality	$(\leq nR) \sqcap (\geq nR)$
subClassOf	$C_1 \sqsubseteq C_2$
equivalentClass	$C_1 \equiv C_2$
disjointWith	$C_1 \sqsubseteq \neg C_2$
subPropertyOf	$R \sqsubseteq S$
equialentProperty	$R_1 \equiv R_2$
domain	$\exists R.\top \sqsubseteq C$
range	$\top \sqsubseteq \forall R.C$
FunctionalProperty	$\top \sqsubseteq \leq nR$
InverseFunctionalProperty	$\top \sqsubseteq \leq nR^-$
inverseOf	$R_1 \equiv R_2^-$
TransitiveProperty	$Trans(R)$
SymmetricProperty	$R \equiv R^-$
sameAs	$o_1 = o_2$
differentFrom	$o_1 \neq o_2$

Table 2.3: Correspondence between OWL DL and $\mathcal{SHOIN}(\mathbf{D})$ syntax

Reasoning Tasks

Description Logics have been the result of a tradeoff between expressiveness and computational complexity of reasoning. One of the goals of Description Logics is to provide reasoning procedures that are sound and complete. In this section, we briefly discuss the key reasoning tasks for both the TBox and the ABox.

For a TBox $\mathcal{T}$, the following reasoning tasks can be performed for concepts of $\mathcal{T}$: *satisfiability*, *subsumption*, *equivalence* and *disjointness* checking:

- **Satisfiability** A concept C is satisfiable w.r.t. $\mathcal{T}$ if there exists a model $\mathcal{I}$ of $\mathcal{T}$ such that $C^{\mathcal{I}} \neq \emptyset$.
- **Subsumption** A concept C_1 is subsumed by a concept C_2 w.r.t. $\mathcal{T}$ if $C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$ for every model $\mathcal{I}$ of $\mathcal{T}$.
- **Equivalence** Two concepts C_1 and C_2 are equivalent w.r.t. $\mathcal{T}$ if $C_1^{\mathcal{I}} = C_2^{\mathcal{I}}$ for every model $\mathcal{I}$ of $\mathcal{T}$.
- **Disjointness** Two concepts C_1 and C_2 are disjoint w.r.t. $\mathcal{T}$ if $C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}} = \emptyset$ for every model $\mathcal{I}$ of $\mathcal{T}$.

For an ABox $\mathcal{A}$, the following reasoning tasks are generally considered, i.e., *ABox consistency checking* and *instance checking*, although other reasoning tasks are possible, e.g., the *retrieval problem* (find all individuals a that are instances of a particular concept) and the *realization problem* (find the most specific concept an individual a is an instance of). We explain both the ABox consistency checking and instance checking task below:

- **ABox consistency** An ABox $\mathcal{A}$ is consistent w.r.t. a TBox $\mathcal{T}$ if there is an interpretation $\mathcal{I}$ that is a model of both $\mathcal{A}$ and $\mathcal{T}$.
- **Instance checking** An individual a is an instance of a concept C w.r.t. $\mathcal{T}$ if $a^{\mathcal{I}} \in C^{\mathcal{I}}$ for every model $\mathcal{I}$ of $\mathcal{T}$.

Note that all of the above mentioned reasoning tasks can be reduced to one single inference problem: the ABox consistency task. For more details, we refer the interested reader to [4].

Closed- and Open-world Assumptions

Although an analogy between DL knowledge bases and databases is often drawn, i.e., where a database schema is comparable to a TBox and the instance data of the database is comparable to an ABox, the semantics of ABoxes and database instances are in general quite different. Instance data in a database is assumed to be complete, and absence of information is

interpreted as negative information. The semantics of databases are therefore characterized as *closed-world semantics*. The information in ABoxes is in general viewed as being incomplete, and absence of information only indicates lack of knowledge. We therefore say that the semantics of DL knowledge bases are *open-world semantics*.

Consider as example the following assertion: `hasColleague (PETER, SVEN)`. In the case that this assertion would be the only assertion in a database, it would be interpreted as the fact that Peter only has one colleague, Sven. In an ABox, this assertion means that Peter has a colleague Sven, but we don't know whether it is the only colleague or not (different interpretations exist). This difference in semantics has also consequences on whether a database or DL knowledge base is considered 'consistent'. Take for example the assertions `hasColleague(PETER, SVEN)` and $(\geq 2$ `hasColleague)(PETER)`. We would consider a database storing these assertions to be inconsistent as the second assertion states that Peter has at least two colleagues, although only one colleague exists, Sven. On the other hand, an ABox storing the very same assertions would be considered rightly consistent as the set of assertions state that Peter has at least two colleagues, one of these colleagues is Sven and the other colleague(s) are unknown.

2.4 Related Domains

When we consider research in the field of the Semantic Web on *ontology management* – a sub domain which includes subjects as ontology creation, ontology evolution and ontology evaluation – probably the most emphasis has been put on the problem of ontology creation. Research in this area has lead to a number of ontology engineering methodologies. Some of the most well-known methodologies are DILIGENT [72], Dogma [47] and METHONTOLOGY [21]. While these approaches focus their attention on the overall engineering process and the methodology behind it, the focus of this dissertation is on the specific problems of ontology evolution. We therefore focus our related work on related domains concerning ontology evolution.

The problem of evolution has been a long term research topic in various domains and still continues to be an important research topic today. Although evolution has been investigated in a variety of diverge research domains (e.g., software evolution in the field of software engineering [56]), we restrict ourselves to those domains most relevant for the topic of this dissertation. We reckon the following domains as most relevant: temporal databases, database evolution and - of course - the domain of ontology evolution itself.

This section is structured as follows. In Section 2.4.1, we discuss related work in the field of temporal databases. In Section 2.4.2, we present related work on database evolution and versioning. Finally, in Section 2.4.3, we

discuss related work in the domain of ontology evolution, focusing on the different approaches proposed. Note that specific aspects concerning ontology evolution that are of importance for this dissertation are discussed in the subsequent section (see Section 2.5).

2.4.1 Temporal Databases

Although research on temporal databases is typically omitted from discussions about the related work on ontology evolution, we deliberately include it here as a number of the ideas we will present in this dissertation are founded in the domain of temporal databases. Temporal databases provide a uniform and systematic way of dealing with historical data [13]. Conventional databases are designed to capture only one single state i.e., the current state. When new data become available through updates, the existing data values are removed from the database as these databases only capture a snapshot of reality. Although conventional databases serve some applications well, they are insufficient for those applications in which past and/or future data are also required. Temporal databases are databases that fully support the storage and querying of information that varies over time. Temporal databases typically allow a database designer to differentiate between temporal and non-temporal attributes [67]. The temporal database maintains the state history of temporal attributes using timestamps to specify the time during which a temporal attribute's value is valid.

Three temporal aspects are mostly discussed in literature when it comes to the temporal extent of objects. Snodgrass and Ahn [82] defined the following aspects:

- **Valid time.** A valid time of an object is the time when the object is true in the modeled reality.
- **Transaction time.** A transaction time of an object is the time when the object is part of the current state of the database.
- **User-defined time.** A user-defined time is a time attribute which is not interpreted by the database management system. For example, birth date does not refer to when an object became true in the modeled reality nor does it record when an object became valid in a database. It is an attribute of an entity that happens to be a date.

For the interested reader, a detailed survey about temporal databases can be found in [67] and [29].

Besides research on temporal databases itself, a lot of effort has also been put on researching conceptual models for temporal databases. Compared to conceptual models for conventional databases (e.g., ER (Entity-Relationship) models [12] and ORM (Object Role Modeling) [34]), concep-

tual models for temporal databases add design primitives to support temporal aspects. The majority of research done on conceptual models for temporal databases has focused on extensions to standard relational data models (in particular ER models). Furthermore, research has also focused on object-oriented approaches and event-based models. The authors of [29] present an overview of the most important temporal extensions to ER. These extensions to ER models allow database designers to express entities to be either temporary entities (may change over time) or snapshot entities (don't change over time). For object-oriented approaches, we refer to MADS (Modeling for Application Data with Spatio-temporal features) [68] as a representative example. MADS supports both the time stamping of objects, attributes and relationships, as well as the representation of temporal aspects between objects (e.g., this object has created another object). Finally, event-based models seek to define and record the events that change the state of the system being modeled. Event-based models differ from other approaches because it doesn't seek to record past states of a system, but it seeks the events that change the state. A representative example of an event-based model is the TEERM (Temporal Event Entity Relationship Model) [16].

2.4.2 Database Schema Evolution & Versioning

The problem of database schema evolution & versioning have been studied extensively in the past. This problem has been investigated primarily for object-oriented databases and to a lesser extent for relational databases. According to [17], the difference between (database) schema evolution & versioning is defined as follows:

- **Schema evolution:** a database system supports schema evolution if it facilitates the modification of the database schema without the loss of existing data.
- **Schema versioning:** a database system supports schema versioning if it facilitates the querying of all data through user-definable version interfaces.

The authors of [24] argue that schema evolution can be considered a special case of schema versioning where only the current schema version is retained. A detailed survey on the various issues of schema evolution & versioning can be found in [78]. As becomes clear by this survey, research in database schema evolution & versioning has mainly focused on the problems of *semantics of change* and *change propagation*. The former refers to the problem of understanding changes and their effect on the database schema in order to maintain schema consistency, the latter refers to the problem of propagating the changes to the actual data in order to maintain consistency with the modified schema. Although other issues were investigated (e.g., data

conversion, access rights, ...), these topics are less relevant as related work for this dissertation. We discuss the problems of semantics of change and change propagation below.

Two different approaches have been proposed to solve the problem of semantics of change. The first approach introduces invariants and rules. Invariants define conditions that need to be met in order to consider a schema consistent, rules are used to restore consistency when invariants are not met after a change. This kind of approach is applied in systems as ORION [7] and O_2 [22]. The second approach is based on the introduction of axioms formalizing the dynamic schema evolution. These axioms (with an inference mechanism) ensure that a schema evolves into a consistent version, without actually having to check for inconsistencies. E.g., the authors of [71] propose an axiomatic model for dynamic schema evolution for object-based systems. The model allows to infer all schema relations based on two input sets i.e., essential supertypes and essential properties. Although the second kind of approach works for primitive changes, the use of complex changes no longer guarantees schema consistency after a change [10].

Most solutions for the change propagation problem rely on conversion functions to adapt the instance data to the changed database schema (e.g., [7] and [70]). In most cases, simple built-in conversion mechanisms can be used. Sometimes, user-supplied conversion functions must be defined for non-trivial object conversions.

Another approach followed by [3] is based on description logics for modeling dynamic information, and covers the combination of temporal database modeling and database schema evolution. They introduced a temporal conceptual data model able to represent time varying data (similar to the temporal conceptual models mentioned in the previous section). Furthermore, they introduced an object-oriented conceptual data model enriched with schema change operators, which are able to represent the explicit temporal evolution of the schema while maintaining a consistent view on the data. Both data models are encoded in Description Logics. The advantage of using a Description Logic to formalize a conceptual data model lies on the fact that complete logical reasoning can be employed using an underlying DL inference engine to infer implicit facts and to manifest any inconsistencies.

2.4.3 Ontology Evolution

In this section, we give a general overview of the state-of-the-art in the domain of ontology evolution. We discuss the differences between ontology evolution and database schema evolution, and focus on different existing ontology evolution approaches. Note that we only present a general overview of these approaches. We only go into more detail about specific aspects of the ontology evolution problem that are relevant for this dissertation in the following section (see Section 2.5).

Definition

In [83], the author introduced the following definition for the term 'ontology evolution':

Definition (Ontology Evolution). *Ontology Evolution is the timely adaptation of an ontology to the arisen changes and the consistent propagation of these changes to dependent artifacts.*

According to the autor, the ontology evolution process encompasses the set of activities that ensures that the ontology continues to meet organizational objectives and users' needs in an efficient and effective way. The author recognizes that changes in an ontology can cause inconsistencies in other parts of the ontology, as well as in depending artifacts. However, the author seems to define ontology evolution from the viewpoint of a centralized environment as she relies on propagation of changes to maintain consistency. In this dissertation, we argue that the mechanism of propagation of changes no longer holds in a decentralized environment (see Chapter 7).

In [83], the author also makes a clear distinction between *ontology modification*, *ontology evolution* and *ontology versioning*, similar to the terminology from the database community (see Section 2.4.2). Note however that the interpretation of ontology evolution & versioning rather differs from the interpretation given to database schema evolution & versioning. The difference between the three aforementioned terms is defined as follows:

- **Ontology modification** is accommodated when an ontology management system allows changes to the ontology that is in 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.

The author of [51], on the other hand, argues that the traditional distinction between evolution and versioning is no longer applicable for ontologies. Depending artifacts of an ontology cannot be forced to update to a new version, so different depending artifacts are likely to depend on different versions of the same ontology. This means that multiple versions of the same ontology are bound to exist and must be supported. In this dissertation, we reconcile both views. Although we maintain a clear distinction between ontology evolution & versioning in a similar fashion as defined by [83], we recognize that past versions of an ontology must remain accessible even when we consider ontology evolution. Similar to [24] in the database

community, also ontology evolution can be seen as a special case of ontology versioning where only the latest version is retained. Note that in the approach proposed in this dissertation, we explicitly focus on the problem of ontology evolution.

In [64], the authors make a comparison between ontology evolution and database schema evolution, where they identify a number of important differences between both. A first difference is that ontologies are data too e.g., Classes can form the result of a query on an ontology. A second difference is that ontologies are logical systems that themselves incorporate formal semantics. Thirdly, ontologies are intended to be reused and extended by other ontologies. A fourth difference is that ontologies are decentralized by nature i.e., we can neither know who uses an ontology, how many users there are, nor prevent or require users to use a particular ontology (version). A fifth difference is that ontologies are in general much richer than a typical database schema. Finally, in ontologies something can be both an instance and a class at the same moment in time, whereas in databases a clear distinction is made between schema and data. Note that, when considering OWL, this last difference only holds for OWL Full, as OWL Lite and OWL DL also keep a clear distinction between classes and instances.

In the following two sections, we give an overview of the proposed approaches in literature on respectively ontology evolution and ontology versioning.

Ontology Evolution

The authors of [51] [52] [53] [54] propose a component-based approach supporting ontology evolution for the OWL ontology language. This framework consists of the following four main components: a meta-ontology of change operations, complex change operations, transformation sets, and the specification of relations between different ontology versions. The meta-ontology of changes defines a set of basic change operations that the different users of this framework need to agree upon. A standard set of basic changes is proposed for the OWL ontology language. Besides basic change operations, the framework also proposes complex change operations. Complex change operations provide a mechanism for grouping a number of basic change operations that together constitute a logical entity. Another important element of the framework is the notion of a transformation set. A transformation set is a set of change operations that specifies how an old version of an ontology can be transformed into a new version. A transformation set may contain both basic and complex change operations. A minimal transformation set contains only change operations that are necessary and sufficient to specify a transformation. The fourth component of the framework is the specification of how two ontology versions are related. This specification can be seen as a mapping ontology and consists of the following (optional) information:

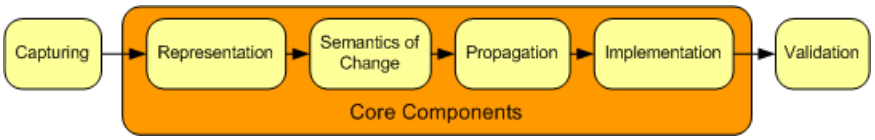

Figure 2.4: Six-phase ontology evolution framework

- **Descriptive meta-data:** information like date of release, the author of the changes, ...
- **Minimal transformation set:** a complete specification of a change that can be used to re-execute the change, to translate or re-interpret data sets, and as a basis for deriving additional information about the change.
- **Conceptual relations:** the relation between concepts across versions (e.g., the new version of a Class is a subclass of the old version of that Class).
- **Complex changes:** allow a higher-level description of changes giving a clearer view on the evolution of an ontology.
- **Change rationale:** specifies the intention behind a change (e.g., a fix of an error, an update to the real world situation, ...).

The focus of attention of the framework is mainly on the problem of identifying changes and providing semantic specifications for the changes between different versions. The framework provides methods for finding changes when only two versions of an ontology are available, and methods for deriving additional change information. Note that these last methods resemble the same goal as the change detection phase in our ontology evolution framework, although the approach taken by both differs fairly as we will discuss in Section 2.5.2. The framework itself doesn't take into account the problem of consistency maintenance after changes are applied.

Another ontology evolution approach is proposed by [60] [61] [83] [84] [85]. The authors propose a six-phase ontology evolution framework for the KAON ontology language [63] consisting of the following phases: *change capturing phase*, *change representation phase*, *semantics of change phase*, *change propagation phase*, *change implementation phase*, and *change validation phase*. Figure 2.4 gives an overview of the different phases of the ontology evolution framework. The first phase, the *change capturing phase*, captures changes either from explicit requirements or from the result of change discovery methods. They distinguish structure-driven change discovery (e.g., a concept without Properties is a candidate for deletion), data-driven change discovery (e.g., a concept without instances may be deleted), and usage-driven change discovery (e.g., concepts that are never queried

may be deleted). The second phase, the *change representation phase*, distinguishes between elementary and composite changes, similar to the previous approach. The purpose of the *semantics of change phase* is to resolve possible inconsistencies introduced after a change. The task of the *change propagation phase* of the ontology evolution process is to bring automatically all dependent artifacts into a consistent state after an ontology update has been performed. The role of the next phase of the ontology evolution framework, the *change implementation phase* is (1) to inform an ontology engineer about all consequences of a change request, (2) to apply all the changes and (3) to keep track about performed changes. Finally, the *change validation phase*, enables justification of performed changes and undoing them at user's request. It has as purpose to increase the usability of the evolution process. Compared to the previously discussed ontology evolution framework, this framework concentrates more on ontology consistency checking and inconsistency resolving, although the focus is on structural consistency. The framework, however, doesn't provide mechanisms for detecting complex changes.

The authors of [31] [32] [33] have taken the previously discussed framework as basis and extended it to support evolution for the OWL ontology language, instead of solely supporting the KAON ontology language. Their work has mainly focused on handling inconsistencies introduced by a change to an ontology. Checking consistency and resolving inconsistencies for OWL is quite different then for the KAON ontology language mainly due to the difference between respectively the open-world vs. closed-word assumption taken by both languages. They focus on different forms of consistency, including structural consistency, logical consistency and user-defined consistency. In the occurrence that an inconsistency cannot be resolved, they propose an approach to reason with an inconsistent ontology. Another extension concerns the support for collaborative and usage-driven evolution of ontologies. This extension targets personal ontologies i.e., ontologies that are shared among different users, but that are customized and personalized to a specific user. So, every user may have a slightly different version of the original ontology. Their approach allows users to annotate concepts and axioms of the their ontology with a rating (both positive and negative) indicating the importance that a user attached to the concept or axiom. Based on the ratings given by various users, recommendations of ontology changes are suggested to other users.

A last ontology evolution approach that we discuss here is the approach proposed by [23]. They present as a solution to the problem of ontology evolution the incorporation of results and intuitions from the field of belief change, which deals with the adaptation of knowledge stored in knowledge bases to new information. The authors focus on the AGM theory, which is the most influential approach in the field of belief change. They have investigated, besides other Description Logic variants, whether OWL is compatible

with the AGM approach. The outcome of this study is that a few Description Logic variants are indeed compatible with the AGM theory. Unfortunately, the list of compatible DLs excludes a large number of Description Logic variants, among which OWL Lite, OWL DL, and OWL Full. As a consequence, this outcome hinders the approach proposed as a complete solution for the OWL ontology language.

Ontology Versioning

In [36], the authors discuss the problems associated with managing ontologies in a distributed environment as the Web. They proposed SHOE (Simple HTML Ontology Extensions), a small extension to HTML which allows web page authors to annotate their web documents with machine-readable knowledge. An important feature of SHOE is its support for multiple versions of ontologies. The paper discusses the problem of ontology versioning, the effects of ontology revision on SHOE web pages, and methods for implementing ontology integration using SHOEs extension and version mechanisms.

The authors of [44] propose an approach to reason on different versions of the same ontology in order to verify compatibility for existing applications. Similar to our approach, the proposed approach is based on a temporal logic. Although their approach allows to reason on different versions, they don't allow to specify compatibility requirements that can be used to automatically check compatibility of an ontology version with a previous version. Furthermore, the proposed approach requires that previous ontology versions are explicitly retained, which is not a requirement of the approach proposed in this dissertation. Also, the authors only consider the addition and deletion of subsumption relations between different versions, so the detection of more complex changes is not considered. Finally, they don't propose a temporal logic-based language to define changes that can be used by ontology engineers both for requesting changes and detecting additional changes.

In the next section (Section 2.5), we go into more details concerning specific aspects of the ontology evolution process that are relevant for this dissertation.

2.5 Ontology Evolution Specific Aspects

In this section, we provide further details concerning specific aspects of the ontology evolution process that other approaches have dealt with and that are relevant for this dissertation. We consider the following aspects: change representation, change detection, consistency checking & inconsistency resolving, and ontology evolution in a distributed and decentralized environment. For these different aspects, we make a comparison between related approaches and our approach.

2.5.1 Change Representation

In [83], Stojanovic presents the use of an *evolution log* to represent changes between two versions of an ontology. An evolution log tracks the history of an ontology as an ordered sequence of ontology changes. An evolution log is expressed in terms of an evolution ontology. The evolution ontology is a common shared model of ontology changes. It distinguishes between elementary changes and composite changes. Composite changes are expressed as a composition of elementary changes and other composite changes. The author provides a number of example elementary and composite changes for the KAON ontology language. An evolution log is then built up from instantiations of the concepts defined in the evolution ontology.

The author of [51] represents changes between two versions of an ontology as a kind of mapping between both versions. As already mentioned in the previous section, this mapping consists of a variety of information: descriptive meta-data, a minimal transformation set, conceptual relations, complex changes and change rationale. Both the minimal transformation set and the complex changes are expressed in terms of change operations. Similar to [83], the author distinguishes between basic change operations and complex change operations, the difference being that these change operations are defined for OWL instead of the KAON ontology language. The author provides a complete set of basic change operations for OWL. Complex change operations are expressed as a composition of other change operations. The minimal transformation set in the mapping consists of only basic change operations.

Our approach differs in this aspect from the aforementioned approaches in the sense that we represent the evolution of an ontology by means of a *version log*, and that we use this version log to automatically create an evolution log. In contrast to an evolution log, a version log stores for each concept ever defined in the ontology, the different versions it passes through during its life cycle: from its creation, over its modifications, until its retirement. We define changes in terms of a temporal language w.r.t. the version log. Contrary to the aforementioned approaches, the different changes are formally defined. The advantage of using both a version log and an evolution log is that different interpretations (in terms of changes) for the same evolution can be specified. Just as different views may exist on the same domain, so may there exist different views on the evolution of a domain. Furthermore, when relying on an evolution log, it turns out to be more difficult to enrich the version log with additional complex changes (as we will see in the next section).

2.5.2 Change Detection

The author of [51] proposes an approach to find complex change operations in order to enrich the mapping between two ontology versions. He introduced a detection mechanism based on rules and heuristics to detect complex change operations between two ontology versions (V_{old} and V_{new}). While their approach is applicable in specific cases, in general, the approach has serious limitations:

- The approach requires that V_{old} is still available, because detection rules rely on both V_{old} and V_{new}. Unfortunately, when an ontology is modified, the original version is often no longer available.

- Multiple changes to V_{old} may interfere, thereby possibly invalidating defined change detection rules. This would mean that the rule, as formulated, no longer applies.

The authors of [51] try to overcome these problems by introducing heuristics to change the precise criteria of the rules to approximations. While heuristics may provide the ontology engineer with some flexibility in the rule definitions, it is clear that it doesn't offer a bullet-proof solution as it makes the detection process imprecise and unpredictable.

The change detection mechanism of our approach doesn't rely on rules and heuristics, instead it is based on the fact that changes are formally defined in terms of a temporal logic-based language. Therefore, occurrences of change definitions can be detected by evaluating change definitions as temporal queries on a version log. As a version log describes the evolution of an ontology, the result of the temporal queries provides information concerning the kind of changes that have occurred.

2.5.3 Consistency Checking & Inconsistency Resolving

On the topic of dealing with consistency checking & inconsistency resolving in the context of evolving ontologies, only little research has been done. In the ontology evolution framework proposed by [83], both the task of consistency checking as well as inconsistency resolving are targeted for the KAON ontology language. Consistency checking is based on a set of invariants, while inconsistencies can be resolved by applying a set of rules. As an inconsistency can, in general, be resolved in different ways, the author proposes the notion of evolution strategies. An ontology engineer uses these evolution strategies to indicate how different types of inconsistencies need to be resolved. However, the focus of the ontology evolution framework rests on structural consistency, while logical consistency is mainly left out of the discussion. The focus in this dissertation is on maintaining logical consistency.

The authors of [32] discusses three different forms of consistency: structural consistency, logical consistency and user-defined consistency[9]. To resolve structural inconsistencies, they present an approach based on rewrite rules that allows to transform axioms into the desired variant of OWL (possibly with a loss of semantics). To resolve logical inconsistencies, they present two alternative approaches to localize an inconsistency for an OWL ontology based on the notion of respectively a maximal consistent subontology and a minimal inconsistent subontology. Inconsistencies are checked using the well-known OWL reasoning features. A maximal consistent subontology is the largest ontology that is still consistent. While this approach resolves an inconsistency, it can hardly be called advanced as it solely relies on removing axioms. The second approach, using a minimal inconsistent subontology, is very similar to the concept of a MUPS (Minimal Unsatisfiability Preserving Sub-TBox) introduced by [79]. Both the minimal inconsistent subontology and the MUPS represent the smallest set of axioms forming an inconsistent ontology, and reveals the axioms causing the detected inconsistency. Although removing one axiom from the minimal inconsistent subontology will resolve an unsatisfiable concept, it can not be guaranteed that this will solve the true cause of the inconsistency (as we will discuss in Chapter 6). Furthermore, the approach only treats axioms as a whole, while only specific parts of an axiom may be the cause of an inconsistency.

The authors of [46] present an approach to check for unsatisfiability in ORM conceptual schemes within the DOGMA approach [47] where the ORM conceptual schemes are used as a representation language for ontologies. Their approach relies on the definition of a number of heuristics in order to identify unsatisfiabilities in (a restricted form of ORM schemes). A drawback of their approach is that, because the approach relies on a number of heuristics, the unsatisfiability checking is by no means complete.

Other related work has been carried out in explaining inconsistencies found in OWL ontologies. Interesting to mention is the work of [6]. They present a Symptom Ontology that aims to serve as a common language for identifying and describing semantic errors and warnings. Note that the aim of the Symptom Ontology is not to identify the cause of the ontology nor to offer possible solutions to resolve an inconsistency.

Related to the topic of explaining inconsistencies, is the research field of ontology debugging. The aim of this research field is to provide the ontology engineer with a more comprehensive explanation of the cause of the inconsistency than is generally provided by 'standard' ontology reasoners, and possibly suggest solutions to overcome the ontology inconsistency. In general, two different types of approaches are distinguished in literature: black-box vs. glass-box approaches. The former treats the OWL reasoner

[9] Although user-defined consistency can be seen as a form of structural consistency (see Chapter 6).

as a 'black-box' and uses standard inference to locate the source of the inconsistency, the latter modifies the internals of the reasoner to reveal the cause of the problem.

The authors of [89] propose a black-box approach based on a number of rules and heuristics to detect and explain inconsistencies in OWL ontologies. The rules and heuristics are used to detect a number of common error patterns. The disadvantage of this approach is that, because it is heuristic based, the cause of an unsatisfiability can not be determined in every case. Another black-box approach is proposed by [48]. The authors propose an approach to categorize the unsatisfiable classes in an ontology into two types: *root unsatisfiable classes* and *derived unsatisfiable classes*. A root unsatisfiable class is an unsatisfiable class in which a clash or contradiction found in the class definition (axioms) does not depend on the unsatisfiability of another class in the ontology. A derived unsatisfiable class, on the other hand, is an unsatisfiable class in which a clash or contradiction found in a class definition either directly (via explicit assertions) or indirectly (via inferences) depends on the unsatisfiability of another class. The classification of unsatisfiable classes into root and derived classes needs to help ontology engineers in pinpointing the cause of the inconsistency. The advantage of black-box techniques is that it doesn't require a specialized reasoner and doesn't put an additional overhead on the reasoning process. A disadvantage, however, is that black-box techniques don't reveal the exact axioms that are causing an inconsistency, but rather hints a direction to look at. As a consequence of this disadvantage, we consider black-box approaches less suited in the context of ontology evolution, as we aim to pinpoint the exact cause of an inconsistency and offer possible solutions to the ontology engineer to resolve inconsistencies.

As far as we are aware of, only one single glass-box approach has been proposed. The authors of [69] present an approach that modifies the internals of an OWL reasoner to be able to trace the axioms that are causing an inconsistency. The outcome of a reasoning process checking for unsatisfiable classes is, besides of course a set of unsatisfiable classes, also a set of axioms for each unsatisfiable class that form the cause the unsatisfiability. Removing one of the axioms of such a set resolves that particular unsatisfiable class, and therefore can be seen as a MUPS.

The approach regarding consistency checking & inconsistency resolving presented in this dissertation can be categorized as a glass-box approach, and is therefore similar to the previously discussed approach. The difference between the two approaches is that our approach brings the notion of root and derived unsatisfiable classes from black-box approaches into a glass-box approach. This allows us to further restrict the selection of axioms causing an inconsistency (see Chapter 6). Furthermore, we offer a set of rules to weaken the selected axioms causing an inconsistency in order to resolve the ontology inconsistency.

2.5.4 Distributed and Decentralized Environments

Oliver [66] introduced the term *ontology synchronization* and proposed an ontology synchronization approach called CONCORDIA. She defines synchronization as the periodic process by which developers update the local vocabulary to obtain the benefits of shared-vocabulary updates, while maintaining local changes that serve local needs. In other words, synchronization is the process of keeping a local version of an ontology up-to-date with a global evolving version. The proposed approach is based around a *log model* which is similar to an evolution log. The synchronization is performed by processing the log model containing changes that are made in the shared version during the time period between the last synchronization actions and the current moment. For each of the changes in the log, a list of action choices is defined. Klein [51] has adapted this approach to the OWL ontology language and integrated it in its ontology evolution framework.

The ontology evolution framework of [83] also deals with distributed ontology evolution management. They propose the use of *replications*. Ontologies that use and extend other ontologies don't directly depend on the original ontology, but rather depend on a replica of the original ontology. Whenever the original ontology changes, consistency is maintained as depending ontologies depend on an unchanged replica. To update a depending ontology, the changes to the original ontology needs to be propagated to the replica (similar to the synchronization process mentioned in the previous paragraph). They have adopted a pull-based approach for this purpose. However, their proposed solution seems to approach the problem of ontology evolution in a distributed environment from a centralized point of view. A characteristic that does not hold for an environment as the WWW (or the Semantic Web). Neither does their approach take into account that ontologies not willing to update can prevent other ontologies from updating.

In this dissertation, we take a different approach to the problem of ontology evolution in a distributed and decentralized environment. While normally an ontology just depends on another ontology, we suggest to let an ontology depend on another ontology *at a specific moment in time*. An ontology at a certain moment in time can always be recreated by means of the version log, even if it has changed in the meantime. Furthermore, we offer a solution to bypass blocked ontologies caused by the refusal to update of ontologies they depend on (see Chapter 7).

2.6 Summary

In this chapter, we gave an overview of relevant background and related work for the topic of Web-based ontology evolution. We discussed a definition of ontologies and provided an overview of existing Semantic Web ontology languages. We focused in particular on OWL DL as it forms the language

of choice of this dissertation. Furthermore, the syntax and semantics of $\mathcal{SHOIN}(\mathbf{D})$, the Description Logic variant OWL DL is based upon, were described, an overview of the different reasoning tasks was provided and the exact correspondence with OWL DL was discussed.

Moreover, we discussed the domains of temporal databases, database evolution & versioning, and ontology evolution & versioning as related domains. We especially focused on existing work concerning more specific aspects of the problem of ontology evolution. We discussed the aspects of change representation, change detection, consistency checking & inconsistency resolving and evolution in a distributed and decentralized environment. We compared existing work on these aspects with the approach we present in this dissertation.

Chapter 3

Ontology Evolution Framework

In the previous chapter, we discussed background and related work for this dissertation. We gave a definition of the term *ontology* that we will abide by in this dissertation. We discussed a variety of ontology languages, thereby focusing on the different Semantic Web languages. Dictated by the choice of OWL DL as the ontology language in this dissertation, we furthermore provided a short overview of Description Logics as it forms the foundation of this ontology language. In addition, we gave an overview of existing research in related domains as database evolution, maintenance of knowledge base systems, and the domain of ontology evolution itself. Finally, we focused on existing solutions to specific aspects of the ontology evolution problem relevant for this dissertation.

In this chapter, we give a general overview of the ontology evolution framework that we propose in this dissertation. Details of the framework have been published in [73], [74] and [75]. We focus on the specific characteristics of our framework, motivate the design decisions taken, and present an overview of the different phases of our framework.

The structure of this chapter is as follows. First we describe the main characteristics of the ontology evolution framework (see Section 3.1). In the following section, we give an overview of the framework itself and the purpose, structure and functioning of its different phases (see Section 3.2). Finally, Section 3.3 provides a summery of the chapter.

3.1 Characteristics

In general, an ontology on the Web doesn't exist in isolation. On the contrary, other ontologies, Web sites, applications, ... may depend on this ontology for their own particular purpose. E.g., web pages that contain meta-

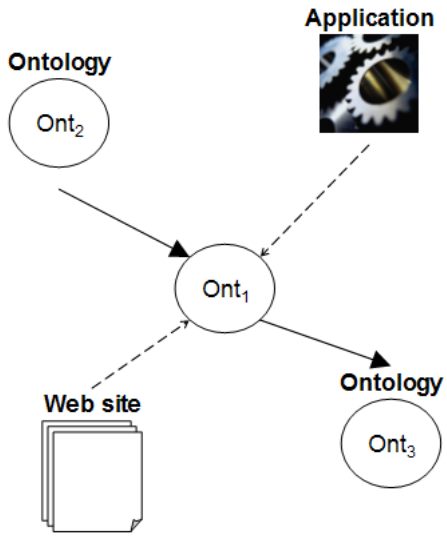

Figure 3.1: Example usage of an ontology on the Web

information about the creator of the page may depend on the Dublin Core[1] ontology for this purpose. The other way around, the ontology itself may again depend on other ontologies. We use the term *depending artifact* to refer to an ontology, Web site, application, etc. that depends on a certain ontology. Figure 3.1 shows a representative example of the use of an ontology Ont_1 on the Web. The ontology Ont_1 has a number of depending artifacts, e.g., an ontology Ont_2 that extends the concepts defined in Ont_1, an application that uses ontology Ont_1 as internal model, another example is a Web site where the content is annotated with instances of concepts of Ont_1. In its turn, Ont_1 itself is built on top of another ontology Ont_3.

As already mentioned in the introduction, the ontology evolution framework should (1) allow ontology engineers to request changes for the ontologies they manage, (2) ensure, when an ontology engineer decides to change an ontology, that the ontology evolves from one consistent state into another consistent state, (3) guarantee that the depending artifacts remain consistent with the evolved ontology, and (4) provide a detailed overview of the evolution of an ontology supporting different levels of granularity, views and interpretations.

The first requirement (requirement 1) means that the ontology evolution framework functions as a mediator between an ontology engineer and the ontology itself. The ontology engineer doesn't apply changes directly to an ontology itself, but rather formulates its request for changes to the ontology evolution framework. It is the responsibility of this framework to correctly handle the formulated changes requests.

[1] See `http://www.dublincore.org`

To satisfy the second requirement (requirement 2) the ontology evolution framework should be able to verify whether changes requested by an ontology engineer will maintain ontology consistency when applied. Changes to one part of the ontology may possibly introduce contradictions with other parts of the ontology. The framework therefore should check whether requested changes will introduce inconsistencies. Inconsistencies are in general resolved by changing other parts of the ontology to overcome the contradictions i.e., the change requested by the ontology engineer leads to a number of additional changes, called *deduced changes*. The framework should inform the ontology engineer about the consequences of his requested change (the possible inconsistencies it might introduce), and the possible solutions to resolve the inconsistencies.

Ensuring that depending artifacts remain consistent with the changed ontology (requirement 3) differs from consistency maintenance of a single ontology because propagation of changes no longer holds due to a number of reasons:

- A distinctive characteristic of the Web is its distributed and decentralized nature. This means that an ontology engineer doesn't necessarily has sufficient permissions to apply changes to depending artifacts. As a consequence, an approach of propagating changes to depending artifacts does no longer hold in this situation.

- Propagating changes to depending artifacts turns out to be even more problematic as we are usually unaware of all the depending artifacts of an ontology.

- Some depending artifacts may be in a format that makes them extremely hard or even impossible to change (e.g., applications in binary format). As a consequence, the approach of change propagation fails.

- Propagating changes to depending artifacts may even be unwanted because either the maintainer of a depending artifact doesn't want to update at that moment in time or doesn't want to update at all. Consider for example a Web site that uses ontology Ont_1 as source for the annotation of its content. It doesn't necessarily mean that whenever the ontology changes, the defined annotations or the content of the Web site have to change accordingly. Once more, the approach of change propagation fails, as it would forces users to update.

Taking these aforementioned reasons into account, the ontology evolution framework should offer alternative methods to guarantee consistency between ontologies and their depending artifacts.

Important to mention is that the decision whether to update a depending artifact or not rests with its maintainer. To be able to make a well-founded

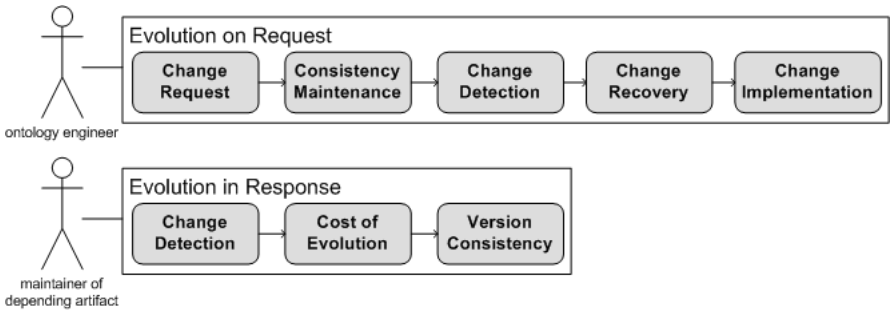

Figure 3.2: Overview of the ontology evolution framework

decision, the maintainers of depending artifacts should have a good *understanding of the changes* that occurred to the ontology (reguirement 4). Therefore, an important characteristic is that the framework should offer these maintainers a complete overview of the occurred changes with support for different levels of granularity (i.e., fine-grained changes vs. more high-level changes), different views, and different interpretations of changes (see Section 3.2.1 for more details). Furthermore, the framework should be able to determine whether the changed ontology is *backward compatible* with its previous version for a particular depending artifact i.e., can the old version of the ontology be replaced by the new version without breaking the depending artifact. Finally, when the maintainer of a depending artifact decides to update, the framework should inform him of the consequences i.e., the changes to the depending artifacts required to maintain consistency with the changed ontology.

3.2 Ontology Evolution Framework Overview

The ontology evolution framework consists of two parts corresponding to two different tasks. The first task handles the evolution process of an ontology as consequence of a change request by an ontology engineer. We use the term *evolution on request* to refer to this task. The second task handles the evolution process of a depending artifact as consequence of changes to an ontology it depends on. In this dissertation, we mainly focus on ontologies depending on other ontologies when considering depending artifacts. We refer to this second task with the term *evolution in response*. Figure 3.2 shows an overview of the different phases for both tasks of the ontology evolution process. The framework consists of the following phases: (1) Change Request, (2) Consistency Maintenance, (3) Change Detection, (4) Change Recovery, (5) Change Implementation, (6) Cost of Evolution, and (7) Version Consistency.

A fundamental prerequisite of any ontology evolution approach is the

ability to keep track of all the changes applied to the ontology. For this purpose, existing approaches construct an *evolution log*[2]. Such an evolution log lists all the changes ever applied to the ontology (e.g., `addClass(C)`, `setTransitivity(P)`, ...). We take a different approach. Our ontology evolution framework is based on the notion of a *version log*. In contrast to an evolution log, the version log stores for each concept ever defined in the ontology, the different versions it passes through during its life cycle: from its creation, over its modifications, until its retirement. In other words, the version log keeps track of the state of all concepts at the different moments in time. The use of a version log together with the formal definitions of the different changes that can be applied to an ontology in terms of a temporal logic based language, allows to detect occurrences of changes and makes it possible to automatically construct an evolution log listing all applied changes for a given ontology. Note that in our approach, maintainers of depending artifacts can define their own set of change definitions suiting their own particular needs, and are not restricted to the collection of change definitions used by the ontology engineer of the ontology they depend on. As a consequence, maintainers of depending artifacts can create their own evolution log for an ontology they depend on that complements the evolution log of the ontology itself. The version log, the temporal logic based language and the evolution log are described in detail in Chapter 4.

In the following two subsections, we give an overview of the different phases for both the *evolution on request* and the *evolution in response* task. Section 3.2.1 discusses the phases of the *evolution on request* task, Section 3.2.2 discusses the phases of the *evolution in response* task.

3.2.1 Evolution on Request

In this section, we discuss the various phases of the ontology evolution framework to fulfill the *evolution on request* task: (1) Change Request, (2) Consistency Maintenance, (3) Change Detection, (4) Change Recovery, and (5) Change Implementation. Before we give a detailed overview of the different phases, we first summarize each phase's goals below:

- **Change Request:** the Change Request phase allows ontology engineers to express their request for change in terms of primitive and complex changes.
- **Consistency Maintenance:** as the requested changes by the ontology engineer possibly turn the ontology into an inconsistent state, the goal of the Consistency Maintenance phase is to check whether consistency is maintained and to generate deduced changes to resolve inconsistencies when needed.

[2]In literature, the term *change log* is sometimes also used instead.

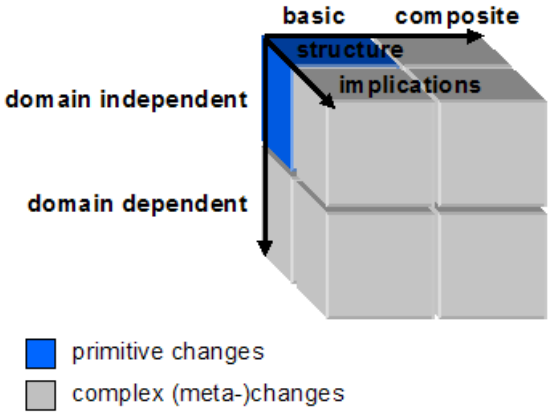

Figure 3.3: Different dimensions of change

- **Change Detection:** the purpose of the Change Detection phase is to detect (complex) changes from the version log that were not explicitly listed in the change request, in order to provide a better understanding of the evolution of an ontology.
- **Change Recovery:** when a sequence of changes is used instead of the intended complex change, unnecessary deduced changes may be added after each step in the sequence of changes causing an unintended loss of information. The goal of the Change Recovery phase is to recover the ontology from unnecessary deduced changes.
- **Change Implementation:** the purpose of the Change Implementation phase is to implement the requested and deduced changes into an actual ontology.

Change Request

This phase lets ontology engineers express their request for change in terms of predefined changes. E.g., to specify a request to add a subclass relation between the Classes A and B the ontology engineer can use the change `addSubClassOf(A, B)`. The framework supports different types of changes. They are defined along three dimensions (see Figure 3.3):

Basic vs. composite This dimension corresponds to differences in granularity. Basic changes are fine-grained changes that modify *exactly one* element of the ontology, and which cannot be decomposed any further into simpler changes (e.g., `addClass(A)` adds a new class with ID A to the ontology). Composite changes are more coarse-grained changes that modify *more than one* element of the ontology. Composite changes offer a higher level of abstraction and reflect better the intention of the ontology engineer (e.g., `deleteMutualDisjointness(C,`

`D)` removes the mutual disjointness that exists between the Classes C and D).

Domain independent vs. domain dependent As the terminology suggests, domain independent changes are changes independent of the domain described by the ontology. These domain independent changes are expressed as modifications to the constructs of the ontology language (e.g., `addClass(A)`). Domain dependent changes are changes defined for a particular domain (e.g., `addNewHivVariant(HIV1-Type-O)` adds a new variant of the HIV virus to a medical ontology). The advantage of domain dependent changes is that they don't require an in-depth-knowledge of the ontology language used, and closer represents changes of the real world. A disadvantage is that domain dependent changes are only applicable for one particular domain.

Changes vs. meta-changes The last dimension corresponds to the difference between respectively changes and meta-changes. Changes specify *what* has to change to the structure of the ontology, while meta-changes rather specify implications of a change i.e. information about a change. For example, a meta-change may specify that the constraints on a Property are *weakened* when the domain of that property is replaced by a superclass of the original domain.

In the remainder of this dissertation, we use the term *primitive changes* to refer to basic, domain independent changes. The set of primitive changes is exhaustive as it is derived from the underlying ontology language. Furthermore, we use the term *complex changes* to refer to both domain dependent (meta-)changes and composite (meta-)changes. The set of complex changes is infinite as new complex changes can always be defined [54]. The complete set of primitive changes and a number of examples of complex changes for OWL DL, together with their formal definition are presented in Chapter 5.

To request a change to an ontology, an ontology engineer specifies a change request in terms of pre-defined changes. The result is a set of *requested changes*. A change will always result in some new concept versions in the version log, representing the new state of the concepts changed. These concept versions are marked as 'pending' and are called *pending versions*. The status 'pending' is needed because it is not yet checked whether the new states maintains the consistency of the ontology. The changes are also not yet implemented in the actual ontology. Note that meta-changes are not useful for requesting changes as they specify implications of a change instead of specifying what has to change. However, they play an important role in understanding the evolution of an ontology, as will be shown in Section 3.2.1.

Consistency Maintenance

As modifications to an ontology risk turning the ontology into in inconsistent state, it should be verified whether the ontology remains consistent after the changes are applied. When the applied changes result into an inconsistent ontology, possible solutions should be suggested to the ontology engineer to avoid the inconsistency. It is then the responsibility of the ontology engineer to select a satisfactory solution. The consistency maintenance phase consists of three tasks: *consistency checking*, *inconsistency resolving*, and *backward compatibility checking*.

To check whether an ontology remains consistent after applying the requested changes, the pending versions in the associated version log are first transformed into an actual ontology copy. To check for ontology consistency, we can use one of the existing reasoners (e.g., Racer [62], Fact [37], Pellet [81], ...). While such reasoners allow detecting inconsistencies, determining *why* the ontology is inconsistent and *how* to resolve these inconsistencies is far from trivial. The problem with current reasoners is that they provide very little information about which concept definitions (or axioms in terms of DL) of the ontology are causing the inconsistency. In this framework, we introduce an approach that determines the axioms (or parts of axioms) causing an inconsistency. We do this by extending the tableau algorithm [4], on which most state-of-the-art reasoners are built, so that it explicitly keeps track of the internal handling of the axioms by the algorithm.

An inconsistent ontology is the result of axioms that are too restrictive as a whole, and thereby introduce contradictions. To resolve an inconsistency, the framework offers a set of rules that can be used by the ontology engineer to resolve the inconsistency detected. These rules are used by the ontology engineer to eliminate contradictions by weakening one or more conflicting axioms, and to guide the ontology engineer to a solution. The selection of applicable rules in a particular situation is based on the selection of axioms extracted from an adapted reasoner. Note that it is still the task of the ontology engineer to select the axioms he wants to change and the rule he wants to apply in the case that more than one rule is applicable. A rule either calls another rule or adds a new change to the change request. We call such a change a *deduced change*. Note that the process of consistency maintenance is an iterative process as new deduced changes may possibly introduce new inconsistencies. At worst, the consistency maintenance phase results in a cyclic process where changes continuously undo each other. Such cycles can be recognized using the version log and should be cut short by the framework.

Besides detecting inconsistencies in the ontology and offering suggestions to the ontology engineer to resolve these inconsistencies, it is equally important that the consequences for the depending artifacts, managed by the same ontology engineer, are made clear. Ontologies are built for a certain purpose

and often serve as backbone for an application, Website or other depending artifact. When both the ontology and the depending artifact are controlled by the same individual or group, it makes sense to inform the ontology engineer whether a changed ontology still fulfills the needs of the depending artifacts. Therefore, it is important to know whether the changed ontology is still backward compatible with its predecessor for a given depending artifact i.e. can the old version of the ontology be replaced by the new version without causing problems for the depending artifact.

In Section 6.4, we discuss how to specify compatibility requirements. A new version of an ontology should fulfill these compatibility requirements in order to be considered backward compatible with its previous version for a given depending artifact. The framework is able to verify whether an ontology fulfills these compatibility requirements. This information (i.e. backward compatible or not) can be used to conduct the ontology evolution process. As inconsistencies in an ontology can be resolved in more than one way, indicating which of the solutions breaks or maintains backward compatibility is of great benefit in making decisions.

As a result of this phase, the ontology evolves from one consistent state to another. Possible inconsistencies introduced by the requested changes (see previous section) are resolved by the ontology engineer by selecting the appropriate corrective rules proposed by the ontology evolution framework. The applied rules add a number of deduced changes to the change request. When the ontology engineer agrees with the complete change request (including both requested and deduced changes), all pending versions in the version log are changed to *confirmed versions*, indicating that they are confirmed for implementation. Alternatively, the ontology engineer may reject any proposed solution, and subsequently roll-back the requested changes which caused the inconsistency in the first place. A detailed discussion of the consistency maintenance process is presented in Chapter 6.

Change Detection

The purpose of the change detection phase for the *evolution on request* task is to detect (complex) changes and meta-changes that occur as a consequence of modifications to the ontology, but that were not explicitly specified in a change request. While other approaches exclusively base their evolution log on the changes specified in the change requests, we extend the evolution log with changes detected by our ontology evolution framework. This means the the evolution log in our approach is constructed from requested changes, deduced changes and detected changes.

There are a number of reasons why we believe the detection of changes should be an intrinsic part of ontology evolution framework for the *evolution on request* task:

- **Richer evolution log.** The purpose of the change detection phase

is to result into a semantically richer evolution log then is achievable when only changes listed in change requests are used to populate an evolution log. A semantically richer evolution log must provide a better understanding of the evolution of an ontology. Note that offering complex changes to an ontology engineer certainly doesn't guarantee that they will be used at all times when appropriate. Due to the complexity of the matter involved, it is not always trivial for an ontology engineer to select the intended complex change he wants to apply. Instead, he may rely on primitive changes to achieve the desired result step by step, evaluating the progress after each step. In the end, he might unwittingly have applied a complex change. In existing approaches, the (un-)intended complex change will not get listed in the evolution log. The change detection phase overcomes this flaw.

- **Reduce importance of tool support.** Related to the previous item, the change detection phase reduces the importance of tool support. When the change log is only populated with changes specified in change requests, the resulting change log depends very much on the tool used to author the ontology. Some tools may offer the ontology engineer a comprehensive set of complex changes, while with other tools this set is rather limited. The change detection mechanism is able to smoothen out differences in tool support.
- **Reduce loss of information.** As mentioned in the first item, an ontology engineer sometimes uses a sequence of changes, although a complex change that corresponds to the actual intended change is defined. The drawback of using a sequence of changes instead of the intended complex change, is that it possibly goes together with unnecessary information loss. Information loss can occur because the framework will check for consistency after each step in the sequence of changes, possibly adding deduced changes to resolve the detected inconsistencies. When considering the sequence of changes as a whole, some of the deduced changes may turn out to be superfluous.

As will be shown in Section 4.3, we define changes in terms of a Change Definition Language, which is a temporal logic based language. Change detection is consequently an evaluation of change definitions defined in the Change Definition Language as temporal queries on a version log. The details concerning change detection are discussed in Section 5.3. The output of this phase is an elaborated evolution log containing requested changes, deduced changes as well as detected changes by the framework.

Change Recovery

As already mentioned in the previous section, the use of a sequence of changes instead of the intended complex change may by a cause for un-

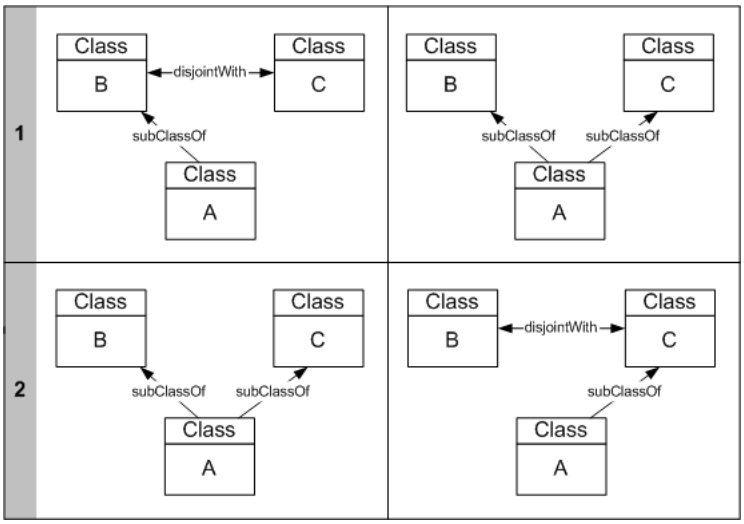

Figure 3.4: Example of change recovery

necessary loss of information. Consider as illustration the example shown in Figure 3.4. The ontology engineer wants to modify the subclass property for A from B to C. Although the change could for example be realized using a complex change `changeSubClassOf(A, B, C)`, the ontology engineer decides to use a sequence of changes instead. In step 1, the ontology engineer requests to add a new *subClassOf* Property between Class A and Class C using `addSubClassOf(A, C)` as requested change. Applying the requested change to the ontology would bring the ontology into an inconsistent state as the disjointness between B and C no longer applies. A possible solution to resolve the inconsistency is to delete the disjointness between B and C, i.e., the deduced change `deleteDisjointWith(B, C)` is added to the change request. In step 2, the ontology engineer deletes the *subClassOf* Property between A and B by requesting the `deleteSubClassOf(A, B)` change. As described in Section 3.2.1, the framework is able to detect complex changes. This detection mechanism will reveal that the sequence of changes corresponds to the complex change `changeSubClassOf(A, B, C)`. The question arises if the deduced change, that was added after the intermediary change to avoid ontology inconsistency, is still necessary when considering the sequence as a whole. In other words, the framework should offer the possibility to recover from unnecessary applied deduced changes. In our example, the framework would suggest to restore the disjointness between the Classes B and C.

To recover from unnecessary deduced changes, the framework looks up those versions in the version log that are the result of deduced changes that were added after the intermediary change requests of the detected sequence. It undoes these versions in the version log and verifies if the ontology remains

consistent without these versions. If the ontology remains consistent, the framework suggests the ontology engineer to remove these versions permanently, otherwise, these versions are restored and the deduced changes are not recovered. Note that it is the decision of the ontology engineer whether to recover changes or not.

Change Implementation

Until now, the requested changes and possible deduced changes, have been solely applied to a local copy of the ontology. It is the purpose of this phase to implement these changes to the public version of the ontology. This is straightforward as it only consists of replacing the original, public ontology with the evolved, local copy of the ontology. In contrast with ontology versioning where the different versions of an ontology coexists next to each other (each of them with its own namespace), we each time overwrite the previous public version with the latest version. Previous versions remain accessible through the version log, as will be explained in Section 3.2.2.

Note that this phase does not necessarily need to be performed at the end of each update cycle of the ontology. In general, changes to the ontology are not immediately made publicly available. Ontology engineers first further develop a local copy of an ontology (by means of the previous phases), before they decide to release a new version of an ontology to the public.

3.2.2 Evolution in Response

In this section, we discuss the various phases of the ontology evolution framework to fulfill the *evolution in response* task: (1) Change Detection, (2) Cost of Evolution, and (3) Version Consistency. Before we give a detailed overview of the different phases, we first summarize each phase's goals below:

- **Change Detection:** the change detection phase for the *evolution in response* task allows maintainers of depending artifacts to create an individual interpretation of the changes occurred using an own set of (complex) change definitions independent of the ontology they rely on.
- **Cost of Evolution:** an important factor that helps maintainers of depending artifacts to decide whether or not to update a depending artifact to a new version of an ontology it depends on, is the cost to update. The purpose of this phase is to reveal the inconsistencies that an update would introduce and to indicate the latest backward compatible version.
- **Version Consistency:** the purpose of the Version Consistency phase is to keep a depending artifact consistent with a changed ontology it depends on, regardless whether or not the maintainer of the depending artifact decides to update.

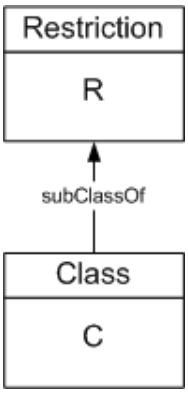

Figure 3.5: Example ontology illustrating a situation where two changes may result in the same modification

Change Detection

The evolution process for the *evolution in response* task starts with the change detection phase. It allows maintainers of depending artifacts to gain a better understanding of the evolution of the ontology they depend on. For this purpose, these maintainers don't have to rely on the evolution log associated with the ontology under consideration, but can define, if wanted, their own set of complex changes. The outcome of this change detection phase, is an evolution log at the side of a maintainer of a depending artifact, describing the changes of an ontology they depend on. This evolution log may differ from the evolution log created for the *evolution on request* task, as maintainers of depending artifacts may choose to use different change definitions. The evolution log published together with the ontology and the evolution log resulting from this phase offer maintainers of depending artifacts a detailed overview of the ontology evolution. Based on this information, maintainers of depending artifacts should decide whether they still agree with the changed ontology and whether they find the changes to the ontology sufficient to consider an update to the latest version of the changed ontology they depend on.

The reasons to include a change detection phase for the *evolution in response* task are as follows:

- **Different changes may give the same result.** Different complex changes may in practice result in the same ontology modification. Consider as example the changes `deleteSubClassOf(C, R)` and `removeRestriction(C, R)`. The former simply deletes the *subClassOf* Property between C and R, the latter removes the restriction R from the Class C. Note that the second change is more fine-grained defined change in comparison to the first change as it requires R to be a Restriction and not just a Class. Applying both changes to the situation

shown in Figure 3.5 would result in the same modification. Assume that the ontology engineer uses `deleteSubClassOf(C, R)` to requests his change. This would mean that, without a change detection phase for the *evolution in response* task, it would be more troublesome for maintainers of depending artifacts to reveal whether or not a restriction was removed.

- **Meta-changes.** Meta-changes play a valuable role in the understanding of an ontology evolution as they define the implications of a change. As already mentioned, meta-changes are not suited to specify a request for change as they don't define what has to change. Moreover, ontology engineers don't want to be burden with the task of manually adding meta-changes to the evolution log. The change detection phase makes it possible to automatically detect meta-changes, thereby improving the understanding of an ontology evolution for maintainers of depending artifacts.

- **Infinite number of complex changes.** The number of complex changes that can be defined is infinite. Nevertheless, ontology engineers only use a finite number of these complex change definitions. An approach that only populates the evolution log with changes listed in change requests, obliges maintainers of depending artifacts to restrict themselves to the same set of complex change definitions as used by the ontology engineer of the ontology they depend on. Nevertheless, other complex change definitions may be more appropriate for them to understand the evolution of the ontology. As a consequence, ontology engineers and maintainers of depending artifacts are able to define and use different sets of complex change definitions.

- **Difference in interpretation.** As discussed in the definition of an ontology (see Section 2.1), an ontology is a 'shared conceptualization', meaning that the different stakeholders agree on its representation of the world. This doesn't imply that the different stakeholders also have to agree on the interpretations of the ontology changes. Ontology engineer and maintainers of depending artifacts may adhere to different interpretations for the same ontology modification.

Note that the change detection process itself is not different for the *evolution on request* task then it is for the *evolution in response* task. The point in having two change detection phases is that it allows ontology engineers and maintainers of depending artifacts to use a different set of change definitions and to have different interpretations for the same modifications. This last one turns out to be very useful when considering depending artifacts depending on multiple ontologies. Consider as example a depending artifact D that depends on two ontologies O_1 and O_2. When we assume that the

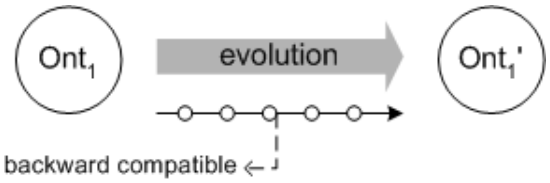

Figure 3.6: Cost of evolution

ontology engineers of both ontologies have different interpretations for the same ontology modifications, it would become very difficult for maintainers of depending artifacts, without the change detection phase, to understand the evolution of both ontologies.

Cost of Evolution

As is shown in Figure 3.1, depending artifacts exist in a variety of forms: other ontologies, annotations of Web sites, applications, ... As the topic of this dissertation is 'ontology evolution', we mainly focus on ontologies depending on other ontologies when considering depending artifacts.

Besides understanding the ontology changes that have occurred, another key element in the decision whether to update a depending artifact or not is the *cost* of updating. Figure 3.6 schematizes the *cost of evolution* phase. In this phase, the framework checks to which (intermediate) version[3] the ontology remains backward compatible for the given depending artifact. In other words, to which (intermediate) version can one update without any cost i.e. without requiring changes to the depending artifact. Another task of the framework is to determine what the consequences are for the depending artifact when updating to a version that is not backward compatible for the depending artifact i.e. which parts of the depending artifact need to change during the update. Both tasks boil down to the consistency checking and backward compatibility checking task as described in Section 3.2.1.

The outcome of both tasks will probably influence the decision whether or not to update to a great extent. If there exists an (intermediate) version of the ontology that is backward compatible and comprehends all the changes a maintainer of a depending artifact wishes for, this maintainer may decide to update the depending artifact only to the last backward compatible (intermediate) version at no cost. In other situations, the cost to update may be considered too high so that the maintainer may abandon the plans to update altogether.

[3]We use the term 'intermediate version' to refer to one of the versions in the version log that together have lead to a publicly available version (e.g., Ont'_1 in Figure 3.6), but that never has been published as a public version on its own. An intermediate version is rather a version in-between towards a public version.

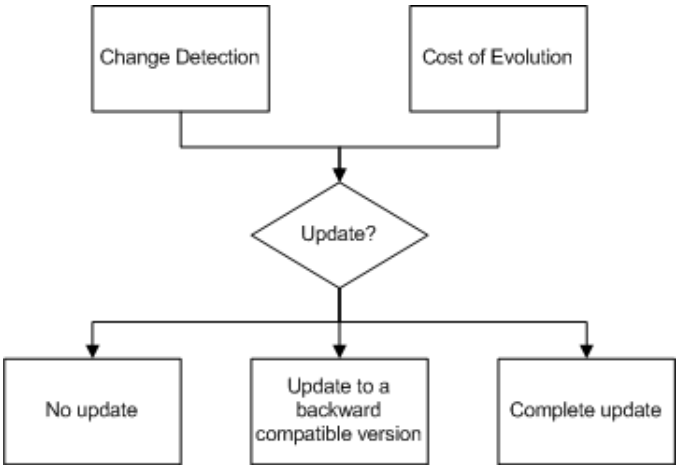

Figure 3.7: Understanding of changes and cost of evolution influence the decision whether to update or not

Version Consistency

As mentioned earlier, several reasons may withhold maintainers of depending artifacts from updating to the latest version: the maintainer doesn't agree with the changes applied (i.e., a shared conceptualization is no longer possible), the changes applied don't offer any additional value for the maintainer, all interesting changes are available in a backward compatible intermediate version (i.e., the need to update completely is absent), the cost of updating completely is considered too high, organizational causes (e.g., updates are only planned every six months), etc.

Figure 3.7 illustrates the input that the framework provides for the decision whether to update completely, partially or not at all. The change detection phase allows maintainers to gain understanding of the changes that have occurred, while the cost of evolution phase informs the maintainer of which (intermediate) version is still backward compatible (i.e., updating to this version can be done at no cost) and which parts of the depending artifact are candidate to changes when updating completely.

As maintainers of depending artifacts are not obliged to update completely, the framework should support the retrieval of past versions of the ontology. As the version log stores all past states of the different ontology concepts ever defined in a particular ontology, any past version (including intermediate versions) can be reconstructed. In the case of updating to the latest version (i.e., a complete update), we only consider ontologies as depending artifacts in this dissertation, excluding other depending artifacts (e.g., applications). To update a depending ontology so that it remains consistent with the latest version of the ontology it depends on, a new iteration of the ontology evolution framework for the depending ontology is initiated.

The change request will be populated by deduced changes necessary to restore consistency with the ontology it depends on (see Section 3.2.1).

3.3 Summary

In this chapter, we informally introduced the ontology evolution framework that we propose in this dissertation. The ontology evolution framework allows ontology engineers to request changes for the ontologies they manage, guarantees that an ontology always evolves from one consistent state into another consistent state while maintaining the consistency of depending artifacts with the evolved ontology, and provides a detailed overview of the evolution of an ontology supporting different levels of granularity, views and interpretations.

The proposed ontology evolution framework consists of two major parts corresponding to two different tasks: the evolution on request task and the evolution in response task. The former handles the evolution process of an ontology as consequence of a change request by an ontology engineer, while the latter handles the evolution process of a depending artifact as consequence of changes to an ontology it depends on. Each task consists of a number of phases, each phase with its own particular objective.

The evolution of an ontology is in our approach represented by means of a version log. A version log stores for each concept ever defined in the ontology, the different versions it passes through during its life cycle. Note that the version log doesn't represent the evolution in terms of changes, but rather keeps track of the different states of each concept over time. Whenever an ontology engineer wants to modify an ontology, he specifies a change request in terms of changes that he wants to apply (i.e., requested changes). In our approach, changes are formally defined in terms of a temporal logic based language, called the Change Definition Language. To avoid that the changes requested by an ontology engineer turn the ontology into an inconsistent state, the framework may add extra changes, called deduced changes, to the change request in order to restore consistency.

To allow for a better understanding of the evolution of an ontology, the framework is able to create an evolution log. Such an evolution log is, in contrast with a version log, a particular interpretation of the evolution of an ontology in terms of change definitions. The framework is able to automatically detect changes that satisfy the change definitions by evaluating change definitions as temporal queries on a version log. We use the term detected changes to refer to the occurrence of changes that have been detected by the framework. Due to the clear separation between the representation of an evolution (by means of a version log) and the interpretation of an evolution (by means of an evolution log), it becomes possible to associate different interpretations (i.e. evolution logs) with the same evolution.

Chapter 4

Foundations

In the previous chapter, we gave an informal overview of the ontology evolution framework that forms the subject of this dissertation. The overview merely gave a general overview of the different phases of the framework and the role they fulfill, without going into much detail for each phase. In this chapter, we discuss the foundations on which the ontology evolution framework and its individual phases are based. In Section 4.1, we discuss the version log that keeps track of the different versions that the concepts of an ontology pass through during their lifetime. In Section 4.2, we introduce a temporal logic as a temporal extension of the $\mathcal{SHOIN}(\mathbf{D})$ description logic. This temporal logic forms the basis for our Change Definition Language as we will discuss in Section 4.3. The Change Definition Language allows ontology engineers to formally define the changes they are interested in. In Section 4.4, we give an overview of the evolution log. We conclude this chapter with a short summary in Section 4.5.

4.1 Version Log

Our ontology evolution approach associates with each ontology it manages a version log. The purpose of such a version log is to keep track of the different versions an ontology concept passes through during its lifetime: starting from the creation of the concept, over its modifications until its eventual retirement. In this section, we first discuss the general approach we took regarding the version log (see Section 4.1.1), followed by the specification of a formal model of the version log (see Section 4.1.2).

4.1.1 General Approach

To describe the evolution of an ontology, several approaches can be taken. One possible approach is to describe the evolution of an ontology directly in terms of a sequence of changes that were applied to the ontology. This

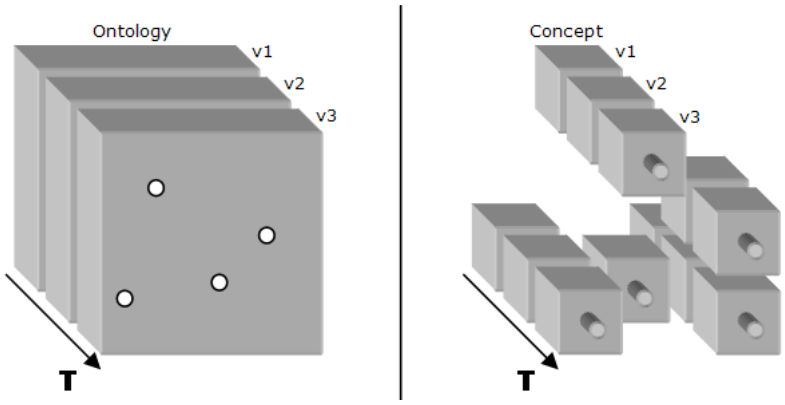

Figure 4.1: Different forms of the snapshot approach

approach is followed by most current ontology evolution approaches. We take a different approach as we describe an ontology evolution in terms of versions of ontology concepts, rather then storing a sequence of changes. In other words, the evolution of an ontology is captured by preserving its history. This approach resembles the approach generally taken by temporal databases [67]. However, in the case of temporal databases only the history of the instance data is stored, while the database schema is assumed to remain static. As motivated in Section 3.2.1 and 3.2.2, our approach has a number of advantages compared to the first one.

Preserving the history of an ontology can be achieved in different manners. First of all, a *timestamp* approach or *snapshot* approach can be taken. Although both approaches are equally expressive, their manner of representation differs. The former consists of labeling the ontology elements subject to change with a timestamp. The latter is based on taking snapshots to capture the different states of an ontology over time. The history of an ontology is then described as a sequence of snapshots. Snapshots can be taken either of the ontology as a whole, where each snapshot contains all facts valid at a given moment in time, or rather of single concept definitions, in which case a sequence of snapshots describes the evolution of a single ontology concept. Figure 4.1 illustrates the difference between the two forms of the snapshot approach. The first form is considered highly inefficient as each snapshot stores the complete ontology. Moreover, capturing the evolution of the individual concepts is required if one wants to pose sensible queries to the version log. This is not feasible in the first form as it doesn't capture relations between successive versions of ontology concepts. For the version log, we therefore have adopted a snapshot approach that keeps track of the evolution of each individual ontology concept, instead of the evolution of the ontology as a whole.

In temporal databases, two dimensions of time are in general considered:

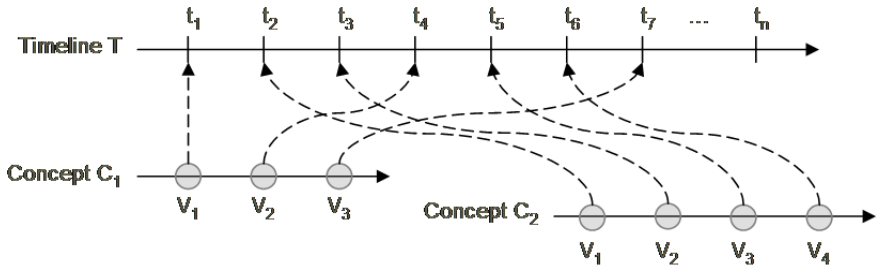

Figure 4.2: Schematized representation of a version log

transaction time and *valid time* [82]. Transaction time represents the time when data is actually stored in the database, while valid time represents the time when data is valid in the modeled world. The same two dimensions can also be applied to ontologies. With regard to the version log, transaction time would reflect the time when a new version of a concept is created, while valid time would represent the moment in time when the concept is valid in the described domain. We only keep track of the transaction time in the version log as we are merely interested in the moment in time when changes are applied to the ontology.

Furthermore, the representation of the timeline can be based on *time points* as well as *time intervals* [1]. As we record in the version log the moments in time an ontology concept changes, we adopt the point-based approach. We consider time as a discrete and linearly ordered timeline (as is also done in virtually all temporal databases). The timeline is furthermore *single-level*, meaning that no different granularities exist to refer to time points.

Figure 4.2 shows a schematized representation of the version log. A version log is associated with one particular ontology. For each concept (Class, Property, or Individual) defined in the associated ontology, a sequence of snapshots (or versions) of this concept is kept describing its history (see V_1, ..., V_n). A version of a concept contains the definition of that particular concept at a given moment in time. Each version is linked to a time point of the version log timeline **T** (see t_1, ..., t_n), and represents the transaction time. The transaction time of a concept version indicates the start of a version. We also explicitly store the end point of a version. Furthermore, each version has a state tag (*pending*, *confirmed*, *implemented*) indicating the state of the version. Although the states were already mentioned in Section 3.2), we briefly summarize the meaning of the three different states below:

- *Pending:* a version is tagged as 'pending' to indicate that the version is the result of a change request that hasn't yet been checked on consistency nor has it been implemented;

- *Confirmed:* a version is tagged as 'confirmed' if the version has passed the consistency check and is confirmed for implementation (but hasn't been implemented yet in the public version of the ontology);

- *Implemented:* a version is tagged as 'implemented' if the version has been implemented in the public version of the ontology;

The relation between requested changes (i.e., changes that were explicitly requested by an ontology engineer) and deduced changes (i.e., changes that were added as a result of the consistency checking process) in the change request is also reflected in the version log. A version that is the result of a requested change r may contain references to other versions that were the result of the deduced changes of r in the change request. We call these versions *deduced versions*.

Furthermore, every version also keeps track of the ID of the concept at that specific moment in time. The ID conforms to the `rdf:ID` attribute used in OWL. For OWL concepts without an `rdf:ID` attribute, the ID in the version log is left blank. In the next subsection, we present a formal model for the version log.

4.1.2 Formal Model

In this section, we present a formal model for the version log. Instead of using the native OWL syntax (i.e., the RDF/XML syntax), we rely on the syntax of the $\mathcal{SHOIN}(\mathbf{D})$ description logic variant to represent concept definitions. This means that the definition of an ontology concept consists of a set of DL axioms.

We first define the set of all concept names (and more specifically all Class names, Property names and Individual names) that are used in concept definitions during the complete lifetime of the associated ontology of a version log.

Definition 4.1 (Concept Names). *Assume the set $\boldsymbol{S}$ to be the set of all possible strings. The finite set of all Class names used $\boldsymbol{CN}$ is defined as $\boldsymbol{CN} \subseteq \boldsymbol{S}$, the finite set of all Property names used $\boldsymbol{PN}$ as $\boldsymbol{PN} \subseteq \boldsymbol{S}$, and the finite set of all Individual names used $\boldsymbol{IN}$ as $\boldsymbol{IN} \subseteq \boldsymbol{S}$. The set of all concept names $\boldsymbol{N}$ is then defined as $\boldsymbol{N} = \boldsymbol{CN} \cup \boldsymbol{PN} \cup \boldsymbol{IN}$.*

As already explained, for a version log, we consider time to be a discrete, linearly ordered timeline. We define the timeline of a version log as follows:

Definition 4.2 (Timeline). *A discrete, linearly ordered timeline $\boldsymbol{T}$ of a version log is defined as the finite set $\boldsymbol{T} \subseteq \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers. The time point 'now' $\in \boldsymbol{T}$, also referred to as current time, is defined so that $\forall t \in \boldsymbol{T}.(t \leq now)$.*

The version log keeps track of the history of each Class, Property or Individual that is defined in the associated ontology by storing their successive versions. A version of a concept contains the definition of that concept that holds at a given moment in time. In DL, the definition of a concept is formed by a set of axioms. Before introducing the notion of a *concept version*, we first define when we consider an axiom ϕ as part of the definition of a given concept name. We distinguish between Class, Property and Individual definitions. In order to classify axioms as Class, Property or Individual axioms, we require axioms to be in *normal form*. We therefore first define when we consider an axiom to be in normal form. These normalization and simplification functions are an adaptation of the normalization and simplification functions presented in [4].

Definition 4.3 (Concept Normalization and Simplification).

$$Norm(A) = A \tag{4.1}$$

$$Norm(\neg A) = \begin{cases} \bot & \text{if } A = \top \\ \top & \text{if } A = \bot \\ \neg A & \text{otherwise} \end{cases} \tag{4.2}$$

$$Norm(\neg\neg C) = Norm(C) \tag{4.3}$$

$$Norm(C_1 \sqcap \ldots \sqcap C_n) = Norm(C_1) \sqcap \ldots \sqcap Norm(C_n) \tag{4.4}$$

$$Norm(\neg(C_1 \sqcap \ldots \sqcap C_n)) = Norm(\neg C_1) \sqcup \ldots \sqcup Norm(\neg C_n) \tag{4.5}$$

$$Norm(C_1 \sqcup \ldots \sqcup C_n) = Norm(C_1) \sqcup \ldots \sqcup Norm(C_n)) \tag{4.6}$$

$$Norm(\neg(C_1 \sqcup \ldots \sqcup C_n)) = Norm(\neg C_1) \sqcap \ldots \sqcap Norm(\neg C_n)) \tag{4.7}$$

$$Norm(\forall R.C) = Simp(\forall R.Norm(C)) \tag{4.8}$$

$$Norm(\neg\forall R.C) = \exists R.Norm(\neg C) \tag{4.9}$$

$$Norm(\exists R.C) = \exists R.Norm(C) \tag{4.10}$$

$$Norm(\neg\exists R.C) = Simp(\forall R.Norm(\neg C)) \tag{4.11}$$

$$Norm(\neg(\leq nR)) = (\geq (n+1)\ R) \tag{4.12}$$

$$Norm(\neg(\geq nR)) = (\leq (n-1)\ R) \tag{4.13}$$

$$Simp(\forall R.C) = \begin{cases} \forall R.C_1 \sqcap \ldots \sqcap \forall R.C_n \\ \qquad \text{if } C = C_1 \sqcap \ldots \sqcap C_n \\ \forall R.C \\ \qquad \text{otherwise} \end{cases} \tag{4.14}$$

Besides the normalization and simplification functions introduced above, we will also use the following equivalence to split an axiom into two ore more separated axioms:

Definition 4.4 (Axiom Normalization).

$$C \sqsubseteq D_1 \sqcap \ldots \sqcap D_n \Longleftrightarrow C \sqsubseteq D_1 \text{ and } \ldots \text{ and } C \sqsubseteq D_n \tag{4.15}$$

Now that we have defined the normalization form, we are able to define when we consider an axiom as part of the definition of a given concept name. We start with defining a Class definition. We define an axiom ϕ to be part of the definition of a Class, if the following definition holds:

Definition 4.5 (Class Definition). *An axiom ϕ is defined to be part of a Class definition for a Class with name $\sigma \in \boldsymbol{CN}$, notation ClassDefinition(ϕ, σ), iff ϕ is in normal form and ϕ is of one of the following forms:*

- $\sigma \sqsubseteq C$ *(σ is a subclass of C);*
- $\sigma \equiv C$ *(σ is equivalent to C) or $C \equiv \sigma$ (C is equivalent to σ);*
- $\sigma \sqsubseteq \neg C$ *(σ is disjoint with C);*

An axiom ϕ is part of the definition of a Property, if the following definition holds:

Definition 4.6 (Property Definition). *An axiom ϕ is defined to be part of a Property definition for a Property with name $\sigma \in \boldsymbol{PN}$, notation PropertyDefinition(ϕ, σ), iff ϕ is in normal form and ϕ is of one of the following forms:*

- $\sigma \sqsubseteq R$ *(σ is a subproperty of R);*
- $\sigma \equiv R$ *(σ is equivalent to R);*
- $\exists\sigma.\top \sqsubseteq C$ *(C is the domain of σ);*
- $\top \sqsubseteq \forall\sigma.C$ *(C is the range of σ);*
- $\top \sqsubseteq (\leq 1\ \sigma)$ *(σ is a functional Property);*
- $\top \sqsubseteq (\leq 1\ \sigma^{-})$ *(σ is an inverse functional Property);*
- $\sigma \equiv R^{-}$ *(σ is an inverse Property of R);*
- $Trans(\sigma)$ *(σ is a transitive Property);*
- $\sigma \equiv \sigma^{-}$ *(σ is a symmetric Property).*

An axiom ϕ is part of the definition of an Individual, if the following definition holds:

Definition 4.7 (Individual Definition). *An axiom ϕ is defined to be part of an Individual definition for an Individual with name $\sigma \in \boldsymbol{IN}$, notation IndividualDefinition(ϕ, σ), iff ϕ is in normal form and ϕ is of one of the following forms:*

- $C(\sigma)$ *(Individual* σ *is an instantiation of a Class* C*);*
- $R(\sigma, o)$ *(Individual* σ *is the subject of a Property instantiation of a Property* R *with object* o*);*
- $\sigma = o$ *(Individual* σ *is the same as Individual* o*);*
- $\sigma \neq o$ *(Individual* σ *is different from Individual* o*).*

Now that we have defined when an axiom is part of the definition of a Class, Property and Individual, we are able to formulate the definition of a Concept:

Definition 4.8 (Concept Definition). *A concept definition for a Concept with name* $\sigma \in \boldsymbol{N}$ *is defined as a set* $\boldsymbol{A}$ *where* $\boldsymbol{A} = \{\phi \mid$ *ClassDefinition(*ϕ*,* σ*)* $\vee$ *PropertyDefinition(*ϕ*,* σ*)* $\vee$ *IndividualDe-finition(*ϕ*,* σ*)* $\}$*. If* $\boldsymbol{A}$ *is the concept definition of a concept with name* σ*, we note ConceptDefinition(*$\boldsymbol{A}$*,* σ*).*

As mentioned before, the version of a concept contains all the axioms that together form the definition of that particular concept at that particular moment in time. This moment in time indicates the transaction time (i.e., the start time) of the version. Besides the start time, we also keep track of the end time of the version. When a particular version hasn't ended yet, the end time is set equal to the *now* time point. Both the start and end time are elements from the version log timeline. Each concept version contains the concept name (might be the empty string), the set of axioms that together form its definition, and a set of deduced concept versions. Furthermore, a concept version contains the status of the version (pending, confirmed, or implemented). Its definition is given as follows:

Definition 4.9 (Concept Version). *A concept version* υ *for a given concept name* $\sigma \in \boldsymbol{N}$ *is a tuple* $\langle \sigma, \boldsymbol{A}, \boldsymbol{D}, s, t_s, t_e \rangle$ *where* σ *is the concept name,* $\boldsymbol{A}$ *is the set of axioms that together form the definition of* σ *i.e., ConceptDefinition(*$\boldsymbol{A}$*,* σ*),* $\boldsymbol{D} = \bigcup \upsilon_i$ *is a set of deduced concept versions,* $s \in \{$*'pending', 'confirmed', 'implemented'* $\}$ *is the status of the version, and* $t_s,\ t_e \in \boldsymbol{T}$ *are respectively the start- and end time of the version. The start- and end time denote a closed interval* $[t_s, t_e]$*.*

The transaction time of concept versions introduces a total ordering between concept versions. We therefore introduce the precedence relation $\prec_\upsilon$ for concept versions. The precedence relation expresses that one concept started before a second concept version started.

Definition 4.10 (Ordering of Concept Versions). *We say that* $\upsilon_1 \prec_\upsilon \upsilon_2$ *iff* $t_s \leq t'_s$ *where* $\upsilon_1 = \langle \sigma, \boldsymbol{A}, \boldsymbol{D}, s, t_s, t_e \rangle$ *and* $\upsilon_2 = \langle \sigma, \boldsymbol{A'}, \boldsymbol{D'}, s', t'_s, t'_e \rangle$*. The order relation* $\prec_\sigma$ *is a antisymmetric, transitive, and total relation.*

We can now define the complete history of an ontology concept as a sequence of concept versions for this concept. For this purpose, we introduce the notion of a concept evolution:

Definition 4.11 (Concept Evolution). *A concept evolution* $\boldsymbol{E}_\sigma$ *for a Concept with name* $\sigma \in \boldsymbol{N}$ *is a finite set of concept versions so that* $\forall \upsilon \in \boldsymbol{E}_\sigma.(\upsilon = \langle \sigma, \boldsymbol{A}, \boldsymbol{D}, s, t_s, t_e \rangle)$.

Before we formulate the definition of a version log, we first define a number of auxiliary definitions. The first definition states under which condition we consider a concept version υ to be the version for a given concept with name σ and given time point t.

Definition 4.12. *A concept version* υ *is the version for a given Concept with name* $\sigma \in \boldsymbol{N}$ *and given time point* $t \in \boldsymbol{T}$*, notation* $version(\upsilon, \sigma, t)$*, iff* $\upsilon = \langle \sigma, \boldsymbol{A}, \boldsymbol{D}, s, t_s, t_e \rangle$ *and* $t_s \leq t \leq t_e$ *holds.*

Based on the previous definition, we are able to define when we consider a concept version to be the previous version for a given concept and given time point. Furthermore, we also define when we consider a concept version to be the first version for a given concept name. Both definitions will be used in the interpretation of the temporal logic we define in Section 4.2.

Definition 4.13 (Previous Concept Version). *A concept version* υ *is defined as the previous concept version for a given Concept with name* σ *w.r.t. a given time point* $t \in \boldsymbol{T}$*, notation* $PreviousConceptVersion(\upsilon, \sigma, t)$*, iff* $\exists \upsilon_i \in \boldsymbol{E}_\sigma.(version(\upsilon_i, \sigma, t) \wedge (\upsilon \prec_\upsilon \upsilon_i) \wedge \neg \exists \upsilon_j \in \boldsymbol{E}_\sigma.(\upsilon \prec_\upsilon \upsilon_j \wedge \upsilon_j \prec_\upsilon \upsilon_i)))$.

Definition 4.14 (First Concept Version). *A concept version* υ *is defined as the first concept version for a given Concept with name* $\sigma \in N$*, notation* $FirstConceptVersion(\upsilon, \sigma)$*, iff* $\neg \exists \upsilon_i \in \boldsymbol{E}_\sigma.(\upsilon_i \prec_\upsilon \upsilon)$.

This leaves us with the definition of a version log. A version log simply contains a set of concept evolutions. For each concept ever created in the associated ontology, a concept evolution for this concept must exist in the version log. A version log is therefore defined as follows:

Definition 4.15 (Version Log). *We define a version log* Ω *as the tuple* $\langle \boldsymbol{O}, \boldsymbol{V} \rangle$ *where* $\boldsymbol{O}$ *is the associated ontology, so that*

$$\boldsymbol{O} = \{\langle \sigma, \boldsymbol{A} \rangle \mid \sigma \in \boldsymbol{N} \wedge ConceptDefinition(\boldsymbol{A}, \sigma)\}$$

and $\boldsymbol{V}$ *is the set of concept evolutions, so that*

$$\boldsymbol{V} = \bigcup_{\forall \sigma \in \boldsymbol{N}} \boldsymbol{E}_\sigma$$

We conclude this section with a last constraint. In a version log, for all time points of its timeline, there must exist a concept version that starts at that time point.

$$\forall t \leq now, \exists \upsilon.(\upsilon = \langle \sigma, \mathbf{A}, \mathbf{D}, s, t_s, t_e \rangle \ \wedge \ t = t_s)$$

4.1.3 Example

In this section, we illustrate the use of the version log by example. We take as example the ontology introduced in Figure 3.4 in Section 3.2.1 on Page 50. The ontology O before the appliance of the first change corresponds to the following set of axioms in $\mathcal{SHOIN}(\mathbf{D})$ syntax:

$$\mathbf{O} = \{\langle A, A \sqsubseteq B\rangle, \langle B, B \sqsubseteq \neg C\rangle\}$$

When we assume that the set of axioms as listed above is the initial state of the ontology, the associated version log looks as follows:

$$\Omega = \langle \mathbf{O}, \{\mathbf{E}_A \cup \mathbf{E}_B \cup \mathbf{E}_C\}\rangle$$

$$\begin{aligned}
\mathbf{E}_A &= \{\upsilon_{A,0} = \langle \text{`A'}, \{A \sqsubseteq B\}, \{\}, \text{`implemented'}, 0, now\rangle\} \\
\mathbf{E}_B &= \{\upsilon_{B,0} = \langle \text{`B'}, \{B \sqsubseteq \neg C\}, \{\}, \text{`implemented'}, 0, now\rangle\} \\
\mathbf{E}_C &= \{\upsilon_{C,0} = \langle \text{`C'}, \{\}, \{\}, \text{`implemented'}, 0, now\rangle\}
\end{aligned}$$

The version log Ω contains a reference to the associated ontology O and a set of concept evolutions i.e., one concept evolution for every concept in the associated ontology. Every concept evolution contains exactly one concept version i.e., the initial version of the concept. Every concept version stores the axioms that form the definition of that particular concept. Note that every concept versions starts at time point 0, and has the end time point set equal to *now* (where $now = 0$).

As already mentioned, when the ontology engineer formulates a change request to add a subclass property between Class A and Class C, applying this single change would turn the ontology into an inconsistent state. One possible solution to avoid the ontology from becoming inconsistent is to remove the disjointness between B and C ($B \sqsubseteq \neg C$) from the ontology. This solution leads to the addition of a new deduced change to the change request to remove the disjointness. After both changes are applied to the ontology, the version log looks as follows:

$$\Omega = \langle \mathbf{O}, \{\mathbf{E}_A \cup \mathbf{E}_B \cup \mathbf{E}_C\}\rangle$$

$$\begin{aligned}
\mathbf{E}_A &= \{\upsilon_{A,0} = \langle \text{`A'}, \{A \sqsubseteq B\}, \{\}, \text{`implemented'}, 0, 0\rangle, \\
&\qquad \upsilon_{A,1} = \langle \text{`A'}, \{A \sqsubseteq B, A \sqsubseteq C\}, \{\upsilon_{B,1}\}, \text{`implemented'}, 1, now\rangle\} \\
\mathbf{E}_B &= \{\upsilon_{B,0} = \langle \text{`B'}, \{B \sqsubseteq \neg C\}, \{\}, \text{`implemented'}, 0, 0\rangle, \\
&\qquad \upsilon_{B,1} = \langle \text{`B'}, \{\}, \{\}, \text{`implemented'}, 1, now\rangle\} \\
\mathbf{E}_C &= \{\upsilon_{C,0} = \langle \text{`C'}, \{\}, \{\}, \text{`implemented'}, 0, now\rangle\}
\end{aligned}$$

The changes listed in the change request are applied one after each other, and result in each into a new concept version. The *now* element is augmented after each requested change. The version log still contains the same evolution concepts as no new concept were added to the ontology. The end time points

of the evolution concepts $\mathbf{E}_A$ and $\mathbf{E}_B$ are closed, and new concept versions ($\upsilon_{A,1}$ and $\upsilon_{B,1}$) are added to their respective evolution concepts to represent the present situation. Note that the concept version $\upsilon_{A,1}$ keeps a reference to $\upsilon_{B,1}$ as deduced concept version i.e., it represents that the creation of concept version $\upsilon_{B,1}$ is caused by the creation of concept version $\upsilon_{A,1}$ in order to maintain consistency. Furthermore, note that the deduced concept version has the same start time as the concept version that caused the creation of the deduced concept version.

4.2 Temporal Logic

As the version log represents the history of an ontology and its concepts over time, it allows to define changes in terms of differences over time in the version log. In Section 4.3, we define a *Change Definition Language* that allows us to define changes w.r.t. the version log. In this section, we introduce the syntax and semantics of the temporal logic underpinning this Change Definition Language.

The term Temporal Logic has been broadly used to cover approaches to the representation of temporal information within a logical framework. Temporal logics differ from atemporal logics in the sense that statements have a truth-value that can vary over time. In general, two approaches to Temporal Logics are taken in literature: predicate-logic and modal-logic approaches. In the predicate-logic approach, the temporal dimension is captured by augmenting each proposition or predicate with an extra variable to be filled by an expression of time. Modal-logic approaches introduce modal-operators to gain the ability to model properties like belief, time-dependence, obligation, and so on. When applied to the temporal domain, modal-logic approaches introduce so called *tense operators* as modal-operators.

The logical language of tense logic contains, besides the usual logical operators, four basic tense operators $\mathcal{P}$ ('some time in the past'), $\mathcal{F}$ ('some time in the future'), $\mathcal{H}$ ('always in the past'), and $\mathcal{G}$ ('always in the future') where $\mathcal{P}$ and $\mathcal{F}$ are known as weak tense operators, and $\mathcal{H}$ and $\mathcal{G}$ are known as strong tense operators [76]. A number of common extensions to these basic tense operators were introduced including the binary $\mathcal{S}$ ('since') and $\mathcal{U}$ ('until') operator by [49]. Another common extension for discrete timelines is the next and previous time operator (respectively $\mathcal{N}$ and $\mathcal{R}$) to refer to respectively the immediately succeeding and preceding moment in time. In addition to the tense operators previously mentioned, we also introduce the tense operator $\mathcal{A}$ ('after') as a weak version of the $\mathcal{S}$ tense operator. Table 4.1 gives an overview of the informal semantics of these operators.

In this dissertation, we take as foundation for the Change Definition Language a *hybrid-logic approach* [2]. As the term already suggests, hybrid-logic approaches are midway between the modal-logic and the predicate-

Operator	Informal semantics
$\mathcal{P}\phi$	It has at some time in the past been the case that ϕ was true
$\mathcal{F}\phi$	It will at some time in the future be the case that ϕ is true
$\mathcal{H}\phi$	It has always been the case that ϕ was true
$\mathcal{G}\phi$	It will always be the case that ϕ is true
$\phi_1\ \mathcal{S}\ \phi_2$	ϕ_1 has always been true since the time when ϕ_2 was true
$\phi_1\ \mathcal{U}\ \phi_2$	ϕ_1 will always be true until a time when ϕ_2 is true
$\mathcal{N}\phi$	It will at the immediately succeeding time step be the case that ϕ is true
$\mathcal{R}\phi$	It has at the immediately preceding time step been the case that ϕ was true
$\phi_1\ \mathcal{A}\ \phi_2$	ϕ_1 has at some time been true since the time when ϕ_2 was true

Table 4.1: Overview of common tense operators

logic approaches. Hybrid-logic approaches supplement modal-logics with the ability to refer explicitly to specific states in formulas. This is achieved by formulas called *nominals*, which are true in exactly one state. In general, the @-operator is used to refer to a specific state. In the context of time, the @-operator is used to refer to specific time points, and is informally defined as follows:

$$\phi@t \text{ is true iff } \phi \text{ is true on time point } t$$

In the context of ontology evolution, we are interested in how we can formally define changes based on a temporal logic. These change definitions are formulated as an expression merely involving current and past versions of an ontology; future versions are not considered. We therefore restrict ourselves to those tense operators considering past times ($\mathcal{P}$, $\mathcal{H}$, $\mathcal{S}$, $\mathcal{A}$, and $\mathcal{R}$) in combination with the @-operator. As the tense operators considering future times are not useful for our purpose, we omit them from further discussion.

4.2.1 Parameterized and Non-parameterized Tense Operators

The version log offers two different views on the evolution of an ontology: a first one on the level of an ontology itself, a second one on the level of a single ontology concept. The former considers the complete set of concept versions listed in the version log and describes the complete history of the ontology, while the latter merely considers the concept versions that are part of a certain concept evolution and therefore only describes the evolution of

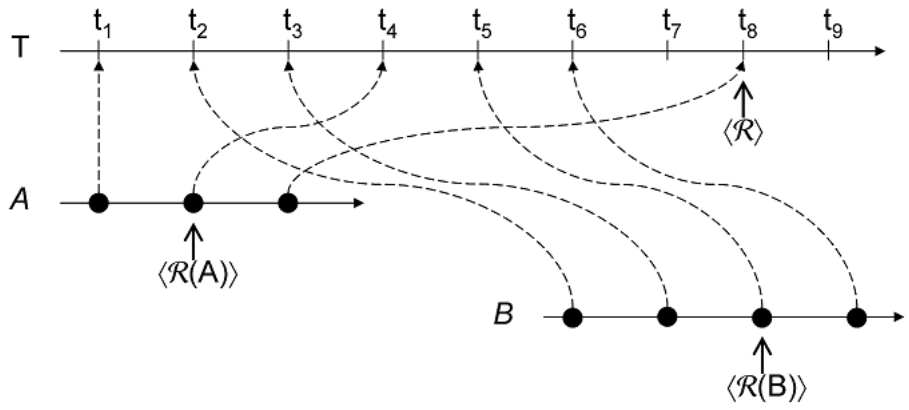

Figure 4.3: Example of a parameterized and non-parameterized version of the $\mathcal{R}$ tense operator

that particular concept. With respect to the timeline **T**, the first view considers the complete timeline as it takes all concept versions into account. The second view only considers a part of the timeline i.e., from the moment of creation of a concept until either the current time or the moment of its deletion (depending on the situation).

To reflect both views in our temporal logic, we extend the set of tense operators. We therefore introduce a parameterized version of the tense operators $\mathcal{H}$, $\mathcal{P}$ and $\mathcal{R}$. The original, non-parameterized version of the tense operators correspond to the view that takes all concept versions of the ontology into consideration, while the parameterized version, where the parameter refers to a concept name, only considers the concept versions associated with that particular concept. Figure 4.3 visualizes the difference between parameterized and non-parameterized tense operators (using the $\mathcal{R}$ tense operator as example). The element *now* is in the example equal to t_9. Note that the non-parameterized version of $\mathcal{R}$ refers to time point t_8, while the parameterized versions $\mathcal{R}(A)$ and $\mathcal{R}(B)$ respectively refer to time point t_4 and t_5.

To further clarify the difference between non-parameterized and parameterized tense operators, we give a small example expression illustrating the use and difference between the two. The exact syntax and semantics of the logic used in the example will be defined in the following subsection.

- $\langle\mathcal{R}\rangle\,(C \sqsubseteq D)$
  "In the previous version of the ontology, C was a subclass of D"

- $\langle\mathcal{R}(C)\rangle\,(C \sqsubseteq D)$
  "In the previous version of the concept C, C was a subclass of D"

4.2.2 Syntax and Semantics

In this section, we more precisely define the syntax and semantics of the modal extensions $\mathcal{H}$, $\mathcal{P}$, $\mathcal{S}$, $\mathcal{A}$, and $\mathcal{R}$ that we introduced before. We define the temporal logic, referred to as $\mathcal{SHOIN}(\mathbf{D})_T$, as a temporal extension of $\mathcal{SHOIN}(\mathbf{D})$. The syntax is defined as follows:

Definition 4.16 (Syntax). *Assume $\sigma \in \boldsymbol{N}$ to be a concept name. If ϕ and ψ are axioms, then so are $\langle\mathcal{H}\rangle\,\phi$ and $\langle\mathcal{H}(\sigma)\rangle\,\phi$, $\langle\mathcal{P}\rangle\,\phi$ and $\langle\mathcal{P}(\sigma)\rangle\,\phi$, $\langle\mathcal{R}\rangle\,\phi$ and $\langle\mathcal{R}(\sigma)\rangle\,\phi$, $\phi\,\langle\mathcal{S}\rangle\,\psi$, $\phi\,\langle\mathcal{A}\rangle\,\psi$, and $\langle@\ t\rangle\,\phi$ (the tense operators); $\neg\phi$, $\phi\wedge\psi$, $\phi\vee\psi$, and $\phi\rightarrow\psi$ (the common logic operators).*

Note that we only allow axioms to be modalized with the introduced tense operators, and prohibit the use of tense operators in concept definitions. The use of tense operators in concept definitions would make it possible to express temporal information within a domain (e.g., $TopManager \sqsubseteq Manager \sqcap \langle P\rangle\,\exists manages.TopProject$ expresses that a top manager is a manager and has managed at least one top project in the past). As we are not interested to express temporal information within a domain, but rather temporal relations between axioms, we restrict ourselves to the modalization of axioms solely.

To shorten the definition of the semantics of $\mathcal{SHOIN}(\mathbf{D})_T$, we introduce the following definitions. We define the disjunction $\phi\vee\psi$ and the implication $\phi\rightarrow\psi$ in terms of respectively the negation $\neg\phi$ and the conjunction $\phi\wedge\psi$.

$$\phi\vee\psi \;=\; \neg(\neg\phi\wedge\neg\psi) \tag{4.16}$$

$$\phi\rightarrow\psi \;=\; \psi\vee\neg\phi \tag{4.17}$$

The @-operator $\langle@\ t\rangle$, used to refer to an explicit moment in time, can be defined recursively in terms of the $\langle\mathcal{R}\rangle$ tense-operator as follows:

$$\begin{array}{lcll}\langle@\ t\rangle\,\phi & = & \phi & \text{if } t = now\\ \langle@\ t\rangle\,\phi & = & \langle@\ t+1\rangle\,(\langle\mathcal{R}\rangle\,\phi) & \text{if } t < now\end{array}$$

Before defining the semantics of $\mathcal{SHOIN}(\mathbf{D})_T$, we first introduce a number of auxiliary definitions. We define how we can derive a snapshot view from the version log given a point in time. A snapshot view $\mathbf{S}(t)$ is the set of all axioms ϕ kept in a version log that hold at a given moment in time $t\in\mathbf{T}$.

Definition 4.17 (Snapshot). *For a given time point $t\in\boldsymbol{T}$, a snapshot of a version log at moment t, notation $\boldsymbol{S}(t)$, is defined as follows*

$$\boldsymbol{S}(t) = \bigcup_{\forall\upsilon}\{\phi\in\boldsymbol{A}\mid \upsilon=\langle\sigma,\boldsymbol{A},\boldsymbol{D},s,t_s,t_e\rangle\wedge t_s\leq t\leq t_e\}$$

To be able to define the semantics of the parameterized tense operators $\langle \mathcal{R}(\sigma) \rangle$ and $\langle \mathcal{H}(\sigma) \rangle$, we define two views on the timeline $\mathbf{T}$. The first view, called $\mathbf{T}^{p}_{\sigma,t}$, contains the time points of the directly preceding concept version for a given concept name σ w.r.t. a given time point t. The second view, called $\mathbf{T}^{a}_{\sigma,t}$, contains the time points of all preceding concept versions for a given concept name σ w.r.t. a given time point t.

Definition 4.18 (Timeline $\mathbf{T}^{p}_{\sigma,t}$). *The set $\boldsymbol{T}^{p}_{\sigma,t} \subseteq \boldsymbol{T}$ is defined as the timeline spanning the previous concept version for a given concept name $\sigma \in \boldsymbol{N}$ w.r.t. a given time point $t \in \boldsymbol{T}$, so that*

$$\boldsymbol{T}^{p}_{\sigma,t} = \{t' \in \boldsymbol{T} \mid PreviousConceptVersion(\upsilon, \sigma, t) \wedge t_s \leq t' \leq t_e\}$$

Definition 4.19 (Timeline $\mathbf{T}^{a}_{\sigma,t}$). *The set $\boldsymbol{T}^{a}_{\sigma,t} \subseteq \boldsymbol{T}$ is defined as the timeline spanning all preceding concept versions for a given concept name $\sigma \in \boldsymbol{N}$ w.r.t. a given time point $t \in \boldsymbol{T}$, so that*

$$\begin{aligned}\boldsymbol{T}^{a}_{\sigma,t} = \{t' \in \boldsymbol{T} \mid \exists \upsilon = \langle \sigma, \boldsymbol{A}, \boldsymbol{D}, s, t_s, t_e \rangle, \exists \upsilon' = \langle \sigma, \boldsymbol{A}', \boldsymbol{D}', s', t'_s, t'_e \rangle .(\\ FirstConceptVersion(\upsilon, \sigma) \wedge PreviousConceptVersion(\upsilon', \sigma, t) \wedge \\ t_s \leq t' \leq t'_e)\}\end{aligned}$$

We define the semantics of the temporal extensions to $\mathcal{SHOIN}(\mathbf{D})$ by the following definition:

Definition 4.20 (Semantics). *Given an axiom ϕ, an interpretation $\mathcal{I}$ and a time point $t \in \boldsymbol{T}$, we extend the entailment relation $\mathcal{I}, t \vDash \phi$ (ϕ holds in $\mathcal{I}$ at moment t) as follows:*

$$\begin{array}{lll}
\mathcal{I}, t \vDash \phi & \text{iff} & \boldsymbol{S}(t) \vDash \phi, \quad \text{where } \phi \in \mathcal{SHOIN}(\boldsymbol{D}) \\
\mathcal{I}, t \vDash \phi \wedge \psi & \text{iff} & \mathcal{I}, t \vDash \phi \text{ and } \mathcal{I}, t \vDash \psi \\
\mathcal{I}, t \vDash \neg\phi & \text{iff} & \mathcal{I}, t \nvDash \phi \\
\mathcal{I}, t \vDash \langle \mathcal{R} \rangle \phi & \text{iff} & \exists t' \in \boldsymbol{T}.(t' = t - 1 \wedge \mathcal{I}, t' \vDash \phi) \\
\mathcal{I}, t \vDash \langle \mathcal{H} \rangle \phi & \text{iff} & \forall t' \in \boldsymbol{T}.(t' < t \wedge \mathcal{I}, t' \vDash \phi) \\
\mathcal{I}, t \vDash \langle \mathcal{P} \rangle \phi & \text{iff} & \exists t' \in \boldsymbol{T}.(t' < t \wedge \mathcal{I}, t' \vDash \phi) \\
\mathcal{I}, t \vDash \langle \mathcal{R}(\sigma) \rangle \phi & \text{iff} & \exists t' \in \boldsymbol{T}^{p}_{\sigma,t}.(\mathcal{I}, t' \vDash \phi) \\
\mathcal{I}, t \vDash \langle \mathcal{H}(\sigma) \rangle \phi & \text{iff} & \forall t' \in \boldsymbol{T}^{a}_{\sigma,t}.(\mathcal{I}, t' \vDash \phi) \\
\mathcal{I}, t \vDash \langle \mathcal{P}(\sigma) \rangle \phi & \text{iff} & \exists t' \in \boldsymbol{T}^{a}_{\sigma,t}.(\mathcal{I}, t' \vDash \phi) \\
\mathcal{I}, t \vDash \phi \langle \mathcal{S} \rangle \psi & \text{iff} & \exists t' \in \boldsymbol{T}.(t' < t \wedge \mathcal{I}, t' \vDash \psi \wedge \forall t'' \in \boldsymbol{T}.(t' < t'' < t \wedge \\
 & & \quad \mathcal{I}, t'' \vDash \phi)) \\
\mathcal{I}, t \vDash \phi \langle \mathcal{A} \rangle \psi & \text{iff} & \exists t' \in \boldsymbol{T}.(t' < t \wedge \mathcal{I}, t' \vDash \psi \wedge \exists t'' \in \boldsymbol{T}.(t' < t'' < t \wedge \\
 & & \quad \mathcal{I}, t'' \vDash \phi))
\end{array}$$

4.2.3 Examples

In this section, we illustrate the possibilities of the temporal logic proposed in the previous section by means of a few examples. This temporal logic

makes it possible to express temporal relations between axioms, and serves as foundation for the Change Definition Language we will introduce in the next section. Consider the following examples:

- "Over the course of history of an ontology, the Property *name* has always been a functional property and still is a functional property."

  $\langle \mathcal{H} \rangle \top \sqsubseteq (\leq 1\ name) \wedge \top \sqsubseteq (\leq 1\ name)$

- "During the existence of the Property *name*, it has always been a functional property and still is a functional property."

  $\langle \mathcal{H}(name) \rangle \top \sqsubseteq (\leq 1\ name) \wedge \top \sqsubseteq (\leq 1\ name)$

- "If in the previous version of an ontology, the Class *Company* was the range of the Property *worksFor*, then *Company* must still be the range of *worksFor*."

  $\langle \mathcal{R} \rangle \top \sqsubseteq \forall worksFor.Company \rightarrow \top \sqsubseteq \forall worksFor.Company$

- "The Class *Employee* has never been a subclass of the Class *Person* in the past, but it has become now."

  $\neg \langle \mathcal{P} \rangle\ Employee \sqsubseteq Person \wedge Employee \sqsubseteq Person$

4.3 Change Definition Language

In this section, we describe in detail the Change Definition Language. The purpose of this Change Definition Language is to allow the specification of change definitions i.e., to state the conditions a modification to an ontology should satisfy to be considered as an occurrence of a particular change definition.

The Change Definition Language makes it possible to define changes in a declarative fashion. Compared to other ontology evolution approaches, this Change Definition Language offers a number of advantages:

- **Extensibility of change definitions.** In current approaches, the set of changes offered by the approach are an integral part of the tool supporting the approach. The semantics of the changes are entangled in the source code, making it difficult to add, change or remove changes from this set as it requires an update of the software. In a commercial setting, this may even become impossible. In our approach, the set of changes are easier to modify as it only consists of adding, changing or removing change definitions, and doesn't require an update to the supporting tool.

- **Intuitive definitions.** We believe that change definitions are easier to specify in a declarative fashion than is the case with functional definitions. The Change Definition Language only requires ontology engineers to specify the situation before and after the change, instead of specifying *what* has to change. The latter requires a deep understanding of the API of the supporting tool, while the former only requires an understanding of the OWL ontology language (which is assumed to be already known by the ontology engineer) and of the Change Definition Language.

- **Overwrite change definitions.** In current approaches, maintainers of depending artifacts must rely on the changes (and their semantics) associated with a depending ontology, as these cannot be overwritten. This may result in awkward situations as is illustrated in Figure 4.4. An ontology O_3 depends on two ontologies O_1 and O_2. Assume that the evolution log of both ontology O_1 and O_2 lists a change C, but with different semantics. This situation makes it extremely confusing for the ontology engineer of ontology O_3 to understand the changes that have occurred to ontologies O_1 and O_2, as the semantics of the changes listed in the evolution logs may differ. In our approach, maintainers of depending artifacts can easily define their own change definitions, thereby overwriting the change definitions of depending ontologies.

- **Meta-changes.** The functional approach in defining changes taken by current approaches is not suited for defining meta-changes as meta-changes don't specify what has to change. In our approach, the same formalism (i.e., the Change Definition Language) can be used for the definition of both changes and meta-changes.

- **Domain dependent changes.** Domain dependent changes are defined in terms of a particular domain, and cannot be used directly for other domains. Other ontology evolution approaches don't support domain dependent changes. Moreover, the difficulty to extend the changes supported by current tools, makes supporting domain dependent changes even more unfeasible.

In the remainder of this section, we first introduce a meta-schema for OWL in Section 4.3.1. The reason to introduce this meta-schema is that the identifiers that can be used in the body of a change definition must be Classes and Properties defined in this meta-schema. The meta-schema itself is expressed in OWL. In Section 4.3.2, we discuss step by step the syntax of the Change Definition Language by means of its EBNF (Extended Backus-Naur Form) description. Although EBNF is not more powerful than standard BNF, its additional operators greatly improve the readability of the syntax. The full EBNF specification of the Change Definition Language

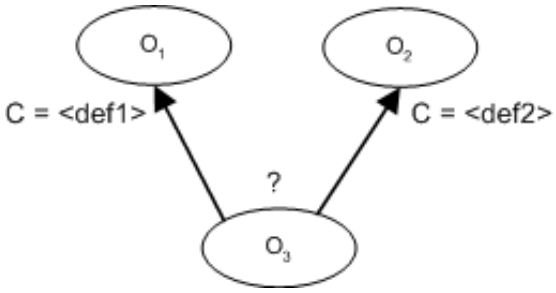

Figure 4.4: Changes with different semantics

can be found in Appendix 1. Instead of using the extended Description Logic syntax for the Change Definition Language, as introduced in Section 4.2.2, we adopt a more readable and computer-processable syntax instead.

We conclude this section with an overview of change definition sets (see Section 4.3.3). In its simplest form, a change definition set is a collection of change definitions. Ontology engineers and maintainers of depending artifacts can each define their own change definition set if desirable.

4.3.1 OWL Meta-Schema

In this Section, we describe a meta-schema of OWL. The Classes and Properties defined in this meta-schema form the vocabulary of the statements in the Change Definition Language that we introduce in the next section. As we focus on OWL DL in this dissertation, the meta-schema is restricted to OWL DL (thereby including OWL Lite) features only. The names of the Classes and Properties are kept as close as possible to the original OWL names. The meta-schema is expressed in OWL. In this section, we give a simplified overview of the most important aspects of the meta-schema. Interested readers are referred to `http://wise.vub.ac.be/ontologies/OWLmeta.owl` for the complete version of the OWL meta-schema.

Figure 4.5 shows the hierarchy of main Classes of the OWL meta-schema. The top Class is *Concept* with subclasses *Class*, *Property* and *Individual*. All *Concepts* have at most one ID (*hasID*). Two disjoint subclasses of *Property* exist i.e., *ObjectProperty* and *DatatypeProperty*. Note that *Restriction* is a subclass of *Class*.

Figure 4.6 depicts all possible properties of a *Class*. A *Class* can be defined as either a subclass (*subClassOf*) of another *Class* or equivalent (*equivalentClass*) to another *Class*. Furthermore, a *Class* can be disjoint with (*disjointWith*) other *Classes*, or can be the complement of (*complementOf*) exact one other *Class*. Finally, *Classes* can also be described in terms of a list of *Classes* using the *unionOf* and *intersectionOf* Properties, and as an enumeration of *Individuals* using the *oneOf* Property.

The different Properties of the *Property* Class are shown in Figure 4.7.

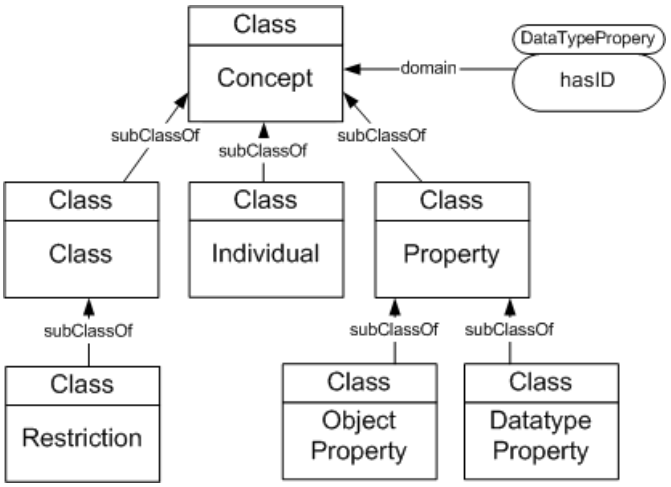

Figure 4.5: Main OWL concepts

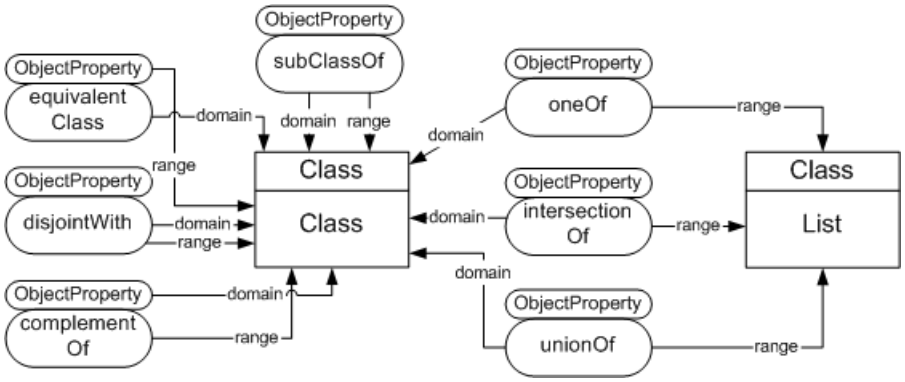

Figure 4.6: Properties of a Class

OWL allows to specify the *domain* and *range* of a *Property*. The *domain* of a *Property* is restricted to a *Class*, the *range* of a *Property* is restricted to either a *Class* or a *Datatype* where the *Datatype* represents an XML datatype (e.g., string, integer, boolean, ...). A *Property* can be defined as being a subproperty of another *Property* (*subPropertyOf*), or as being equal to another *Property* (*equivalentProperty*). Furthermore, a *Property* can be specified as being functional (*isFunctional*). Also *ObjectProperties* can be specified as being transitive (*isTransitive*), symmetric (*isSymetric*) and inverse functional (*isInverseFunctional*). Finally, an *ObjectProperty* can be the inverse (*inverseOf*) of another *ObjectProperty*.

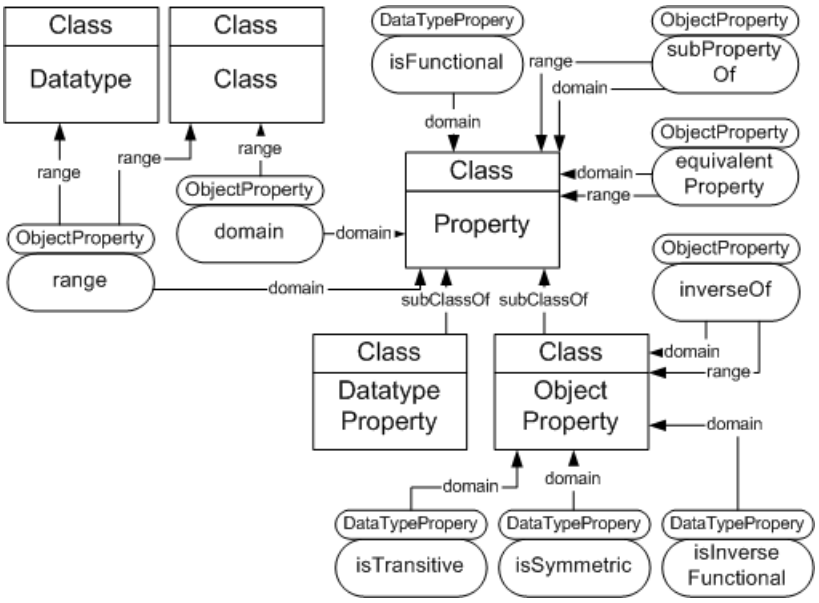

Figure 4.7: Properties of a Property

Figure 4.8 shows the different Properties of the *Restriction* Class. A *Restriction* is linked to exactly one *Property* using the *onProperty* Property. Two different restrictions can be distinguished: value restrictions and cardinality restrictions. For value restrictions, it can be specified that either all values or some values of the linked *Property* must be instances from a certain *Class* or *Datavalue* using respectively the *allValuesFrom* and *someValuesFrom* Properties. Value restrictions can also restrict a *Property* to a number of specific values using the *hasValue* Property. The range of this *hasValue* Property is either an *Individual* or a *Datarange*. For a cardinality restriction, either a minimum cardinality (*minCardinality*), a maximum cardinality (*maxCardinality*), or a specific cardinality (*cardinality*) can be specified for a given *Property*.

We conclude this section with an overview of the Properties of an *Individual*. Figure 4.9 shows the Properties of the *Individual* Class. An *Individual*

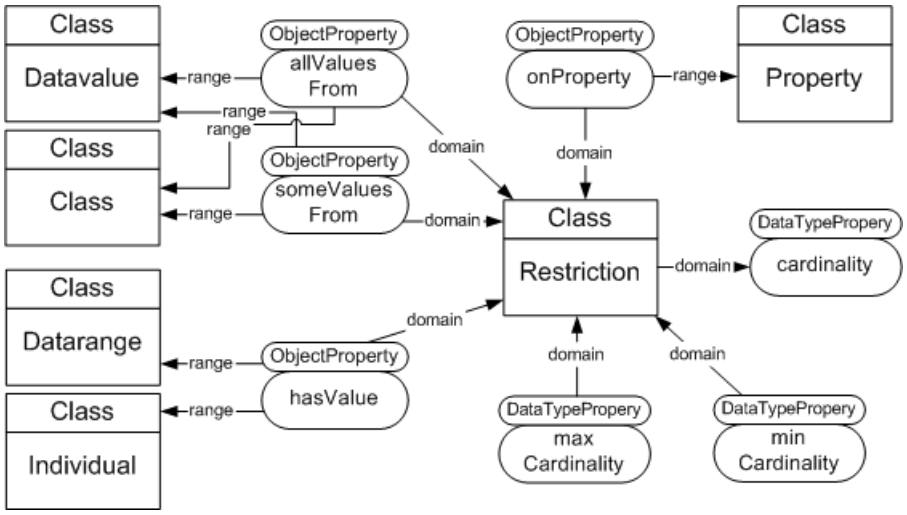

Figure 4.8: Properties of a Restriction

can be specified as an instance of multiple *Classes* using the *instanceOf* Property. Furthermore, it can have a number of property values (*withPropertyValue*). The range of *withPropertyValue* is the *PropertyValue* Class. A *PropertyValue* is linked to a *Property* and has as object an *Individual* or a *DataValue*. Finally, an *Individual* can be specified as being different from or as being the same as another *Individual* using respectively the *differentFrom* and *sameAs* Properties.

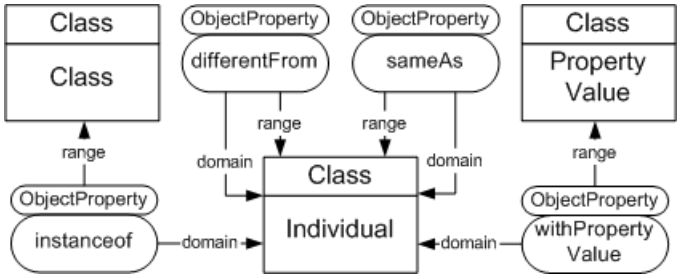

Figure 4.9: Properties of an Individual

4.3.2 Syntax

In this section, we describe the syntax of the Change Definition Language in detail by means of its EBNF description. To define its semantics, we link the Change Definition Language to the temporal logic $\mathcal{SHOIN}(\mathbf{D})_T$ introduced in the previous section. A change definition is a particular interpretation of an ontology modification i.e. the definition of a change formally specifies the modifications that should correspond with this change. These change

definitions are used in two ways in our approach. Firstly, ontology engineers specify their request for change in terms of change definitions. The definition of the change specifies how the ontology has to change (see section 5.2). Secondly, the change definitions also allow detecting other changes i.e., the modifications may satisfy other change definitions then the one used in the change requests). This is possible by evaluating change definitions against the version log (see Section 5.3).

Change Definition A change definition is composed of a header followed by a body. The header consists of both a name (i.e., the identifier) that identifies the definition and an arbitrary number of parameters (i.e. the parameter list). The parameters of the parameter list are separated from each other by means of a comma and always start with a question mark.

Furthermore, the parameters in the header can optionally be annotated to reflect the role of the parameters in the change definition. These annotations are both used to clarify the role that the parameters play in the change definition, and play a role in the change detection and change recovery phases (as we will discuss in Section 5.3). We distinguish the following three types of roles:

- **Subject.** This type of role indicates that the associated parameter represents the ontology concept that is the subject of the change i.e., the ontology concept that is changed.
- **Old.** This type of role indicates that the associated parameter represents the old object of the Property that has changed in the subject.
- **New.** This type of role indicates that the associated parameter represents the new object of the Property that has changed in the subject.

Consider as example the header shown below. The parameter annotations help to understand the role that these parameters play in the definition: the subclass property of subject `?x` changes from `?y` (the old object) to `?z` (the new object).

```
changeSubClassOf([subject]?x, [old]?y, [new]?z)
```

The EBNF description of a change definition is given below:

```
changeDefinition  ::=  identifier
                       '(' (parameterListH)? ')' body ';'
parameterListH    ::=  parameterH (',' parameterH)*
parameterH        ::=  ('[' role ']')? '?' IDENT
role              ::=  ('subject' | 'old' | 'new')
```

Change Definition Body The body of a change definition consists of a condition type (i.e., either ‘:=’ or ‘:<’) followed by a condition. The condition is an expression, which will be discussed in the next paragraph. The meaning of the condition type can be best compared with the notion of *necessary conditions* and *necessary & sufficient conditions* as found in logics. The difference between the two condition types is explained as follows:

- **Necessary condition (‘:<’).** This condition type indicates that the condition of the change definition is only a necessary condition. This has as consequence that each change that is an occurrence of such a change definition must conform to the condition of the change definition, but this does not mean that each change that conforms to the condition of the change definition is automatically an occurrence of that change definition. In this case, it is the responsibility of the user to decide whether or not it concerns an occurrence of the detected change.
- **Necessary & sufficient condition (‘:=’).** This condition type indicates that the condition of the change definition is both a necessary and sufficient condition. This has as consequence that each change that is an occurrence of such a change definition must conform to the condition of the change definition, and that each change that conforms to the condition of the change definition is by definition an occurrence of that change definition.

The two condition types play an important role in the change detection phase of our approach. Occurrences of change definitions are detected by querying the version log using the change definitions. When a change to the ontology conforms to the condition of a change definition with the necessary & sufficient condition type, this change is automatically added to the evolution log as an occurrence of the change definition without any human intervention. On the other hand, when a change to the ontology conforms to the condition of a change definition with the necessary condition type, this change cannot be added automatically to the evolution log. Instead, a warning is send to the ontology engineer to inform him of a possible occurrence of a change definition. It is his duty to decide whether or not the change is an occurrence of the detected change definition.

Consider the following example as illustration of the usefulness of the necessary condition type. A company maintains an ontology describing properties of their employees and the projects they are assigned to. When an employee is removed from the ontology, it either means that the employee has resigned or was fired by the company (assuming no alternatives exist). From the information stored in the ontology, it is impossible to decide automatically the cause of the removal. However, the company would like to keep track of the cause of the change in an evolution log. The ontology

engineer therefore defines change definitions for both cases as follows (the exact syntax of the condition will be explained further on in this section):

```
employeeResigned(?x) :<
    <PREVIOUS>(instanceOf(?x, Employee)) AND
    (NOT instanceOf(?x, Employee));
employeeFired(?x)    :<
    <PREVIOUS>(instanceOf(?x, Employee)) AND
    (NOT instanceOf(?x, Employee));
```

The conditions of both change definitions are the same. It specifies that `?x` was in the previous version of the ontology still an employee, but this is no longer the case. When a change to the ontology conforms the condition, both change definitions are suggested to the ontology engineer as a possible candidate. It is the responsibility of the ontology engineer to select the correct one.

The EBNF description of the change definition body is as follows:

```
body              ::=  (':=' | ':<') condition
condition         ::=  expression
```

Expression The condition of a change definition consists of an expression. Expressions can be formed using the well-known logical operators $\neg$, $\wedge$ and $\vee$, respectively `NOT`, `AND` and `OR` in the syntax of the Change Definition Language. The standard precedence rules apply for the logical operators i.e., first `NOT`, then `AND`, followed by `OR`. Parentheses can be used to change the standard preceding rules (see `parenExpression`). Compared to the temporal logic introduced in the previous section, the `NOT` operator corresponds to $\neg\phi$, the `AND` operator corresponds to $\phi \wedge \psi$, and the `OR` operator corresponds to $\phi \vee \psi$. The building blocks of expressions are statements, temporal expressions and native functions.

```
expression        ::=  term
term              ::=  factor ('OR' factor)*
factor            ::=  secondary ('AND' secondary)*
secondary         ::=  (primary | 'NOT' primary)
primary           ::=  (statement |
                        parenExpression |
                        tempExp |
                        nativeFunction)
parenExpression   ::=  '(' expression ')'
```

Statement A statement consists of an identifier and one or two arguments. The first argument is called the *subject* of the statement, the second

argument is called the *object*. The identifier is either a Class or a Property defined in the OWL meta-schema (see Section 4.3.1). The subject is a parameter or an identifier, while the object can also be a value (i.e., to be able to express datatype Properties). Consider as example the following statements:

- `Class(?x)` states that `?x` is a Class;
- `subClassOf(?x, Person)` states that `?x` is a subclass of the Class Person;
- `isTransitive(?x, "true")` states that `?x` is a transitive Property.

An optional asterix symbol (`'*'`) following the identifier of the statement indicates whether the transitive characteristic of a Property (when present) should be taken into account. Omitting the asterix symbol restricts the statement to only direct properties. So `P(?x, ?y)` without the asterix symbol can be expressed as $\forall x, \forall y.(P(x,y) \wedge \neg\exists z.(P(x,z) \wedge P(z,y) \wedge z \neq x \wedge z \neq y))$.

The EBNF description of the statement syntax is given below:

```
statement          ::=  identifier ('*')? '(' subject
                        (',' object)? ')'
subject            ::=  (parameter | identifier)
object             ::=  (parameter | identifier | value)
parameter          ::=  '?' IDENT
identifier         ::=  IDENT
value              ::=  '"' IDENT '"'

IDENT              ::= ('a'..'z' | 'A'..'Z')
                       ('a'..'z' | 'A'..'Z' | '_' |
                        '0'..'9')*
```

Temporal Expression Temporal expressions are used to formulate expressions that held in the past. A temporal expression is either a unary temporal expression or a binary temporal expression. The former consists of a unary tense operator or a time reference followed by an expression, the latter consists of a binary tense operator followed by two comma-separated expressions. The unary tense operators available are `ALWAYS`, `SOMETIME` and `PREVIOUS`, respectively corresponding to the tense operators $\mathcal{H}$, $\mathcal{P}$ and $\mathcal{R}$ in the temporal logic. Note that the unary tense operators can be augmented with a parameter to reflect the parameterized version of the tense operators. Also two binary tense operator exists, `SINCE` and `AFTER`, which corresponds to respectively the $\mathcal{S}$ and $\mathcal{A}$ tense operators in the temporal logic. Finally the time reference, `T`, allows to refer explicitly to a moment in time and

corresponds to the @-operator in the temporal logic. The EBNF notation is specified as follows:

```
tempExp          ::=  (unaryTempExp | binaryTempExp)
unaryTempExp     ::=  '<' (unaryTenseOp | timeRef) '>'
                      parenExpression
binaryTempExp    ::=  '<' binaryTenseOp '>'
                      '(' expression ',' expression ')'
unaryTenseOp     ::=  ('ALWAYS' |
                       'SOMETIME' |
                       'PREVIOUS')
                      ( '(' parameter |
                            identifier ')' )?
binaryTenseOp    ::=  'SINCE' | 'AFTER'
timeRef          ::=  'T' '(' INTEGER ')'

INTEGER          ::=  ('0'..'9')+
```

The syntax for temporal expressions explained above, allows us to define the following examples:

- `<ALWAYS>(Class(?x))` states that `?x` has always been a Class during the entire lifetime of the ontology;
- `<ALWAYS(?x)>(Class(?x))` however states that `?x` has always been a Class during the whole lifetime of `?x`;
- `<SINCE>(domain(?x, Student), subClassOf(Student, Person))` states that Student has always been the domain of `?x` since Student was a subclass of Person;
- `<AFTER>(domain(?x, Student), subClassOf(Student, Person))` states that Student has been the domain of `?x` at some time after Student was a subclass of Person;
- `<T(28)>(Class(?x))` states that `?x` was a Class at time point 28.

Native functions The Change Definition Language also supports a few native functions to compare two ontology concept or values. The native function `equal` expresses equality between concepts or values. Concepts are considered equal if their concept names are identical; values are considered equal if the values are identical. The native functions `lt` and `gt` are only applicable for integer, float, string, date and dateTime datatypes. The function `lt` expresses that the first argument is less then the second argument, the function `gt` expresses that the first argument is greater then the second argument.

The EBNF description of the native functions is specified as follows:

```
nativeFunction     ::=  nativeID '('
                        nativeArg ','
                        nativeArg ')'
nativeArg          ::=  (parameter | identifier | value)
nativeID           ::=  'equal' | 'lt' | 'gt'
```

4.3.3 Change Definition Set

We described in the previous section the Change Definition Language that allows ontology engineers and maintainers of depending artifacts to formally specify definitions of changes. As mentioned before, ontology engineers and maintainers of depending artifacts must be able to express their own set of ontology changes they are interested in. They do this by creating their own *change definition set*. A change definition set is a collection of *conceptual change definitions*. Conceptual change definitions are defined in terms of one or more *base change definitions*, which are definitions expressed in the Change Definition Language that we introduced in the previous section. To sum up, a change definition set is the set of conceptual change definitions that an ontology engineer is interested in. A conceptual change definition is defined in terms of base change definitions that are expressed in the Change Definition Language. An actual change to the ontology is an *occurrence* of a particular conceptual change definition when at least one of its base change definitions are satisfied. We first formulate the definition of a conceptual change definition, before we define a change definition set.

To be able to define a conceptual change definition, we assume χ to be a change definition set. Furthermore, we call $\boldsymbol{\Delta}_\chi$ the set of all conceptual change definitions in a given change definition set χ. We then define a conceptual change definition as follows:

Definition 4.21 (Conceptual Change Definition). *Assume $\boldsymbol{L}$ to be the set of all possible base change definitions. We then define a conceptual change definition δ as a tuple so that $\delta = \langle \boldsymbol{R}_\delta, \boldsymbol{C}_\delta, \boldsymbol{H}, \boldsymbol{D} \rangle$ where $\boldsymbol{R}_\delta \subseteq \boldsymbol{L}$ is a finite set of base change definitions to be used in the change request and implementation context, $\boldsymbol{C}_\delta$ is a finite set of base change definitions to be used in the change detection context, $\boldsymbol{H}$ is the set of supertypes of this conceptual change definition so that $\boldsymbol{H} \subseteq \boldsymbol{\Delta}_\chi \setminus \{\delta\}$, and $\boldsymbol{D}$ is a finite set listing all disjoint conceptual change definitions so that $\boldsymbol{D} \subseteq \boldsymbol{\Delta}_\chi \setminus (\boldsymbol{H} \cup \{\delta\})$.*

A conceptual change definition is defined in terms of one or more base change definitions. As a conceptual change definition is used in different contexts (to request and implement changes, and to detect changes), different base change definitions can be provided for these different contexts. A base change definition assigned to one context will only be used for that particular context (e.g., only for detecting changes). A single base change definition can be assigned to both contexts meaning that the definition will be used

in both contexts. Consider as an example the change where an employee changes jobs. The company of the employee describes the current situation of the company by means of an ontology. An employee changes jobs either when he holds a new function within the company or when he trades-in his old company for a new one. To express this change, we could create an conceptual change definition `changedJobs` in terms of the following two base change definitions:

```
newFunction(?x) :=
    <PREVIOUS(?x)>(Individual(?x) AND
      withPropertyValue(?x, ?y) AND
      ofProperty(?y, function) AND object(?y, ?z)) AND
    Individual(?x) AND withPropertyValue(?x, ?u) AND
      ofProperty(?u, function) AND object(?u, ?v) AND
    (NOT equal(?z, ?v));

leftCompany(?x) := <PREVIOUS(?x)>(Employee(?x)) AND
                   (NOT Employee(?x));
```

Note that conceptual change definitions can be defined as a subtype of other conceptual change definitions i.e., if a modification of an ontology is an occurrence of a conceptual change definition, then it is also an occurrence of its super types. Finally, conceptual change definitions can be defined to be disjoint with other conceptual change definitions i.e., if a modification of an ontology is an occurrence of an ontology change, it cannot be an occurrence of its disjoint conceptual change definitions.

Now that we have defined a conceptual change definition, this leaves us with the definition of a change definition set. A change definition set is composed of a set of conceptual change definitions. Furthermore, a change definition set allows to import other change definition sets (e.g., the change definition set of a depending artifact may reuse the change definition set of an ontology it depends on). The definition is specified as follows:

Definition 4.22 (Change Definition Set). *A change definition set χ is a two-tuple so that $\chi = \langle \boldsymbol{S}, \boldsymbol{M} \rangle$ where $\boldsymbol{S} = \{\delta_1, \ldots, \delta_n\}$ is the set of ontology changes, and $\boldsymbol{M} = \{\chi_1, \ldots, \chi_m\}$ is the set of imported change definition sets.*

4.4 Evolution Log

In this section, we describe in more detail the evolution log. The purpose of an evolution log is to give an overview of the evolution of an ontology by listing all changes that have occurred. It serves maintainers of depending artifacts in understanding the changes occurred, and is therefore a great

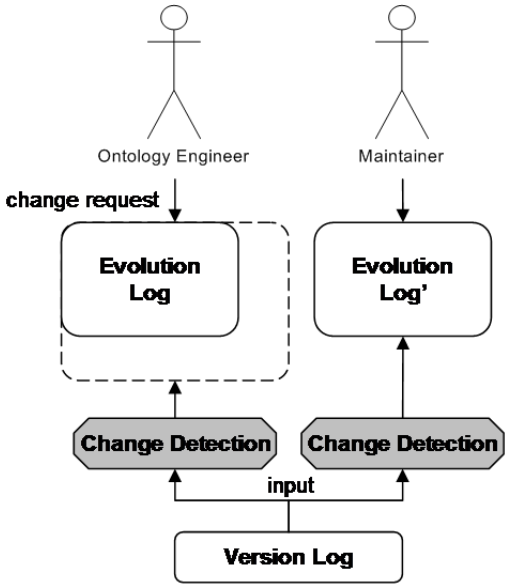

Figure 4.10: Evolution log creation

benefit in the decision whether to update or not. Note the difference between the version log and the evolution log: the former lists the different versions of the ontology concepts, the latter lists the interpretations of these versions in terms of conceptual change definitions.

In current approaches, the evolution log is the result of the changes listed in the change request. As already discussed thoroughly in 3.2.1, this approach has a number of serious drawbacks. In our approach, we therefore extend the evolution log with, besides requested and deduced changes, occurrences of conceptual change definitions detected during the change detection phase. Moreover, different users can create their own evolution log by specifying their own change definition set. Figure 4.10 summarizes it graphically.

An evolution log is populated with *change occurrences*. A change occurrence is an instantiation of a conceptual change definition. A change occurrence keeps a reference to the conceptual change definition it is an occurrence of. Remember that the conceptual change definitions itself are defined in a change definition set. A change occurrence also keeps track of the header of the base change definition it satisfies and parameter bindings by means of key-value pairs $\langle p, v \rangle$ where p is a parameter denoted in the header of the base change definitions associated with the conceptual change definition and v is the value bound to the parameter. Unbound parameters are not listed in a change occurrence. Note that, as we will see in Section 3.2.1, more than one set of parameter bindings may exist for a change occurrence as a consequence of undecidability. Furthermore, it also contains a

time point t representing the moment in time of the change occurrence.

For a given conceptual change definition, we define the change occurrences of that conceptual change definition as follows:

Definition 4.23 (Change Occurrences). *The change occurrences of a given conceptual change definition $\delta \in \boldsymbol{\Delta}_\chi$ (where $\delta = \langle \boldsymbol{R}_\delta, \boldsymbol{C}_\delta, \boldsymbol{H}, \boldsymbol{D} \rangle$) is defined as a set $\boldsymbol{o}_\delta = \{\langle t, \boldsymbol{G} \rangle\}$ so that $t \in \boldsymbol{T}$ is the moment in time of the occurrence and $\boldsymbol{G} = \{\langle d, \boldsymbol{B} \rangle\}$ where $d \in \boldsymbol{R}_\delta \cup \boldsymbol{C}_\delta$ and $\boldsymbol{B} = \{\langle p, v \rangle\}$ is a set of parameter bindings.*

We conclude this section with the definition of an evolution log, defined as a set of change occurrences:

Definition 4.24 (Evolution Log). *For a change definition set χ, an evolution log $\boldsymbol{EL}$ is a set of change occurrences, so that*

$$\boldsymbol{EL} = \bigcup_{\forall \delta \in \boldsymbol{\Delta}_\chi} o_\delta$$

4.5 Summary

In this chapter, we discussed the foundations of our ontology evolution framework i.e., the version log, the temporal logic $\mathcal{SHOIN}(\mathbf{D})_T$ that forms the basis of the Change Definition Language, and the evolution log.

We first introduced the notion of a version log. The general approach taken was discussed and a formal model of the version log was presented. The formal model defined when we consider an axiom to be a definition of respectively a Class, Property or Individual. Furthermore, it introduced the notion of a *concept version*, which describes a version of its concept by listing the axioms that form the definition of a concept at a given moment in time, and a *concept evolution* that describes the evolution of a particular concept by keeping track of its different concept versions. Finally, a version log is defined as a collection of concept evolutions.

Next, we presented the $\mathcal{SHOIN}(\mathbf{D})_T$ temporal logic, which is a hybrid-logic approach. The logic introduces a number of tense operators to express temporal relations of the past tense and an @-operator to refer to specific moments in time. To consider different views on the evolution of an ontology, the temporal logic distinguishes between parameterized and non-parameterized tense operators. The former considers only the evolution of a particular concept, while the latter considers the evolution of the complete ontology. Furthermore, the syntax and semantics of $\mathcal{SHOIN}(\mathbf{D})_T$ are defined.

Based on this temporal logic, we defined the Change Definition Language. We discussed the syntax of the Change Definition Language by means of its EBNF description. Furthermore, we introduced the notion of

a *change definition set* as a collection of *conceptual change definitions*. A conceptual change definition is defined as a set of *base change definitions*. Base change definitions are expressed in terms of the Change Definition Language.

Finally, we introduced the notion of an evolution log that keeps track of occurrences of change definitions at given moments in time.

Chapter 5

Change Definitions

In the previous chapter, we introduced the foundations of our ontology evolution approach which form the basis of the different phases of the approach. These foundations include the version log used to describe the history of an ontology and of its individual concepts, a hybrid-logic temporal extension to the $\mathcal{SHOIN}(\mathbf{D})$ Description Logic to express temporal relations between axioms, a Change Definition Language to formally define the semantics of changes based on the previously mentioned temporal logic, and the evolution log that serves as an interpretation of an ontology evolution in terms of ontology changes defined by means of the Change Definition Language.

Now that the foundations of the approach are in place, the focus of this chapter in on the specification of change definitions in terms of the Change Definition Language. In our approach, the change definitions are used for two different purposes: either to request and apply changes to an ontology or to detect changes that have occurred but that were not explicitly requested. In Section 5.1, we go into more detail on both purposes. Because we use the Change Definition Language for different purposes, we evaluate change definitions differently depending on the purpose concerned. In Section 5.2, we discuss how change definitions can be used to specify a change request and how such a change request is evaluated. In Section 5.3, we discuss how change definitions are evaluated to detect changes that have occurred.

In the second part of this chapter, we explain in detail how the Change Definition Language is used to define changes and meta-changes, thereby highlighting a number of difficulties that should be taken into account. In Section 5.4, we present a complete and minimal set of primitive change definitions for OWL. In Section 5.5, we introduce a number of representative complex change definitions. Note that it is impossible to provide a complete set of complex change definitions, as there exists an infinite number of possible complex changes. In Section 5.6, we do the same for complex meta-changes. We conclude this chapter with a summary in Section 5.7.

5.1 Purpose

As already mentioned in the introduction of this chapter, conceptual change definitions serve two purposes in our approach. Firstly, they are used by ontology engineers to request and implement changes. Secondly, they are used by the ontology evolution framework to detect occurrences of conceptual change definitions. To reflect both purposes of a conceptual change definition, we have defined a conceptual change definition δ as a tuple of the form $\langle \mathbf{R}_\delta, \mathbf{C}_\delta, \mathbf{H}, \mathbf{D} \rangle$ (as shown in Section 4.3.3 in the previous chapter). Important for the discussion in this chapter are the sets $\mathbf{R}_\delta$ and $\mathbf{C}_\delta$ that contain base change definitions respectively to be used for change request or to be used for change detection. Note that a single base change definition may be an element of $\mathbf{R}_\delta \cap \mathbf{C}_\delta$, although this is not always possible as we will illustrate in Section 5.4 and Section 5.5.

The Change Definition Language, introduced in Section 4.3, can be seen as both a manipulation and a query language depending on the purpose of the definition. Base change definitions that are used for change requests result in the manipulation of a version log (and subsequently the ontology), while base change definitions that are used for change detection result in a querying of a version log. While most languages that offer the manipulation and querying of data possess different constructs for both purposes, we don't offer dedicated language constructs in our Change Definition Language for both purposes[1], but rather evaluate base change definitions differently depending on its purpose.

Before we discuss the use and evaluation of conceptual change definitions for the purpose of change requests in Section 5.2 and for the purpose of change detection in Section 5.3, we first introduce an example of a conceptual change definition that will be used throughout both these sections. We take as example the *addSubClassOf* conceptual change definition that represents the addition of a *SubClassOf* Property between two Classes. The conceptual change definition consists of a single base change definition $d \in \mathbf{R}_\delta \cap \mathbf{C}_\delta$. We define the base change definition d as follows:

```
addSubClassOf(?s, ?o) :=
  NOT <PREVIOUS>(subClassOf(?s, ?o)) AND
  subClassOf(?s, ?o);
```

The base change definition is rather straightforward as it states that a certain `?s` was previously not a subclass of `?o`, but that it is a subclass of `?o` at present.

[1] Although some limitations exist on the language constructs allowed for both purposes. See Section 5.2.2 for more information.

5.2 Change Request

As already mentioned in Section 3.2.1, ontology engineers can express their request for changes by means of a change request. In this section, we explain the structure of such a change request (see Section 5.2.1), present the criteria the base change definitions of an ontology change should meet to be applicable in a change request (see Section 5.2.2), and discuss the evaluation of a change request (see Section 5.2.3).

5.2.1 Change Request Specification

An ontology engineer can request changes to an ontology by specifying a change request. A change request is an ordered set where each element of the set is specified in terms of a conceptual change definition that the ontology engineer desires to apply with concrete values for parameters of the base change definition(s) included in the conceptual change definition. More specifically, we define a change request as follows:

Definition 5.1. *Assume χ to be a change definition set. A change request is defined as an ordered set $\boldsymbol{R} = \{\langle \delta, \boldsymbol{G} \rangle\}$ where $\delta \in \Delta_\chi$ is the conceptual change definition one requests and $\boldsymbol{G} = \{\langle p, v \rangle\}$ is a set of parameter bindings.*

Consider as example an ontology engineer who wants to add a *subClassOf* Property between a Class *Student* and a Class *Person*. He formulates a change request $\mathbf{R}$ making use of a conceptual change definition *addSubClassOf* (see Section 5.1 for a definition of this change):

$$\mathbf{R} = \{\langle addSubClassOf, \{\langle ?s, Student \rangle, \langle ?o, Person \rangle\}\rangle\}$$

When specifying a change request, an ontology engineer is not obliged to provide values for all parameters in the header of the base change definition(s) of the used ontology change[2]. When a parameter is not bound to a value, a default value is created when applying the conceptual ontology change. When it concerns a Class (including a Restriction), Property or Individual, this results respectively in an anonymous Class, Property or Individual. When it concerns a datatype, this results in the default value for that respective datatype (e.g., the default value for the 'string'-datatype is the empty string).

Consider as an example of unbound variables an ontology engineer who wants to add an 'allValuesFrom'-restriction on a Property `worksFor` with `Company` as value. He specifies the following change request $\mathbf{R}$ making use of a conceptual change definition *addAllValuesFromRestriction* (see Section 5.4 for a definition of this change):

$$\mathbf{R} = \{\langle addAllValuesFromRestriction, \{\langle ?p, worksFor \rangle, \langle ?o, Company \rangle\}\rangle\}$$

[2]Although this feature should be handled with care as we will see in Section 5.2.3

Note that, as the header of the base change definition of the conceptual change definition is `addAllValuesFromRestriction(?r, ?p, ?o)`, no value is provided for the `?r` parameter in the change request. Evaluating the change request will result in the creation of an anonymous Restriction on the Property `worksFor` with all values set to `Company`. The new anonymous Restriction will be bound to the `?r` parameter.

When the ontology engineer wants to change an ontology so that all values of a Property `worksFor` are instances of `Company` for a Class `Person`, he may use the following change request $\mathbf{R}$ making use of the conceptual changes definitions *addAllValuesFromRestriction* and *addSubClassOf*:

$$\begin{aligned}&\mathbf{R} = \{\\&\quad \langle addAllValuesFromRestriction, \{\langle ?p, worksFor\rangle, \langle ?o, Company\rangle\}\rangle,\\&\quad \langle addSubClassOf, \{\langle ?s, Person\rangle, \langle ?o, ?r\rangle\}\rangle\\&\}\end{aligned}$$

The change requests consists of two elements. The first element requests to add an anonymous Restriction to the ontology, the second element requests to add a subclass Property between the Class `Person` and the newly created anonymous Restriction (as the value bound to `?r` is used as value for the second parameter `?o`).

Recall that a change request by an ontology engineer may turn the ontology into an inconsistent state. To avoid an ontology from becoming inconsistent, deduced changes are added by the approach to a change request to ensure that when the change request is eventually evaluated, the ontology evolves from one consistent version into another consistent version. After each change requested in a change request, our framework verifies if the ontology would remain consistent when the requested change is applied. If consistency is no longer maintained, our framework adds deduced changes to the change request for that particular requested change. The approach we take to determine the deduced changes required to maintain consistency is explained in detail in Chapter 6. To associate a set of deduced changes with a requested change in a change request, we define the set $deducedChanges(\mathbf{R}, u_i)$ as follows:

Definition 5.2 (Deduced Changes). *Assume $\boldsymbol{R}$ to be a change request and $u_i \in \boldsymbol{R}$ a requested change. $deducedChanges(\boldsymbol{R}, u_i) = \boldsymbol{R}_d$ where $\boldsymbol{R}_d$ is a change request filled with deduced changes for a requested change u_i of a change request $\boldsymbol{R}$.*

Note that the set of deduced changes associated with a requested change is in fact again a change request itself. As a consequence, a deduced change can again be associated with a set of deduced changes. Figure 5.1 graphically represents a change request augmented with sets of deduced changes. The change request contains a number of requested changes $(u_1, \ldots, u_n)$ and a

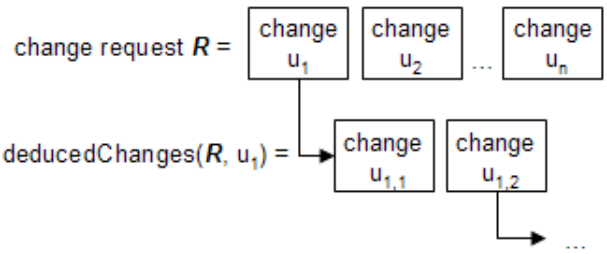

Figure 5.1: A change request with deduced changes

set of deduced changes $(u_{1,1}, u_{1,2})$ for the requested change u_1. In its turn, another set of deduced changes is associated with the deduced change $u_{1,2}$.

5.2.2 Restrictions

For a conceptual change definition δ to be applicable in a change request, its base change definitions $d \in R_\delta$ should meet a number of prerequisites that we will discuss in this section. When evaluating the changes of a change request, we want the outcome to be predictable and consistent. In order to achieve this, we need to restrict the use of the Change Definition Language when defining change definitions for the purpose of change requests to avoid non-determinism.

In general, a base change definition expresses a change in terms of differences between the current version of ontology concepts (by using non-temporal expressions) and preceding versions of ontology concepts (by using temporal expressions). When defining change definitions for change requests, the temporal expressions express the situation before the change and can be seen as preconditions of the change, while the non-temporal expressions express the situation after the change and can be seen as post-conditions of the change. In other words, the body of a change definition consists of the following form:

precondition `AND` *postcondition*

A first restriction concerns the use of tense operators in the precondition of a base change definition. Base change definitions used for requesting changes take as precondition only the version of the ontology right before the change into account. E.g., when adding a new Class *Person* to an ontology, the precondition is that right before this change no Class *Person* exists. We therefore restrict the temporal expressions of the precondition to use solely the `<PREVIOUS>` operator as tense operator.

The postcondition of a base change definition on the other hand expresses the end result of a change i.e., how the new version of the ontology should look like. Important is that the postcondition must be deterministic.

To assure determinism, the following restrictions of the Change Definition Language apply to the postcondition[3]:

- **Prohibit `OR`-expressions:** `OR` expressions typically lead to nondeterminism as either one of the operands can be chosen to result into a valid end result.
- **Prohibit transitive properties:** the use of transitive properties by means of the asterix-symbol ('*') is prohibited as it leads to nondeterminism. E.g., there exists multiple ways to change an ontology so that the expression `subClassOf*(Student, Person)` is satisfied (unless these two Classes are the only two Classes defined).
- **Prohibit abstract concepts:** there exists a number of abstract concepts in the OWL meta-schema we defined in Section 4.3.1 i.e., concepts that can only have indirect instances through their children. An example of such an abstract concept is the *Property* concept as properties are always either object properties or datatype properties (never just a Property). The use of abstract concepts is prohibited in postconditions.
- **Prohibit native functions:** no native functions (i.e., `equal`, `lt`, and `gt`) can be used in preconditions. These native functions are only used to express a comparison between concepts or values, but never can result into effective changes to an ontology when used in the postcondition.

5.2.3 Evaluation

The evaluation of a change request results into one or more new concept versions in the version log representing the new state of the ontology after changes. Recall from Section 3.2 that a change request evaluation doesn't directly result in a modified ontology, but only updates the associated version log. The requested changes are only applied to the actual ontology on execution of the change implementation phase. In this section, we discuss both the change request evaluation as well as the implementation of the changes in the actual ontology.

Change Request Evaluation

As defined in Section 5.2.1, a change request specified by an ontology engineer consists of a list of requested changes. To ensure consistency maintenance, the framework possibly associates deduced changes to requested

[3]For these restrictions, we assume the postcondition to be in Negation Normal Form (NNF)

changes of the change request. To evaluate a change request, all its requested changes are evaluated. Note that a requested change is evaluated first before evaluating the associated set of deduced changes. The evaluation of a set of deduced changes is identical to the evaluation of a *normal* change request. Note that when evaluating a change request, a requested change and all its deduced changes are evaluated in its entirety before evaluating the next requested change of the change request.

Algorithm 1 Change request evaluation

```
evaluateChangeRequest(R, G):
    for all u in R do
        δ = conceptualChangeDefinition(u)
        setParameterBindings(u, G)

        for all d ∈ R_δ in δ do
            if res = evaluatePrecondition(d, G) then
                evaluatePostcondition(d, G)
                continue
            end if
        end for all

        if res then
            dc = deducedChanges(R, u)
            evaluateChangeRequest(dc, G)
        else
            undoChanges
        end if
    end for all
```

The pseudo-code to evaluate a change request is shown in Algorithm 1. The algorithm iterates over all requested changes of a change request **R**. For each requested change u, it retrieves its conceptual change definition δ and sets its parameter bindings. For each base change definition d that may be used in a change request, it is verified whether the precondition of the base change definition holds. If the precondition holds, the postcondition of the base change definition is evaluated and remaining base change definitions are skipped. If the precondition fails, the next base change definition (if any) is tried. Note that ontology engineers may not rely on the evaluation order of the algorithm, but rather should ensure that the preconditions of the different base change definitions are exclusive. When the evaluation of the requested change succeeds, associated deduced changes are retrieved and evaluated. However, when the requested change was not successfully evaluated (because no precondition of any of the base change definitions

was satisfied), the complete change request is canceled and all new concept versions resulted from the change request are undone. After a change request is successfully evaluated, the requested and deduced changes are added to the evolution log as occurrences of change (see Section 4.4). In the following paragraphs, we provide more details concerning the evaluation of pre- and postconditions.

Preconditions The evaluation of a precondition boils down to the execution of the temporal query that the precondition represents on the version log. Before executing the temporal query, the current time point (i.e., *now*) is raised by one to assure that the precondition verifies the state of the ontology just before the change. The result of the query is a set of a set of parameter bindings where the parameters are bound to the concepts satisfying the precondition. A precondition holds if the query execution produces a non-empty result. Otherwise, we say that the precondition doesn't hold or fails. Consider as example the following change request:

$$\mathbf{R} = \{\langle addSubClassOf, \{\langle ?s, Student\rangle, \langle ?o, Person\rangle\}\rangle\}$$

When we look at the base change definition of the *addSubClassOf* conceptual change definition (as given in Section 5.1), the precondition with bound variables is:

```
NOT <PREVIOUS>(subClassOf(Student, Person))
```

The evaluation of this precondition returns as answer either the result $\{\{\langle ?s, Student\rangle, \langle ?o, Person\rangle\}\}$ if $Student$ is not a subclass of $Person$, or an empty result if $Student$ is already a subclass of $Person$. Note that these are the only two possible results as both variables were already bound. Multiple results can only be obtained when at least one unbound variable exists in the precondition. Consider as example the following change request where the `?o` variable is unbound:

$$\mathbf{R} = \{\langle addSubClassOf, \{\langle ?s, Student\rangle\}\rangle\}$$

In this example, the precondition with bound variables is:

```
NOT <PREVIOUS>(subClassOf(Student, ?o))
```

The evaluation of the precondition would return as result all possible values for `?o` i.e., all Classes that don't have as a direct subclass the Class *Student*.

Postconditions A postcondition is evaluated for each element of the result set of the precondition evaluation. As a consequence of the restrictions imposed on the Change Definition Language (see Section 5.2.2), a postcondition (transformed into NNF) must be of the following form:

```
expression AND ... AND expression
```

An expression of a postcondition is of the form *Class*`(?x)`, `NOT` *Class* `(?x)`, *Property*`(?x, ?y)` or `NOT` *Property*`(?x, ?y)` where *Class* and *Property* are respectively Classes and Properties of the OWL meta-schema. The order in which the expressions are evaluated is as follows: (1) *Class*`(?x)` expressions, (2) *Property*`(?x, ?y)` expressions, (3) `NOT` *Property*`(?x, ?y)` expressions, and (4) `NOT` *Class*`(?x)` expressions. Note that for each concept bound to a variable that serves as subject of an expression[4], only one new version can be added to the version log during the evaluation of a base change definition.

Before we discuss the rules to evaluate the different types of expressions, we introduce two definitions that aid us in specifying the evaluation rules. We first define when we consider a concept version to be the current concept version for a given concept name. Subsequently, we define when we consider a concept version to be closed i.e., a concept version is closed if the end time point is not equal to the current time.

Definition 5.3 (Current Concept Version). *A concept version v is defined to be the current concept version for a given Concept with name $\sigma \in \boldsymbol{N}$, notation currentConceptVersion(v, σ), iff $v = \langle \sigma, \boldsymbol{A}, \boldsymbol{D}, s, t_s, t_e \rangle \wedge t_e = now$.*

The definition of a closed concept version is defined as follows:

Definition 5.4 (Closed Concept Version). *A concept version v is defined to be a closed concept version, notation closedConceptVersion(v), iff $v = \langle \sigma, \boldsymbol{A}, \boldsymbol{D}, s, t_s, t_e \rangle \wedge t_e < now$.*

The rules to evaluate each expression are given as follows:

- ***Class*`(?x)`:** the outcome of this expression is different depending on whether `?x` is bound to a value or not. When `?x` is bound, a new concept of type *Class* is added to the version log with in its initial concept version the ID set equal to the value of `?x`. On the other hand, when `?x` is not bound to a value, a new anonymous concept of type *Class* is added to the version log.

  If `?x` is bound, assume $\sigma \in \mathbf{N}$ to be the the value of `?x`. If `?x` is unbound, assume σ to be the empty string. The evaluation of the

[4] `?x` is noted as subject of an expression when `?x` is used as first parameter in the expression.

expression results into a new concept evolution $\mathbf{E}_\sigma$ containing an initial concept version υ so that

$$\upsilon = \langle \sigma, \{\}, \{\}, \text{'pending'}, now, now \rangle$$

In the case that we are evaluating deduced changes, the newly created version υ is a deduced version of a version $\upsilon_p = \langle \rho, \mathbf{A}, \mathbf{D}, s, t_s, t_e \rangle$. To reflect this in the version log, we state $D \cup \{\upsilon\}$.

Consider as an example the following expressions. The expression `Class(Person)` (where `?x` is bound to *Person*) results into the creation of a new Class with ID equal to 'Person', while the expression `Class(?x)` (with unbound variable `?x`) results into the creation of an anonymous Class.

- `NOT` ***Class*****`(?x):`** the outcome of this expression is the deletion of the given concept. Therefore, this type of expression requires `?x` to be bound to a value. The latest concept version of the evolution concept of type *Class* with ID equal to the value of `?x` is closed i.e., the end time point is set to the current time.

  Assume $\sigma \in \mathbf{N}$ to be the value of `?x`. If there exists a concept version υ_c of σ so that $currentVersion(\upsilon_c, \sigma)$, we close this version by setting the end time point of the version equal to $now - 1$ so that $closedConceptVersion(\upsilon_c)$ holds.

  Consider the following example. The expression `NOT Class(Person)` (where ?x is bound to *Person*) sets the end time point of the latest concept version of the evolution concept of *Person* to $now - 1$.

- ***Property*****`(?x, ?y):`** the outcome of this expression is the creation of a property of type *Property* between the values of `?x` and `?y`. It is therefore required that both `?x` and `?y` are bound to a variable. If no new concept version of the concept bound to `?x` has been added to the version log in the evaluation of this conceptual change definition, a new concept version of the concept bound to `?x`, which is a copy of the previous concept version, is added and this previous concept version is closed. Subsequently, a property of type *Property* with as object the value of `?y` is added to the latest concept version of the concept bound to `?x`.

  Assume $\sigma \in \mathbf{N}$ to be the value of `?x` and a concept version $\upsilon_c = \langle \sigma, \mathbf{A}, \mathbf{D}, s, t_s, t_e \rangle$ so that $currentVersion(\upsilon_c, \sigma)$. When no new concept version of σ has been created in the evaluation of the current conceptual change definition, we close the current version by setting the end time point equal to $now - 1$ so that $closedConceptVersion(\upsilon_c)$

holds. Assuming that an axiom ϕ correctly represents the property to be added, a new version υ is added to $\mathbf{E}_\sigma$ so that

$$\upsilon = \langle \sigma, \mathbf{A}, \mathbf{D}, \text{`pending'}, now, now \rangle \wedge \mathbf{A} \vDash \phi$$

When a new concept version of σ has already been created in the evaluation of the current conceptual change definition, the set of axioms of the current version υ_c is adapted so that $\mathbf{A} \vDash \phi$.

Consider as example the following expression. The expression `domain-(name, Person)` adds to the latest concept version of the evolution concept of Property *name* the OWL property *domain* with as object *Person*.

- `NOT` ***Property*****`(?x, ?y):`** the outcome of this expression is the deletion of a property of type *Property* between the values of `?x` and `?y`. It is therefore required that both `?x` and `?y` are bound to a value. If no new concept version of the concept bounded to `?x` has been added to the version log in the evaluation of this conceptual change definition, a new concept version of the concept bound to `?x`, which is a copy of the previous concept version, is added and this previous concept version is closed. Subsequently, a property of type *Property* with as object the value bound to `?y` is deleted from the latest concept version of the concept bound to `?x`.

  Assume $\sigma \in \mathbf{N}$ to be the value of `?x` and a concept version $\upsilon_c = \langle \sigma, \mathbf{A}, \mathbf{D}, s, t_s, t_e \rangle$ so that $currentVersion(\upsilon_c, \sigma)$. When no new concept version of σ have been created in the evaluation of the current conceptual change definition, we close the current version by setting the end time point equal to $now - 1$ so that $closedConceptVersion(\upsilon_c)$ holds. Assuming that an axiom ϕ correctly represents the property to be deleted, a new version υ is added to $\mathbf{E}_\sigma$ so that

  $$\upsilon = \langle \sigma, \mathbf{A}, \mathbf{D}, \text{`pending'}, now, now \rangle \wedge \mathbf{A} \nvDash \phi$$

  When a new concept version of σ has already been created in the evaluation of the current conceptual change definition, the set of axioms of the current version υ_c is adapted so that $\mathbf{A} \nvDash \phi$.

  Consider as example the following expression. The expression `NOT domain(name, Person)` removes from the latest concept version of the evolution concept of Property *name* the OWL property *domain* with as object *Person*.

Note that requesting an ontology change with one or more unbound parameters is a powerful feature as it allows to apply a change to all ontology concepts that satisfy the precondition of the base change definitions of the

conceptual change definition. Assume we want to add a Class *Object* as parent of all Classes of our ontology. Instead of requesting a change for each Class, we can achieve the same result with only one requested change. To do so, one can specify a change request $\mathbf{R}$ consisting of one single requested change with an unbound variable ?s that looks as follows:

$$\mathbf{R} = \{\langle addSubClassOf, \{\langle ?o, Object\rangle\}\rangle\}$$

The change request results to the creation of a *subClassOf* Property between all concepts that were previously not a subclass of the Class *Object*. This feature should be handled with care as the following example illustrates. The following change request results in the creation of *subClassOf* Properties between all Classes of the ontology, meaning that all Classes are set to be equal:

$$\mathbf{R} = \{\langle addSubClassOf, \{\}\rangle\}$$

Change Request Implementation

After a change request is evaluated, the requested changes are implemented in the actual ontology when performing the change implementation phase. Implementing the requested change in the actual ontology is straightforward as it only consists of replacing the old ontology with a latest snapshot of the version log. So, the latest version of the ontology O is created as follows (recall from Section 4.2.2 that $\mathbf{S}(t)$ returns a snapshot of an ontology at the given time point t):

$$O = \mathbf{S}(now)$$

5.3 Change Detection

The evaluation of change requests, as described in the previous section, results into updates of the version log and eventually into the ontology itself when performing the change implementation phase. Furthermore, the requested and deduced changes of a change request are added to the evolution log. The purpose of this evolution log is to describe the evolution of an ontology in terms of occurrences of conceptual change definitions. The change detection phase of our framework has as task to further improve the understandability of the evolution of an ontology. The change detection phase therefore enriches the evolution log by detecting additional occurrences of conceptual change definitions not specified in a change request. To be able to detect occurrences of conceptual change definitions, conceptual change definitions are evaluated as temporal queries on the version log. In Section 5.3.1, we discuss the evaluation of conceptual change definitions for the purpose of change detection. As a side effect of the change detection mechanism, we are able to recover from a number of changes that, in retrospect,

turn out to be needlessly applied changes, but were caused by the use of a sequence of changes instead of an equivalent complex change. The recovery of changes is handled by the change recovery phase and is discussed in Section 5.3.2.

5.3.1 Evaluation

As already mentioned, conceptual change definitions are evaluated as temporal queries on a version log in order to detect occurrences of conceptual change definitions. An evolution log can be enriched (or completely constructed in the absence of manually requested changes) with detected occurrences of conceptual change definitions. The evaluation of conceptual change definitions for the purpose of change detection is schematically described by Algorithm 2.

Algorithm 2 Change detection

```
changeDetection(χ):
  while t++ ≤ now do
    for all δ in Δχ do
      for all d ∈ Dδ in δ do
        result = queryVersionLog(d, t)
        for all row in result do
          addToEvolutionLog(row, δ, t)
        end for all
        if result not empty then
          continue
        end if
      end for all
    end for all
  end while
```

The change detection algorithm iterates over all time points (indicated with the variable t in the algorithm) that were added to the version log since the last time the algorithm ran. The first time the change detection algorithm is executed, the variable t is set to 0. For a given change definition set χ, we iterate over all its conceptual change definitions (including imported ones) $\delta \in \Delta_\chi$. For each base change definition $d \in D_\delta$ that is listed to be applicable for change detection in a conceptual change definition δ, we query the version log at the given moment in time t. For each row in the result of the query execution, we add an occurrence of the conceptual change definition δ with the parameters bindings contained in the row and the given moment in time t. As soon as the query execution of one base change definition of a conceptual change definition δ returns a non-empty

result, we omit the querying of the remaining base change definitions of δ. Note that the occurrence of a conceptual change definition is only added to the evolution log if the same occurrence doesn't already exists in the evolution log (when the change was specified in a change request).

In the remainder of this section, we focus both on the flexibility of the change detection approach proposed as well as on the problem of undecidability of the change detection approach that may occur in certain situations.

Flexibility

Notice that the change detection process should be particularly flexible as a single change can in most cases be realized in different ways (unless it is a primitive change). Therefore, the change detection process should not depend on the steps taken to achieve a particular change neither on the order of these steps. It is the responsibility of the change definitions to specify the flexibility allowed. Figure 5.2 illustrates this with two situations for which both the occurrence of the *rangeWeakened* conceptual change definition will be detected. We give a simplified version[5] of the base change definition of *rangeWeakened* as below:

```
rangeWeakened(?p) :=
    // the range of ?p in the past was ?o
    <SOMETIME(?p)>(range(?p, ?o)) AND
    // but changed to ?s
    (NOT range(?p, ?o)) AND range(?p, ?s) AND
    // and ?o is currently a subclass of ?s
    subClassOf*(?o, ?s)
```

In the first situation (A) shown in Figure 5.2, the range of Property p is changed from Class B to Class A being a superclass of B. This change confirms to the change definition of *rangeWeakened* given above. In the second situation (B), the range of Property p is changed from Class B to Class A going from step 1 to step 2. This change doesn't conform to the change definition of *rangeWeakened* as the `subClassOf` condition is not met. However, going from step 2 to step 3, the subclass relation between B and A is added, resulting as yet in the detection of the *rangeWeakened* change as all conditions of the base change definition are now met.

Some ontology engineers may feel that the definition of *rangeWeakened* as given above is too flexible and the second situation of our example shouldn't be considered as an occurrence of the *rangeWeakened* conceptual change definition because the `subClassOf` Property was only added after the range of Property p had already been changed. These ontology engineers can easily overwrite the *rangeWeakened* conceptual change definition

[5] We introduce the complete version of the *rangeWeakened* base change definition in Section 5.6.

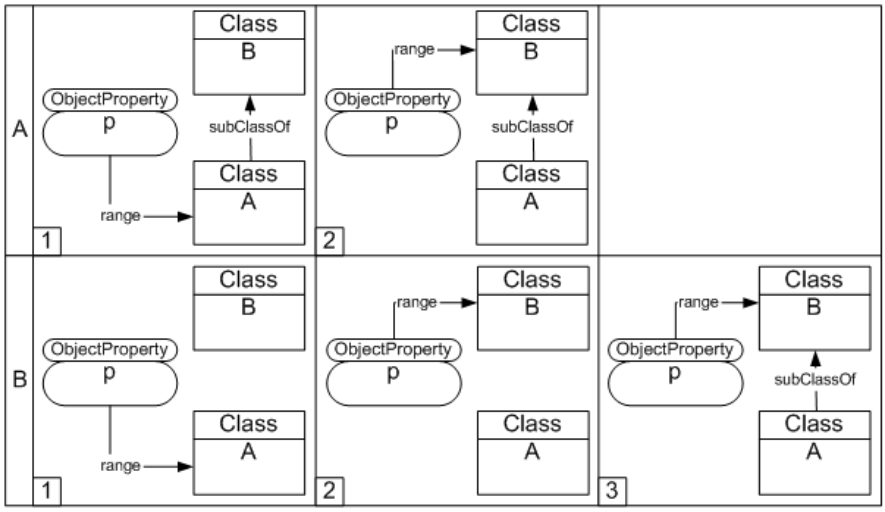

Figure 5.2: Two examples illustrating the flexibility of the change detection process

to require that a `subClassOf` Property must exist before and after the range changes. The base change definition then looks as follows where an additional condition is added to verify that the `subClassOf` Property already existed before:

```
rangeWeakened(?p) :=
    // the range of ?p in the past was ?o
    <SOMETIME(?p)>(range(?p, ?o) AND
    // ?o is already a subclass of ?s
        subClassOf*(?o, ?s)) AND
    // but changed to ?s
    (NOT range(?p, ?o)) AND range(?p, ?s) AND
    // and ?o is currently a subclass of ?s
    subClassOf*(?o, ?s)
```

Undecidability

When detecting changes, undecidability may arise in the sense that at a given moment in time it cannot be determined which conceptual change definition from a number of possibilities did actually occur. The example shown in Figure 5.3 clarifies this statement. In the first step, a Class A is defined as the subclass of both a Class B and a Class C. In the following two successive steps, first the `subClassOf` Property between A and B is deleted, followed by the deletion of the `subClassOf` Property between A and C. From step 3 to step 4, a new `subClassOf` Property is added between A and D. At this moment in time, the change detection process may detect that

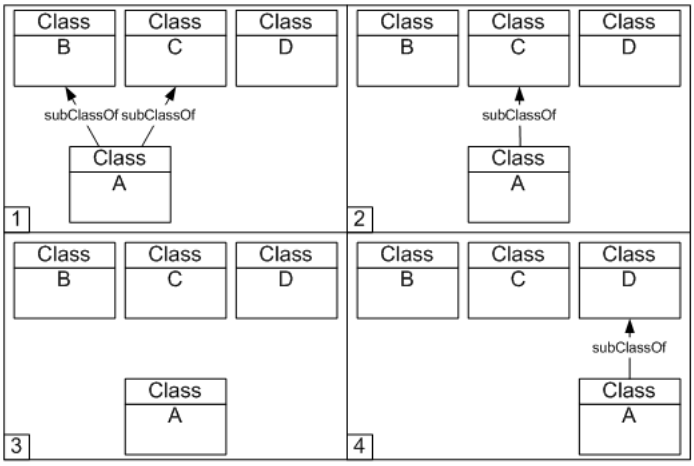

Figure 5.3: Uncertainty in change detection

for a Class A the `subClassOf` Property has changed from B to D, and the `subClassOf` Property has changed from C to D.

The problem we encounter is that in the situation as presented in Figure 5.3, it is impossible that the `subClassOf` Property has changed both from B to D and from C to D at the same moment in time. The only situation in which this can occur is when we are dealing with *transitive changes*, which is not the case in our example. When something changes from x to y and afterwards from y to z, we call the change from x to z a transitive change. Expressed in terms of the *subClassOf* Property, when for a Class A a *subClassOf* Property changes first from B to C and subsequently from C to D, we say that the change of the *subClassOf* Property from B to D is a transitive change. In this case, the change detection approach may detect after the last change two occurrences of a change to the *subClassOf* Property at the same time: one being the change from C to D, the other being the transitive change from B to D. In Section 5.5, we explain how an occurrence of a conceptual change definition can be determined to be a transitive change and the detection of transitive changes can be avoided.

Undecidability can only arise for conceptual change definitions that modify for a subject a certain Property from an old value to a new value. We make use of the parameter annotations `[subject]`, `[old]` and `[new]` in the header of the base change definitions to determine whether undecidability has arisen or not. We are dealing with undecidability when two or more occurrences are detected of the same conceptual change definition, with the same subject, with either the same old or new value and at the same moment in time. Furthermore, none of these occurrences is a transitive change. When undecidability is determined, we don't add an occurrence of the conceptual change definition to the evolution log for each detected occurrence

as we would normally do. Instead we add only one single occurrence of the conceptual change definition for all the detected occurrences involved in the undecidability, but add all the possible parameter bindings to reflect the undecidability in the evolution log. For our example, this means we would add the occurrence of the *changeSubClassOf* conceptual change definition as shown below to the evolution log. All possible parameter bindings are listed in one set indicating that all of them are *correct* parameter bindings, although only one of them reflects the *real* situation. The change occurrence is shown below (note that the time point 23 is randomly chosen):

$$\langle changeSubClassOf,\\ \{\{\langle ?s, A\rangle, \langle ?o1, B\rangle, \langle ?o2, D\rangle\}, \{\langle ?s, A\rangle, \langle ?o1, C\rangle, \langle ?o2, D\rangle\}\},\\ 23\rangle$$

5.3.2 Change Recovery

As discussed in Section 3.2.1, a drawback of using a sequence of changes instead of a dedicated complex change, is possibly an unnecessary loss of data. In our ontology evolution framework, consistency is checked after each step, possibly adding deduced changes to overcome inconsistencies. So when performing a change using a sequence of changes, several deduced changes may have been added that – when considering the sequence of changes as a single complex change – may turn out to be superfluous. However, because our approach allows detecting when a sequence of changes corresponds to a single complex change (see the Change Detection algorithm presented in the previous section), it becomes possible to find unnecessarily deduced changes. In this subsection, we explain our change recovery approach to recover from unnecessary made changes when a complex 'modify'-change is detected.

As explained in Section 5.2.3, deduced changes associated with a requested change in a change request result into deduced concept versions in the version log. In a version log, a deduced concept version v is always specified as part of a concept version $v_{C,n}$ that is part of an evolution concept $\mathbf{E}_C$. This means that the cause of the creation of a deduced concept version is the transition from the version $v_{C,n-1}$ to the concept version $v_{C,n}$ of $\mathbf{E}_C$ where $v_{C,n-1}$ is the directly preceding concept version of $v_{C,n}$. We express this as $cause(v, v_{C,n-1}, v_{C,n})$. Consider as an example the version log introduced in Section 4.1.3 on Page 67. To enhance readability, we repeat the

version log below:

$$\Omega = \langle \mathbf{O}, \{\mathbf{E}_A, \mathbf{E}_B, \mathbf{E}_C\}\rangle$$

$$\begin{aligned}
\mathbf{E}_A = \{&v_{A,0} = \langle \text{`A'}, \{A \sqsubseteq B\}, \{\}, \text{`implemented'}, 0, 0\rangle\,, \\
&v_{A,1} = \langle \text{`A'}, \{A \sqsubseteq B, A \sqsubseteq C\}, \{v_{B,1}\}, \text{`implemented'}, 1, now\rangle\} \\
\mathbf{E}_B = \{&v_{B,0} = \langle \text{`B'}, \{B \sqsubseteq \neg C\}, \{\}, \text{`implemented'}, 0, 0\rangle\,, \\
&v_{B,1} = \langle \text{`B'}, \{\}, \{\}, \text{`implemented'}, 1, now\rangle\} \\
\mathbf{E}_C = \{&v_{C,0} = \langle \text{`C'}, \{\}, \{\}, \text{`implemented'}, 0, now\rangle\}
\end{aligned}$$

In this example, the concept version $v_{B,1}$ is a deduced concept version of $v_{A,1}$. As a consequence, the cause of creation of $v_{B,1}$ is the transition from $v_{A,0}$ to $v_{A,1}$. We therefore say: $cause(v_{B,1}, v_{A,0}, v_{A,1})$. When we would now delete the `subClassOf` Property between A and B, an occurrence of *changeSubClassOf* will be detected for A where a `subClassOf` Property is changed from B to C[6]. After this change, the version log looks as follows:

$$\Omega = \langle \mathbf{O}, \{\mathbf{E}_A, \mathbf{E}_B, \mathbf{E}_C\}\rangle$$

$$\begin{aligned}
\mathbf{E}_A = \{&v_{A,0} = \langle \text{`A'}, \{A \sqsubseteq B\}, \{\}, \text{`implemented'}, 0, 0\rangle\,, \\
&v_{A,1} = \langle \text{`A'}, \{A \sqsubseteq B, A \sqsubseteq C\}, \{v_{B,1}\}, \text{`implemented'}, 1, 1\rangle\,, \\
&v_{A,2} = \langle \text{`A'}, \{A \sqsubseteq C\}, \{\}, \text{`implemented'}, 2, now\rangle\} \\
\mathbf{E}_B = \{&v_{B,0} = \langle \text{`B'}, \{B \sqsubseteq \neg C\}, \{\}, \text{`implemented'}, 0, 0\rangle\,, \\
&v_{B,1} = \langle \text{`B'}, \{\}, \{\}, \text{`implemented'}, 1, now\rangle\} \\
\mathbf{E}_C = \{&v_{C,0} = \langle \text{`C'}, \{\}, \{\}, \text{`implemented'}, 0, now\rangle\}
\end{aligned}$$

As the transition from $v_{A,0}$ to $v_{A,2}$ is detected to be a complex change *changeSubClassOf*, this means that deduced versions caused by one or more concept versions of A between $v_{A,0}$ and $v_{A,2}$ were possibly added needlessly. In our example, it is indeed the case that the disjointness between B and C was deleted needlessly.

To be able to recover from unnecessarily made changes, we first need to determine the deduced concept versions that may be considered for change recovery. To do so, we have to extend the change detection process. Instead of just determining the different ontology concepts that satisfy a particular change definition, we also need to keep track of which versions of these ontology concepts satisfied the change definition. E.g., the ontology change *changeSubClassOf* was detected for the parameters ?s = A, ?o1 = B and ?o2 = C. The versions involved to satisfy the change definition were the following: for A these were $v_{A,0}$ and $v_{A,2}$, for B only $v_{B,0}$, and for C also only $v_{C,0}$. To select the deduced concept versions that possibly may be

[6]Note that the order of changes has no influence on the change detection process. The same complex change will be detected whether we first add `subClassOf(A, C)` and then delete `subClassOf(A, B)`, or first delete `subClassOf(A, B)` and then add `subClassOf(A, C)`.

recovered from, we are only interested in these deduced concept versions that were caused by concept versions of the ontology concept that have changed. We use the `[subject]` parameter annotations in the header of base change definitions to determine the subject of a change. In our example, the Class A is the subject.

For the selected subject, we need to verify if there exists deduced concept versions caused by one of the intermediate concept versions that together form the complex change. The current concept version of A (in our example $v_{A,2}$) may be omitted as it is the end result of the detected complex change. So for our example, we need to check if there exists a v so that $cause(v, v_{A,0}, v_{A,1})$ is true. The only concept version satisfying this condition is $v_{B,1}$.

In the next step, we undo all selected deduced concept versions. Undoing concept versions is straightforward as we can easily return to the previous version using the version log. After the deduced concept versions have been undone, we check if the ontology without these versions remains consistent. Consistency checking is done as described in the consistency maintenance phase (see Section 3.2.1). In our example, this means that we undo the concept version $v_{B,1}$ so that $v_{B,0}$ becomes once more the current concept version for B. When the ontology remains consistent, we successfully recovered from unnecessary made changes. When the ontology becomes inconsistent, the change is impossible to recover and the removed deduced concept version is put back in place.

Important to note is that the permission of the ontology engineer is always required in order to effectively recover a change. The ontology evolution framework merely suggests changes that it can recover to the ontology engineer, but it remains the decision of the ontology engineer whether or not to recover them.

In the remaining sections of this chapter, we present a number of examples of conceptual change definitions. In Section 5.4 and Section 5.5 we discuss respectively primitive and complex change definitions, while in Section 5.6 we discuss complex meta-changes.

5.4 Primitive Change Definitions

In [51], Klein already presented a classification of primitive changes for the OWL ontology language[7]. We take the same classification as a starting point, but some differences exist. E.g., Klein considers 'modify'-changes, which specify that an old value is replaced by a new value, as part of the set of primitive changes. However, all 'modify'-changes can be expressed as a combination of 'delete'- and 'add'-changes, and should therefore not be considered as primitive changes. Klein also identifies different changes

[7]However, no formal definitions for these primitive changes were given.

for adding property restrictions to a Class and adding subclass/equivalence relations. However, from a DL point of view, adding property restrictions is realized by adding either a subclass relation (in the case of necessary conditions) or an equivalence relation (in the case of necessary & sufficient conditions) between a Class and a Restriction. As we desire to offer a complete, but minimal[8] set of primitive changes, we omit the additional changes for adding property restrictions to a Class from the set of primitive changes.

Table 5.1 and Table 5.2 give an overview of our set of primitive changes for OWL. The changes listed form a complete set of primitive changes for OWL DL; the primitive changes for OWL Lite are a subset of this. The first column of the tables shows the header of the base change definition and the second column indicates whether it regards a change for both the Lite and DL variant of OWL ('Lite+DL'), or only the DL variant of OWL ('DL'). 'Add'- and 'delete'-changes are provided for all the different OWL constructs.

We will not provide formal definitions for all primitive changes listed in Table 5.1 and Table 5.2, as the majority of definitions are very similar. Instead, we provide a number of definitions for representative changes and mention the other primitive changes that are defined in a similar way. For the first definition, we give the complete conceptual change definition; while for the other changes we restrict ourselves to the base change definition. For each change, we also give a short informal description. Notice that the body of the base change definitions of all primitive changes is a conjunction of a temporal and non-temporal expression.

The first definition we discuss is the *addClass* conceptual change definition which corresponds to the addition of a new named Class to the ontology. This conceptual change definition and its base change definition are defined as follows:

$$addClass = \langle \{d\}, \{d\}, \{\}, \{\ldots\} \rangle$$

```
addClass(?c) := NOT <PREVIOUS>(Class(?c)) AND Class(?c);
```

The base change definition d is used for both the purpose of change requests and change detection as $d \in \mathbf{R}_\delta \cap \mathbf{C}_\delta$. It specifies that a certain `?c` is currently a Class but was in the previous version not a Class. Note that this definition does not suffice to be used to request the addition of an anonymous Class. The precondition of this definition verifies if `?c` is not yet a Class in order to prevent an ontology from having multiple Classes with the same ID. However, there may exist multiple anonymous Classes i.e., Classes with no ID set, within a single ontology. We therefore introduce the following conceptual change definition to request the addition of anonymous Classes:

[8] Minimal in the sense that removing one of the primitive changes makes it no longer possible to perform all possible changes.

Header	OWL Variant
Class	
addClass(?c)	Lite+DL
addAnonymousClass(?c)	Lite+DLL
deleteClass(?c)	Lite+DL
addSubClassOf(?s, ?o)	Lite+DL
deleteSubClassOf(?s, ?o)	Lite+DL
addEquivalentClass(?s, ?o)	Lite+DL
deleteEquivalentClass(?s, ?o)	Lite+DL
addDisjointWith(?s, ?o)	DL
deleteDisjointWith(?s, ?o)	DL
addToOneOf(?s, ?o)	DL
deleteOneOf(?s)	DL
addToIntersectionOf(?s, ?o)	Lite-DL
deleteIntersectionOf(?s)	Lite-DL
addToUnionOf(?s, ?o)	DL
deleteUnionOf(?s)	DL
addComplementOf(?s, ?o)	DL
deleteComplementOf(?s, ?o)	DL
Restriction	
addAllValuesFromRestriction(?r, ?p, ?o)	Lite+DL
deleteAllValuesFromRestriction(?r)	Lite+DL
addSomeValuesFromRestriction(?r, ?p, ?o)	Lite+DL
deleteSomevaluesFromRestriction(?r)	Lite+DL
addHasValueRestriction(?r, ?p, ?o)	DL
deleteHasValueRestriction(?r)	DL
addMaxCardinalityRestriction(?r, ?p, ?v)	Lite+DL
deleteMaxCardinalityRestriction(?r)	Lite+DL
addMinCardinalityRestriction(?r, ?p, ?v)	Lite+DL
deleteMinCardinalityRestriction(?r)	Lite+DL
addCardinalityRestriction(?r, ?p, ?v)	Lite+DL
deleteCardinalityRestriction(?r)	Lite+DL
Resource	
addLabel(?s, ?v)	Lite+DL
deleteLabel(?s, ?v)	Lite+DL
addComment(?s, ?v)	Lite+DL
deleteComment(?s, ?v)	Lite+DL

Table 5.1: Classification of primitive changes for OWL

Header	**OWL Variant**
Property	
addObjectProperty(?p)	Lite+DL
deleteObjectProperty(?p)	Lite+DL
addDatatypeProperty(?p)	Lite+DL
deleteDatatypeProperty(?p)	Lite+DL
addSubPropertyOf(?s, ?o)	Lite+DL
deleteSubPropertyOf(?s, ?o)	Lite+DL
addDomain(?s, ?o)	Lite+DL
deleteDomain(?s, ?o)	Lite+DL
addRange(?s, ?o)	Lite+DL
deleteRange(?s, ?o)	Lite+DL
addEquivalentProperty(?s, ?o)	Lite+DL
deleteEquivalentProperty(?s, ?o)	Lite+DL
addInverseOf(?s, ?o)	Lite+DL
deleteInverseOf(?s, ?o)	Lite+DL
addFunctional(?p)	Lite+DL
deleteFunctional(?p)	Lite+DL
addInverseFunctional(?p)	Lite+DL
deleteInverseFunctional(?p)	Lite+DL
addTransitive(?p)	Lite+DL
deleteTransitive(?p)	Lite+DL
addSymmetric(?p)	Lite+DL
deleteSymmetric(?p)	Lite+DL
Individual	
addIndividual(?i)	Lite+DL
deleteIndividual(?i)	Lite+DL
addInstanceOf(?i, ?c)	Lite+DL
deleteInstanceOf(?i, ?c)	Lite+DL
addPropertyValue(?p, ?s, ?o)	Lite+DL
deletePropertyValue(?p, ?s, ?o)	Lite+DL
addSameAs(?s, ?o)	Lite+DL
deleteSameAs(?s, ?o)	Lite+DL
addDifferentFrom(?s, ?o)	Lite+DL
deleteDifferentFrom(?s, ?o)	Lite+DL
addToAllDifferent(?s, ?o)	Lite+DL
deleteAllDifferent(?i)	Lite+DL

Table 5.2: Classification of primitive changes for OWL (continued)

- **Name:** addAnonymousClass
  Description: *Represents the change where a new anonymous Class is added the ontology.* This definition differs from the `addClass` definition as there cannot exist more than one named Class with the same ID at the same moment in time within the same ontology, although there can exist numerous anonymous Classes at the same time. Therefore, we do not take the previous version of the ontology into consideration. Note that this change definition is therefore also not suitable to detect the addition of anonymous Classes. However, the change definition of `addClass` suffices to detect the addition of both named and anonymous Classes.
  Definition:
  ```
  addAnonymousClass(?c) := Class(?c)
  ```
  Similarly defined changes: none

The change definitions of the remaining representative primitive changes are enumerated below:

- **Name:** deleteClass(?c)
  Description: *Represents the change where an existing Class is deleted from the ontology.*
  Definition:
  ```
  deleteClass(?c) := <PREVIOUS>(Class(?c)) AND
                     NOT Class(?c);
  ```
  Similarly defined changes: deleteAllValuesFromRestriction, deleteSomeValuesFromRestriction, deleteHasValueRestriction, deleteMaxCardinalityRestriction, deleteMinCardinalityRestriction, deleteCardinalityRestriction, deleteObjectProperty, deleteDatatypeProperty, deleteIndividual

- **Name:** addSubClassOf(?s, ?o)
  Description: *Represents the change where a new subclass relation is added between two Classes or between a Class and a Restriction.*
  Definition:
  ```
  addSubClassOf(?s, ?o) :=
      NOT <PREVIOUS>(subClassOf(?s, ?o)) AND
      subClassOf(?s, ?o);
  ```
  Similarly defined changes: addEquivalentClass, addDisjointWith, addLabel, addComment, addSubProperty, addDomain, addRange, addEquivalentProperty, addInverseOf, addInstanceOf, addSameAs, addDifferentFrom, addComplementOf

- **Name:** deleteSubClassOf(?s, ?o)
  Description: *Represents the change where an existing subclass relation between two Classes or between a Class and a Restriction is*

deleted.
Definition:
```
deleteSubClassOf(?s, ?o) :=
    <PREVIOUS>(subClassOf(?s, ?o)) AND
    NOT subClassOf(?s, ?o);
```
Similarly defined changes: deleteEquivalentClass, deleteDisjointWith, deleteLabel, deleteComment, deleteSubProperty, deleteDomain, deleteRange, deleteEquivalentProperty, deleteInverseOf, deleteInstanceOf, deleteSameAs, deleteDifferentFrom, deleteComplementOf

- **Name:** addToUnionOf(?s, ?o)
  Description: *Represents the change where a Class ?o is added to the 'unionOf' statement of a Class ?s.* Note that the range of a 'unionOf' Property is a list of Class descriptions (?l).
  Definition:
  ```
  addToUnionOf(?s, ?o) :=
      NOT <PREVIOUS>(unionOf(?s, ?l) AND
      member(?l, ?o)) AND
      unionOf(?s, ?l) AND member(?l, ?o);
  ```
  Similarly defined changes: addToOneOf, addToAllDifferentFrom

- **Name:** deleteUnionOf(?s, ?o)
  Description: *Represents the change where a 'unionOf' relation is deleted from a Class.* Note that no primitive change exists to remove a member from the 'unionOf' list, as this change can be realized by first removing the 'unionOf' relation followed by re-adding all its members except the one we desire to delete.
  Definition:
  ```
  deleteUnionOf(?c) := <PREVIOUS>(unionOf(?c, ?l)
                          NOT unionOf(?c, ?l);
  ```
  Similarly defined changes: deleteOneOf, deleteAllDifferentFrom

- **Name:** addAllValuesFromRestriction(?r, ?p, ?o)
  Description: *Represents the change where an 'AllValuesFrom' restriction ?r is added to the ontology for a given Property ?p with as object a Class description or data range ?o.* This primitive change has a similar problem as we encountered with adding named and anonymous Classes. Restrictions are in general added to an ontology as being anonymous (without ID). This means that at the same moment in time, within the same ontology, numerous (anonymous) Restrictions may exist. For the same reason as with *addClass* and *addAnonymousClass*, we define two change definitions for the *addAllValuesFromRestriction*. The first one is aimed to be used in change requests, the second one is used in the change detection phase.

Definition: (change request)

```
addAllValuesFromRestriction(?r, ?p, ?o) :=
    Restriction(?r) AND onProperty(?r, ?p) AND
    allValuesFrom(?r, ?o);
```

Definition: (change detection)

```
addAllValuesFromRestriction(?r, ?p, ?o) :=
    NOT <PREVIOUS>(Restriction(?r) AND
    onProperty(?r, ?p) AND
    allValuesFrom(?r, ?o)) AND
    Restriction(?r) AND onProperty(?r, ?p) AND
    allValuesFrom(?r, ?o);
```

Similarly defined changes: addSomeValuesFromRestriction, addHasValueRestriction, addMaxCardinalityRestriction, addMinCardinalityRestriction, addCardinalityRestriction

- **Name:** addTransitive(?p)
  Description: *Represents the change where the transitive characteristic is added to a Property ?p.*
  Definition:

```
addTransitive(?p) :=
    <PREVIOUS>(isTransitive(?p, "false")) AND
    isTransitive(?p, "true");
```

  Similarly defined changes: addFunctional, addInverseFunctional, addSymmetric

- **Name:** deleteTransitive(?p)
  Description: *Represents the change where the transitive characteristic is removed from a Property ?p.*
  Definition:

```
deleteTransitive(?p) :=
    <PREVIOUS>(isTransitive(?p, "true")) AND
    TransitiveProperty(?p, "false");
```

  Similarly defined changes: deleteFunctional, deleteInverseFunctional, deleteSymmetric

- **Name:** addPropertyValue(?p, ?s, ?o)
  Description: *Represents the change where a property value of a Property ?p is added with subject ?s and object ?o.*
  Definition:

```
addPropertyValue(?p, ?s, ?o) :=
    NOT <PREVIOUS>(PropertyValue(?v) AND
      withPropertyValue(?s, ?v) AND
      ofProperty(?v, ?p) AND
      object(?v, ?o)) AND
    PropertyValue(?v) AND
```

```
withPropertyValue(?s, ?v) AND
ofProperty(?v, ?p) AND
object(?v, ?o);
```

Similarly defined changes: deletePropertyValue

5.5 Complex Change Definitions

The changes we defined in the previous section are called primitive changes as they cannot be realized in terms of other changes. The set of primitive changes consists basically of both 'add'- and 'delete'-changes for the different OWL constructs. Although any ontology modification can be expressed in terms of these primitive changes, restricting the set of changes to only primitive changes would ensue a number of problems:

- **Loss of Semantics:** although a complex change can be realized with a sequence of primitive changes, the semantics of the overall change is lost when opting for primitive changes instead of a complex change. E.g., the complex change `addSiblingClass(B, A)` that adds a new Class B as sibling of A can as well be expressed as a sequence of the primitive changes `addClass(B)` and `addSubClassOf(B, C)` (assuming A is a direct subclass of C). Nevertheless, it is impossible to deduct from the sequence of changes that a sibling Class was added!
- **Loss of Data:** Using a sequence of primitive changes instead of a dedicated complex change may lead to data loss. The ontology is kept consistent after each change in the sequence of changes resulting in the possible appliance of deduced changes. After the complete sequence of changes have been applied, some of these deduced changes may turn out to be superfluous as shown in Section 5.3.2.

As the set of complex changes is non-exhaustive, we present the definitions of a number of representative examples of complex change definitions. Furthermore, we will also illustrate that for a number of complex change definitions it is no longer possible to define a single base change definition that is adequate for both the purpose of change requests and change detection. We also want to stress that there doesn't exist one true definition of a complex change and the definitions given in this section are only one possible interpretation.

5.5.1 Modify-Changes

As discussed in Section 5.4, primitive change definitions only consists of 'add'- and 'delete'-changes. Although a modification can be expressed as the combination of a 'delete'-change followed by an 'add'-change, it is interesting to offer the ontology engineer 'modify'-changes for the different

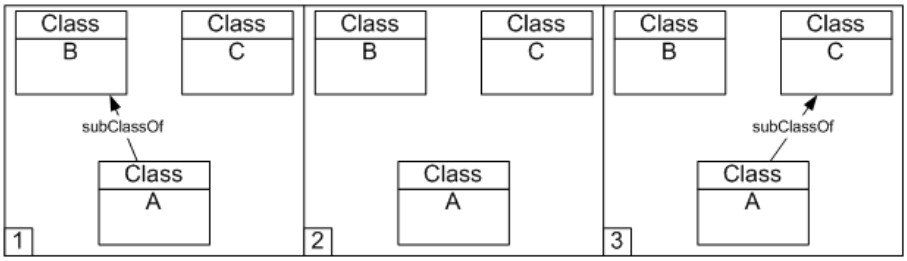

Figure 5.4: Example 1 of a subclass change

OWL constructs. As an example, we consider the conceptual change definition *changeSubClassOf* that represents the change where the object of a subclass property is replaced. We define the base change definition of this conceptual change definition as follows:

```
changeSubClassOf([subject]?s, [old]?o1, [new]?o2) :=
    <PREVIOUS>(subClassOf(?s, ?o1) AND
      (NOT subClassOf(?s, ?o2))) AND
    subClassOf(?s, ?o2) AND (NOT subClassOf(?s, ?o1));
```

The base change definition expresses that in the previous version of the ontology, the subject is a subclass of one object but not of another object, and in the current version it is the other way round. Note that the parameters in the header are annotated with respectively 'subject', 'old' and 'new'. The conceptual change definition is defined as $changeSubClassOf = \langle \{d\}, \{d\}, \{\}, \{\} \rangle$ where d is the base change definition previously given.

The base change definition as it is defined above doesn't leave room for much flexibility when used for the purpose of change detection. Consider as an example the situation depicted in Figure 5.4. The object of subclass Property of a Class A is changed from Class B to Class C in two steps: in the first step (from 1 to 2), the subclass Property between A and B is removed, and in the second step (from 2 to 3), a new subclass Property is added between A and C. Although the two steps combined correspond to a subclass change, it doesn't correspond to the definition given above. The reason is that the definition is too restrictive as it only takes the previous version of the ontology into account.

To overcome this problem we replace the base change definition $d \in \mathbf{C}_\delta$ of the *changeSubClassOf* conceptual change definition with a base change definition that also takes earlier versions of the ontology into account. The new base change definition looks as follows:

```
changeSubClassOf([subject]?s, [old]?o1, [new]?o2) :=
    <SOMETIME(?s)>(subClassOf(?s, ?o1) AND
      (NOT subClassOf(?s, ?o2))) AND
    subClassOf(?s, ?o2) AND (NOT subClassOf(?s, ?o1));
```

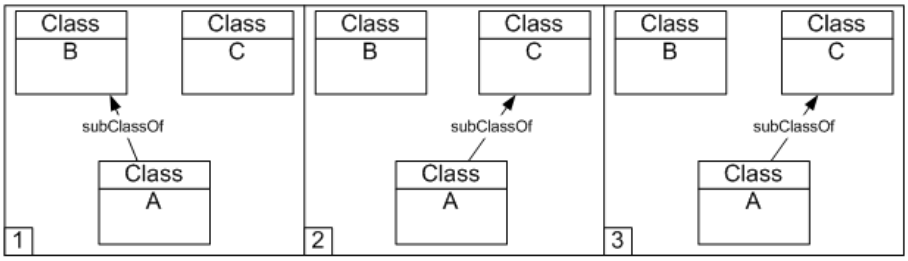

Figure 5.5: Example 2 of a subclass change

The base change definition is defined as a conjunction of a temporal and non-temporal expression where the `SOMETIME` tense operator is used instead of opting for the `PREVIOUS` tense operator. The overall change shown in Figure 5.4 now satisfies the above change definition. However, simply replacing the `PREVIOUS` tense operator with the `SOMETIME` tense operator causes a number of problems. A first problem is illustrated in Figure 5.5. From step 1 to 2, the object of the subclass Property of Class A is changed from B to C; from step 2 to step 3, another change has occurred for A. According to the definition given above, the object of the subclass Property of A has changed from B to C going from step 1 to step 3. Although this is the case, the actual change has occurred already in the previous step. The reason why the change from step 1 to step 3 satisfies the definition is that the definition doesn't take into account the situation in which the subclass already has been changed in the previous version of A. We can solve this by adapting the definition as follows:

```
changeSubClassOf([subject]?s, [old]?o1, [new]?o2) :=
    <SOMETIME(?s)>(subClassOf(?s, ?o1) AND
      (NOT subClassOf(?s, ?o2))) AND
    subClassOf(?s, ?o2) AND (NOT subClassOf(?s, ?o1)) AND
    // check that the postcondition was not already
    // satisfied in the previous version of ?s
    NOT <PREVIOUS(?s)>(subClassOf(?s, ?o2) AND
      (NOT subClassOf(?s, ?o1)));
```

The second problem is illustrated as follows. When the object of a subclass Property of a Class A changes first from B to C, this step corresponds to the change definition given above. When afterwards the object of a subclass Property changes from C to D, not only this step corresponds to the above change definition, but also the overall step from B to D (via C) corresponds to this definition. We call this a *transitive change*. Figure 5.6 visualizes the example. While transitive changes may be appropriate for certain tasks as transitive changes represent the evolution of an ontology concept over a larger period of time, they may not always be desirable. The

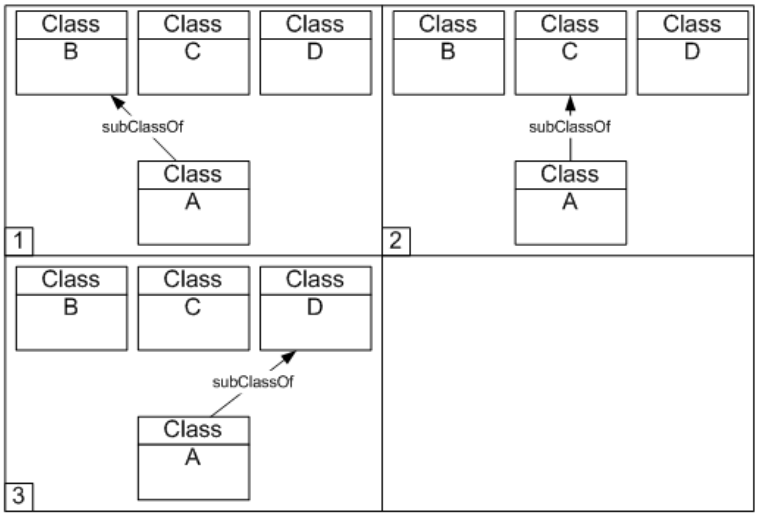

Figure 5.6: Example 3 of a subclass change

change definition can be adapted as follows so that transitive changes are no longer taken into account:

```
changeSubClassOf([subject]?s, [old]?o1, [new]?o2) :=
    <SOMETIME(?s)>(subClassOf(?s, ?o1) AND
      (NOT subClassOf(?s, ?o2))) AND
    subClassOf(?s, ?o2) AND (NOT subClassOf(?s, ?o1))
      AND (NOT eq(?o1, ?o2)) AND

    // check that the postcondition was not already
    // satisfied in the previous version of ?s
    NOT <PREVIOUS(?s)>(subClassOf(?s, ?o2) AND
      (NOT subClassOf(?s, ?o1))) AND

    // remove transitive changes from the result
    // transitive change = ?o1 -> ?x + ?x -> ?o2
    NOT (
     // *** 1. change from ?o1 -> ?x ***
     // situation before change...
      <SOMETIME(?s)>(subClassOf(?s, ?o1) AND
        (NOT subClassOf(?s, ?x)) AND
      // ...situation after change
      <AFTER>(
        subClassOf(?s, ?o1) AND
          (NOT subClassOf(?s, ?x))),
        (NOT subClassOf(?s, ?o1)) AND
          subClassOf(?s, ?x)
```

```
    ) AND

    // *** 2. change from ?x -> ?o2 ***
    // situation before change...
    <SOMETIME(?s)>(subClassOf(?s, ?x) AND
      (NOT subClassOf(?s, ?o2)) AND
    // ...situation after change
    (NOT subClassOf(?s, ?x)) AND subClassOf(?s, ?o2))
  );
```

The additional expressions that were added to the base change definition, assure that transitive changes are no longer taken into account. The definition verifies, when for a certain Class an object of the *subClassOf* Property changes from e.g. B to D, there hasn't been a Class X as object of the *subClassOf* Property for that Class somewhere after the first change.

Similar 'modify'-changes can be defined for the other remaining OWL constructs.

5.5.2 Ambiguity of Changes

Consider the example where we want to change the ID of a Class. We can realize this by applying the following primitive changes: we first delete the Class and its properties from the ontology, followed by the addition of a new Class with the same properties but with a different ID than the deleted one. As deleting and adding a Class isn't a very convenient way to rename resources, we introduce the complex change *changeID*. The base change definition is as follows:

```
changeID([subject]?s, [new]?v) :=
    NOT <PREVIOUS>(hasID(?s, ?v)) AND hasID(?s, ?v);
```

No change definition can be given to detect such a change, as it is impossible to decide whether the deleted and added Class is indeed the same (i.e., it concerns indeed a changed ID) or the deleted and added Class are unrelated.

The same argumentation also holds for the complex changes *changeToObjectProperty* and *changeToDatatypeProperty*. These changes represent the change where a datatype Property, respectively object Property, is transformed into an object Property, respectively datatype Property. Both changes can be realized in terms of primitive changes by first deleting the Property and then adding a new Property of the correct type (either a datatype or an object Property) with the same ID. Although the deleted and newly added Property have the same ID, nothing can guarantee that they represent the same real-world concept as IDs may be reused over time. We present the base change definition of *changeToObjectProperty* below.

The *changeToDatatypeProperty* base change definition can be defined in a similar way.

```
changeToObjectProperty([subject]?p) :=
    <PREVIOUS>(DatatypeProperty(?p)) AND
    ObjectProperty(?p);
```

5.5.3 Property Restrictions

The set of primitive changes contains changes to add restrictions of different kinds (both value and cardinality restrictions) to the ontology. The result of applying such a change is the creation of a single restriction. As restrictions are only meaningful when defined as either a necessary or necessary & sufficient condition of a Class, we may add convenient complex changes for this purpose. For each type of restriction, we would define two conceptual change definitions: one to add the restriction as a necessary and one to add the restriction as necessary & sufficient condition to a Class. As an example, we give the base change definitions of the *addNCardRestriction* conceptual change definition (i.e., add a cardinality restriction as a necessary condition to a Class) and the *addNSCardRestriction* change (i.e., add a cardinality restriction as a necessary & sufficient condition to a Class). The base change definitions of both are given below:

```
addNCardRestriction([subject]?c, [new]?r, ?p, ?v) :=
   NOT <PREVIOUS(?c)>(subClassOf(?c, ?r)
     AND Restriction(?r) AND
     onProperty(?r, ?p) AND cardinality(?r, ?v)) AND
   subClassOf(?c, ?r) AND Restriction(?r) AND
     onProperty(?r, ?p) AND cardinality(?r, ?v);

addNSCardRestriction([subject]?c, [new]?r, ?p, ?v) :=
   NOT <PREVIOUS(?c)>(equivalentClassOf(?c, ?r)
     AND Restriction(?r) AND
     onProperty(?r, ?p) AND cardinality(?r, ?v)) AND
   equivalentClass(?c, ?r) AND Restriction(?r) AND
     onProperty(?r, ?p) AND cardinality(?r, ?o);
```

The necessary conditions are realized by adding a *subClassOf* Property between the Class and the relevant restriction, to express necessary & sufficient conditions, an *equivalentClass* Property is used instead. Note that both base change definitions also suffices for change detection as the addition of a *subClassOf* or *equivalentClass* Property can only occur in one step. We therefore only have to take the previous version of the Class (`<PREVIOUS(?c)>`) into account.

The definition of the conceptual change definition *addNcardRestriction* is defined as $addNcardRestriction = \langle \{d\}, \{d\}, \{\}, \{\} \rangle$ where d is the base change definition shown above. The definition of the conceptual change definition *addNScardRestriction* is analogous.

5.5.4 Sibling Classes

The following conceptual change definition, *addSiblingClass*, represents the change that adds a Class as the sibling of another Class. Note that it doesn't matter whether the superclass of the sibling Class is a Class or a Restriction. As with the previous conceptual change definition, the base change definition given below suffices for both the purpose of change requests and change detection.

```
addSiblingClass([subject]?c, ?s) :=
    NOT <PREVIOUS>(subClassOf(?c, ?x)) AND
    <PREVIOUS>(subClassOf(?s, ?x)) AND
    subClassOf(?c, ?x);
```

5.5.5 Mutual Disjointness

Classes are often defined as being mutually disjoint, meaning that both Classes can have no individuals in common. As the disjointness of Classes is often a possible cause of ontology inconsistencies, it may be interesting to offer the ontology engineer a conceptual change definition to delete the mutual disjointness between two Classes. The conceptual change definition *deleteMutualDisjointness* represents such a change. The base change definition to be used in change requests is defined as follows:

```
deleteMutualDisjointness([subject]?c, [subject]?d) :=
    <PREVIOUS>(disjointWith(?c, ?d) AND
      disjointWith(?d, ?c)) AND
    (NOT disjointWith(?c, ?d)) AND
      (NOT disjointWith(?d, ?c));
```

However, the base change definition shown above doesn't suffice for the purpose of change detection as the change can be realized over a number of steps. We therefore define an additional base change definition as follows:

```
deleteMutualDisjointness([subject]?c, [subject]?d) :=
    <SOMETIME>(disjointWith(?c, ?d) AND
      disjointWith(?d, ?c)) AND
    NOT (disjointWith(?c, ?d)) OR
      disjointWith(?d, ?c)) AND
    <PREVIOUS>(disjointWith(?c, ?d) OR
      disjointWith(?d, ?c));
```

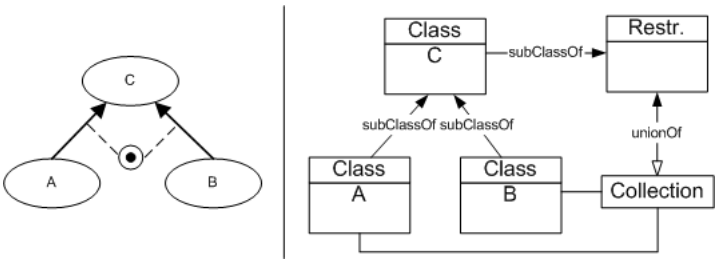

Figure 5.7: Exhaustion constraint in ORM and cover axiom in OWL

Note that, in comparison with the *changeSubClassOf* conceptual change definition, we don't have to take precautions for transitive changes as they cannot occur with a 'delete'-change.

5.5.6 Covering Axioms

In several modeling languages, it is possible to express that when an individual is an instance of a class, it also must be an instance of at least one of the class' children. E.g., the ORM (Object Role Modeling) language [35] has the notion of an *exhaustion constraint* in the form of an ⊙-symbol between two subtype relations. The same constraint can be expressed in OWL by defining the parent Class as the union of its children. In OWL terminology, one refers to the union of children as a *covering axiom*, instead of speaking of an exhaustion constraint. Figure 5.7 shows the ORM and OWL version side by side. In this context, we define a conceptual change definition *addExhaustiveChildren*. This conceptual change definition represents the change where two Classes are added as subclass of a parent Class that on its turn is defined as a cover axiom of the two subclasses. The base change definition shown below suffices for both the purpose of change requests and change detection.

```
addExhaustiveChildren([subject]?c1, [subject]?c2, ?p) :=
    NOT <PREVIOUS(?c1)>(subClassOf(?c1, ?p)) AND
    NOT <PREVIOUS(?c2)>(subClassOf(?c2, ?p)) AND
    NOT <PREVIOUS(?p)>(equivalentClass(?p, ?x) AND
      unionOf(?x, ?l) AND member(?l, ?c1) AND
      member(?l, ?c2)) AND
    subClassOf(?c1, ?p) AND subClassOf(?c2, ?p) AND
      equivalentClass(?p, ?x) AND unionOf(?x, ?l) AND
      member(?l, ?c1) AND member(?l, ?c2);
```

5.5.7 Closure Restriction

As discussed in [77], one of the most frequent mistakes in OWL ontologies is the failure to capture the open world assumption implicit in all OWL expressions. The authors take as example a Class `MargheritaPizza`. The definition of the Class states that a margherita pizza has some tomato and some mozzarella as topping. This is expressed in DL as follows:

$$MargheritaPizza \equiv \exists hasTopping.Tomato \sqcap \exists hasTopping.Mozzarella$$

This definition is incomplete as it allows a margherita pizza to have other toppings then tomato and mozzarella (as long as it has at least one tomato and one mozerella as topping), although a real-world margherita pizza only has tomato and mozzarella as topping and nothing else. The definition can be completed by adding $\forall hasTopping.(Tomato \sqcup Mozzarella)$ to the definition so that it becomes:

$$MargheritaPizza \equiv \exists hasTopping.Tomato \sqcap \exists hasTopping.Mozzarella \sqcap \forall hasTopping.(Tomato \sqcup Mozzarella)$$

The added part to the definition is known as a *closure axiom* because it closes off the possibility of further additions for a given property.

As a possible complex change, we define a conceptual change definition representing the addition of a *closure restriction* to a Class. A closure restriction consists of a 'someValuesFrom'-restriction together with an 'allValuesFrom'-restriction that plays the role of closure axiom. It is impossible to define the conceptual change definition with one base change definition because different cases exist. We explain this by means of a small example. When we want to add a first closure restriction to a Class C as a necessary condition, we can add the following two axioms to the ontology: $\{C \sqsubseteq \exists R.A, C \sqsubseteq \forall R.A\}$. When we want to add for the second time a closure restriction to the same Class, we cannot again add two axioms to the ontology as we did previously. The collection of axioms would be: $\{C \sqsubseteq \exists R.A, C \sqsubseteq \forall R.A, C \sqsubseteq \exists R.B, C \sqsubseteq \forall R.B\}$, requiring that all fillers of R must be instances of both A and B. This is clearly not the intention. The correct operation is to add a 'someValuesFrom'-restriction and adapt the 'allValuesFrom'-restriction so that the collection of axioms looks as follows: $\{C \sqsubseteq \exists R.A, C \sqsubseteq \exists R.B, C \sqsubseteq \forall R.(A \sqcup B)\}$

To handle the different cases correctly, we define three base change definitions. The following three cases exist:

1. *No 'allValuesFrom'-restriction on the relevant Property exists.* The solution is to add a first 'allValuesFrom'-restriction with given value.

2. *An 'allValuesFrom'-restriction on the relevant Property exists where the value of the restriction is not a 'unionOf'-construct.* The solution is to replace the current value with a 'unionOf'-construct containing the current and given value.

3. *An 'allValuesFrom'-restriction on the relevant Property exists where the value of the restriction is a 'unionOf'-construct.* The solution is to extend the 'unionOf'-construct so that it includes the given value.

The preconditions of the base change definitions is used to differentiate between the different cases (see Section 5.2.3 for more details). We show the base change definitions for adding closure restriction as a necessary condition. The change definitions shown below follow the order given by the enumeration list above.

```
addNClosureRestriction([subject]?c, ?p, ?o) :=
    NOT <PREVIOUS(?c)(subClassOf(?c, ?r1) AND
      Restriction(?r1) AND onProperty(?r1, ?p) AND
      someValuesFrom(?r1, ?o) AND subClassOf(?c, ?r2) AND
      Restriction(?r2) AND onProperty(?r2, ?p) AND
      allValuesFrom(?r2, ?o))
    AND
    subClassOf(?c, ?r1) AND Restriction(?r1) AND
      onProperty(?r1, ?p) AND someValuesFrom(?r1, ?o) AND
      subClassOf(?c, ?r2) AND Restriction(?r2) AND
      onProperty(?r2, ?p) AND allValuesFrom(?r2, ?o);

addNClosureRestriction([subject]?c, ?p, ?o) :=
    NOT <PREVIOUS(?c)(subClassOf(?c, ?r1) AND
      Restriction(?r1) AND onProperty(?r1, ?p) AND
      someValuesFrom(?r1, ?o)) AND
    <PREVIOUS(?c)>(subClassOf(?c, ?r2) AND
      Restriction(?r2) AND onProperty(?r2, ?p) AND
      allValuesFrom(?r2, ?x) AND (NOT unionOf(?x, ?l1))
    AND
    subClassOf(?c, ?r1) AND Restriction(?r1) AND
      onProperty(?r1, ?p) AND someValuesFrom(?r1, ?o) AND
      subClassOf(?c, ?r2) AND Restriction(?r2) AND
      onProperty(?r2, ?p) AND allValuesFrom(?r2, ?y) AND
      unionOf(?x, ?l1) AND member(?l1, ?x) AND
      member(?l1, ?o);

addNClosureRestriction([subject]?c, ?p, ?o) :=
    NOT <PREVIOUS(?c)(subClassOf(?c, ?r1) AND
      Restriction(?r1) AND onProperty(?r1, ?p) AND
      someValuesFrom(?r1, ?o)) AND
    <PREVIOUS(?c)>(subClassOf(?c, ?r2) AND
      Restriction(?r2) AND onProperty(?r2, ?p) AND
      allValuesFrom(?r2, ?x) AND unionOf(?x, ?l))
    AND
```

```
subClassOf(?c, ?r1) AND Restriction(?r1) AND
  onProperty(?r1, ?p) AND someValuesFrom(?r1, ?o) AND
  subClassOf(?c, ?r2) AND Restriction(?r2) AND
  onProperty(?r2, ?p) AND allValuesFrom(?r2, ?x) AND
  unionOf(?x, ?l) AND member(?l, ?o);
```

Note that the base change definitions can be used for both the purpose of change requests and change detection. The definition of the conceptual change definition is therefore defined as *addNClosureRestriction*= $\langle \{d1, d2, d3\}, \{d1, d2, d3\}, \{\}, \{\} \rangle$ where $d1$, $d2$ and $d3$ are the base change definitions given above.

5.6 Meta-Change Definitions

In the two previous sections, we have discussed both primitive and complex change definitions. As seen in Section 3.2, we also distinguish meta-change definitions. Meta-change definitions define, in contrast with primitive and complex change definitions not *what* has changed, but rather the *implications* of a change. This has as consequence that meta-change definitions are not useful for the purpose of change requests, and therefore are only used for the purpose of change detection. Consequently, when a conceptual change definition δ is a meta-change definition, the set $\mathbf{R}_\delta$ of the conceptual change definition δ is the empty set i.e., $\mathbf{R}_\delta = \emptyset$.

An example of a meta-change was already given in Section 5.3 to illustrate the flexibility of the change detection approach. It concerned the *rangeWeakened* conceptual change definition for which a simplified definition of its base change definition was given. The conceptual change definition represents the change where the range of Property is weakened i.e., the object of the old range subsumes the object of the new range. We give a more complete base change definition of *rangeWeakened* below:

```
rangeWeakened(?p) :=
    // the range of ?p in the past was ?o
    <SOMETIME(?p)>(range(?p, ?o) AND
    // ? o was already a subclass of ?s
      subClassOf*(?o, ?s)) AND
    // but changed to ?s
    (NOT range(?p, ?o)) AND range(?p, ?s) AND
    // and ?o is currently a subclass of ?s
    subClassOf*(?o, ?s) AND
    // check that the postcondition was not already
    // satisfied in the previous version of ?s
    NOT <PREVIOUS(?p)>(NOT range(?p, ?o) AND
      range(?p, ?s));
```

The base change definition, as defined above, further complements the previously presented definition by ensuring that the postcondition of the definition was not already satisfied in the previous version of the Property. Note that the definition doesn't take transitive changes into account. One can avoid transitive changes from being detected by applying a similar solution as we did for the *changeSubClassOf* ontology change. The *rangeWeakened* conceptual change definition is defined as $\langle\{\}, \{d\}, \{\}, \{\}\rangle$ where d is the base change definition shown above.

Consider as a second example of a meta-change the conceptual change definition *abstractionAdded*. This conceptual change definition represents the change where a level of abstraction is added between either two Classes or two Properties. We define this change by means of two base change definitions shown below. The first base change definition defines the addition of abstraction between two Classes, the second one between two Properties.

```
// abstraction of Classes
AbstractionAdded(?c, ?d) :=
    // ?c [=* ?d
    <PREVIOUS(?c)>(subClassOf*(?c, ?d)) AND
    // ?c [=* ?x and ?x [=* ?d was not the case...
    NOT <PREVIOUS(?c)>(subClassOf*(?c, ?x) AND
      subClassOf*(?x, ?d)) AND
    // ... until now
    subClassOf*(?c, ?x) AND subClassOf*(?x, ?d);

// abstraction of Properties
AbstractionAdded(?c, ?d) :=
    // ?c [=* ?d
    <PREVIOUS(?c)>(subPropertyOf*(?c, ?d)) AND
    // ?c [=* ?x and ?x [=* ?d was not the case...
    NOT <PREVIOUS(?c)>(subPropertyOf*(?c, ?x) AND
      subPropertyOf*(?x, ?d)) AND
    // ... until now
    subPropertyOf*(?c, ?x) AND subPropertyOf*(?x, ?d);
```

As both base change definitions are similar, we only discuss the first one. The definition states that an abstraction was added between `?c` and `?d` whenever `?c` was previously already a subclass of `?d` but there didn't exist a Class `?x` so that `?c` was a subclass of `?x` and `?x` was a subclass of `?d`, although there exists such an `?x` now. The conceptual change definition *AbstractionAdded* consists of both base change definitions and is defined as the tuple $\langle\{\}, \{d_1, d_2\}, \{\}, \{\}\rangle$ where d_1 and d_2 are the base change definitions shown above.

5.7 Summary

In this chapter, we discussed the different purposes of the conceptual change definitions in our ontology evolution approach. In our approach, the conceptual change definitions are used for two different purposes: either to request and apply changes to an ontology or to detect changes that have occurred but that were not explicitly requested. Because we use the Change Definition Language for different purposes, we evaluate change definitions differently depending on the purpose concerned. For the purpose of change requests, we discussed how ontology engineers can specify change requests, what the restrictions are on the constructs of the Change Definition Language in order to be applicable, and how change requests are evaluated. For the purpose of change detection, we also discussed the evaluation of conceptual change definitions in order to detect change occurrences. As a side effect of the change detection mechanism, we discussed the recovering of a number of changes that, in retrospect, turn out to be needlessly applied changes, but were caused by the use of a sequence of changes instead of an equivalent complex change.

In the second part of this chapter, we gave a number of example conceptual change definitions. We provided a complete and minimal set of primitive change definitions for the OWL Web ontology language, and introduced a number of representative examples of both complex change definitions and complex meta-change definitions.

Chapter 6

Conducting Ontology Evolution

In the previous chapter, we explained how both primitive and complex changes as well as complex meta-changes can be defined using the Change Definition Language. For these different types of ontology changes, we gave a number of representative examples together with possible change definitions. We also explained how ontology change definitions can be used by ontology engineers to express a change request, and how a change request is evaluated to lead to the implementation of the requested changes. Finally, we showed how the ontology change definitions are used in the change detection phase to detect additional changes not specified in a change request.

In this chapter, we discuss two facets that conduct the ontology evolution process. The first facet is consistency maintenance; the second facet is backward compatibility.

As ontologies are used to reason about and to infer implicit knowledge from, it is essential for an approach supporting ontology evolution to ensure that ontologies evolve from one consistent state into another consistent state. Therefore maintaining consistency is one of the requirements of our approach. As changes to an ontology may possibly introduce inconsistencies, a method to detect and resolve inconsistencies in the ontology is needed. For OWL DL, several reasoners capable of checking for inconsistencies have been developed (e.g., RACER [62], Fact [37], Pellet [81]). These reasoners are based on the description logic tableau algorithm. While such reasoners allow detecting inconsistencies, determining *why* the ontology is inconsistent and *how* to resolve these inconsistencies is far from trivial. However, (1) pinpointing the axioms that lead to an inconsistent ontology, (2) determining the reasons for the inconsistencies, and (3) using these reasons to offer the ontology engineer suggestions how to resolve these inconsistencies, should be part of an ontology evolution approach.

Moreover, the functionality of an approach supporting ontology evolu-

tion should go beyond conducting the ontology evolution process to assure that consistency is maintained. Ontology engineers and maintainers of depending artifacts are in most cases also interested whether a new version of the ontology remains backward compatible for a given depending artifact i.e., can the old version of the ontology be replaced by the new version without breaking the dependency. The manner in which backward compatibility influences the ontology evolution process is twofold. First, ontology engineers may avoid certain change requests or certain solutions to solve inconsistency knowing that it will break backward compatibility for (some of) their depending artifacts. Secondly, maintainers of depending artifacts may let their decision whether to update to the latest ontology version, an intermediate version, or to not update at all, depend on which (intermediate) version of the ontology is still backward compatible.

The structure of the chapter is as follows. In Section 6.1, we present an approach that determines which axioms are causing an inconsistency. We do this by extending the tableau algorithm most state-of-the-art reasoners rely on by introducing an Axiom Transformation Graph and a Concept Dependency Tree. In Section 6.2, we describe a set of rules that ontology engineers can apply to the axioms involved in the cause of an inconsistency in order to restore consistency. In Section 6.4, we discuss how *compatibility requirements* (i.e., the requirements a version of an ontology should fulfill to be considered backward compatible with its previous version) can be specified and how these can be used to verify whether backward compatibility holds for a given ontology and given depending artifact. Finally, Section 6.5 provides a summary of the chapter.

6.1 Consistency Checking

Checking consistency of an OWL DL ontology can be achieved by running a DL reasoner on the ontology. To achieve this, most state-of-the-art reasoners have adopted the description logic tableau algorithm as mentioned earlier. Although reasoners can be used to identify unsatisfiable concepts, they provide very little information about which axioms are actually causing the inconsistency. This makes it extremely difficult to offer the ontology engineer possible solutions to solve the inconsistency.

In this section, we present an approach to select those axioms of an ontology that are causing an inconsistency. In Section 6.1.1 we discuss the different forms of consistency found in literature and state our focus on logical consistency (see Section 6.1.2). In Section 6.1.3 to Section 6.1.7, we focus on the various aspects of our approach.

6.1.1 Different Forms of Consistency

In literature, three forms of ontology consistency are in general distinguished: *logical consistency*, *structural consistency* and *user-defined consistency* [32]. Current research has mainly focused on techniques to discover and resolve structural- and user-defined inconsistencies, while less attention has been paid to the problem of logical consistency. The difference between these three forms of consistency is as follows:

- **Logical Consistency:** an ontology is considered logically consistent when the ontology conforms to the underlying logical theory of the ontology language. In the case of OWL Lite or OWL DL, this is a variant of description logics. E.g., specifying the range of a Property requires the objects of all instantiations of this Property to be in the range specified. An ontology O (composed of a TBox and ABox) is considered to be consistent if all concepts of the TBox are satisfiable and the ABox is consistent w.r.t. the TBox.

- **Structural Consistency:** an ontology is considered structurally consistent when the constructs provided by the ontology language are correctly used by the ontology. Instead of structural consistency, one often also speaks of the well-formedness of an ontology. Structural consistency can be enforced by checking a set of structural consistency conditions defined for the underlying ontology language. Examples of such structural consistency conditions include: 'the complement of a Class must be a Class', 'a Property can only be a subproperty of a Property', etc. In case of the OWL language, the set of structural conditions depends on the variant of OWL used. E.g., the set of structural conditions for OWL Lite will be more restrictive than these for OWL DL. An ontology is structurally consistent when the ontology meets all structural consistency conditions of the ontology language used.

- **User-defined Consistency:** this form of consistency means that users can add their own, additional conditions that must be met for the ontology to be considered consistent. E.g., users could require that classes can only be defined as a subclass of at most one other class (i.e., preventing multiple inheritance). User-defined consistency conditions can be seen as additional structural consistency conditions to further restrict the language constructs of the underlying ontology language. As user-defined consistency conditions can be written as structural consistency conditions, we won't consider user-defined consistency as a separate form of consistency in the remainder of this section.

An ontology is considered to be consistent when it is both structurally and logically consistent. An ontology must first be structurally consistent

before logical consistency can be verified. In Section 6.1.2, we explain our approach to check logical consistency and resolve logical inconsistencies. To detect and resolve structural inconsistencies, we refer to the work of [32]. The authors define a set of structural consistency conditions and resolution strategies for both the Lite and DL variant of OWL. When an axiom violates a structural consistency condition, the structural inconsistency can be resolved either by removing that axiom or by rewriting that axiom so that it becomes in accordance with the OWL variant chosen. Consider as example (taken from [32]) the following axiom stating that all publications must have at least one author who is not a student:

$$Publication \sqsubseteq \exists author.\neg Student$$

Although this is a valid axiom in OWL DL, it is not in OWL Lite as the latter OWL variant disallows the use of negation. Nevertheless, the structural inconsistency can be resolved by introducing two new axioms: $Student \equiv \exists R.\top$ and $NotStudent \equiv \forall R.\bot$ (where R is a new role name) and rewriting the original axiom to:

$$Publication \sqsubseteq \exists author.NotStudent$$

6.1.2 Logical Consistency

The objective of our approach is to verify whether an ontology remains logically consistent when the requested changes are applied to the ontology. As we are only dealing with logical consistency in our approach, we simply use the term 'consistency' when actually meaning 'logical consistency'. We differentiate between two possible scenarios based on the common distinction found in literature between TBox (terminological or concept knowledge) and ABox (assertional or instance knowledge):

1. **A new axiom was added to the TBox or an existing axiom from the TBox was modified:** to check consistency of the ontology, two tasks are required. First, we verify whether the concepts of the TBox itself are still satisfiable (without considering a possible ABox). We refer to this task as the *TBox Consistency Task*. Secondly, we verify if the ABox remains consistent w.r.t. the modified TBox, called the *ABox Consistency Task*.

2. **A new axiom was added to the ABox or an existing axiom from the ABox was modified:** we verify if the ABox remains consistent w.r.t. its TBox (called *ABox Consistency Task*).

Note that, in the two scenarios described above, we don't take the deletion of an axiom from either the TBox or ABox into account. Because OWL DL is based on a monotonic logic, an ontology can only become inconsistent

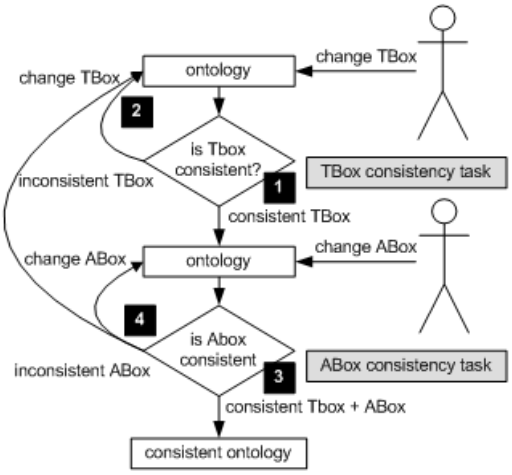

Figure 6.1: Overview of the consistency checking process

when new axioms are added or existing ones are changed i.e., when a set of axioms is consistent, it will still be consistent when deleting any axiom. As a consequence, we only check for ontology consistency in the above mentioned scenarios. An overview of the consistency checking process is shown in Figure 6.1.

The figure reads as follows. When an ontology engineer requests a change to the TBox of an ontology, the TBox Consistency Task is performed first (see 1). When the TBox is determined to be inconsistent, deduced changes are added to the change request to change the TBox to resolve the inconsistency (see 2). Resolving inconsistencies is an iterative process as different inconsistencies are resolved one by one. When the TBox is consistent, the ABox Consistency Task is performed (see 3). Inconsistencies in the ABox can be resolved either by changing particular axioms of the TBox so that the TBox conforms to the changed ABox, or by changing axioms of the ABox so that the changed ABox forms a valid model for the TBox (see 4). When an ontology engineer requests a change that only affects the ABox of an ontology, the TBox Consistency Task is omitted from the consistency checking process.

Inconsistency is resolved by weakening axioms involved in the detected inconsistency so that contradictions among these axioms are solved (see Section 6.2). Note that a weakening of axioms can never introduce new (logical) contradictions between axioms of the ontology, nevertheless it may result into structural inconsistencies. Consider the following small OWL ontology, consisting of three Classes A, B and C, which is obviously inconsistent as A is a subclass of both B and its complement:

```
<owl:Class rdf:ID="A">
  <rdfs:subClassOf>
```

```
      <owl:Class>
        <owl:complementOf rdf:resource="#B"/>
      </owl:Class>
    </rdfs:subClassOf>
    <rdfs:subClassOf rdf:resource="#B"/>
  </owl:Class>
  <owl:Class rdf:ID="B"/>
  <owl:Class rdf:ID="C">
    <rdfs:subClassOf rdf:resource="#A"/>
  </owl:Class>
```

We could solve the contradiction e.g., by completely deleting the Class A from the ontology (although other - and probably better - solutions are feasible). Although this solves the contradiction, it introduces a structural inconsistency as the definition of C now contains references to a non-existing Class A. The most rational solution to solve the incompleteness would be to remove the respective $subClassOf$ Property in the definition of C.

As the state-of-the-art OWL reasoners are based on the DL tableau algorithm, we conclude this subsection with a short overview of the tableau algorithm. The tableau algorithm allows verifying the *satisfiability* of a concept C w.r.t. a given TBox i.e., whether C doesn't denote the empty concept, as well as the *consistency* of a given ABox w.r.t. a TBox i.e., whether the assertions in the ABox form a valid model for the axioms defined in the TBox. An ontology O (composed of a TBox and ABox) is considered to be consistent if all concepts of the TBox are satisfiable and the ABox is consistent w.r.t. the TBox.

The basic principle of the tableau algorithm to check the satisfiability of a concept C is to gradually build a model $\mathcal{I}$ of C i.e., an interpretation $\mathcal{I}$ in which $C^{\mathcal{I}}$ is not empty. The algorithm tries to build a tree-like model of the concept C by decomposing C using tableau expansion rules. These expansion rules correspond to constructors in the description logic. E.g., $C \sqcap D$ is decomposed into C and D, referring to the fact that if $a \in (C \sqcap D)^{\mathcal{I}}$ then $a \in C^{\mathcal{I}}$ and $a \in D^{\mathcal{I}}$. The tableau algorithm ends when either no more rules are applicable or when a clash occurs. A clash is an obvious contradiction and exists in two forms: $C(a) \Leftrightarrow \neg C(a)$ and $(\leq n\ R) \Leftrightarrow (\geq m\ R)$ where $m > n$. A concept C is considered to be satisfiable when no more rules can be applied and no clashes occurred. The tableau algorithm can be straightforwardly extended to support consistency checking of ABoxes. The same set of expansion rules can be applied to the ABox, requiring that we add inequality assertions $a \neq b$ for every pair of distinct individual names.

If we assume C to be a concept, then we use $sub(C)$ to denote the set of subconcepts of the concept C and all concepts D where $C \sqsubseteq^{*} D$. We define a tableau, that is constructed by the tableau algorithm for a concept C, as follows:

Definition 6.1 (Tableau). *A tableau for a concept C is a tuple $\langle \boldsymbol{V}, \boldsymbol{E} \rangle$ where $\boldsymbol{V}$ is the set of nodes and $\boldsymbol{E}$ is the set of edges. Each node $a \in \boldsymbol{V}$ is labeled with a set $L(a) \subseteq sub(C)$ of concepts and each edge $\langle a, b \rangle \in \boldsymbol{E}$ is labeled with a set $L(\langle a, b \rangle)$ of roles (including super roles) occurring in $sub(C)$. We also call $L(a)$ the set of labels of a node a and $L(\langle a, b \rangle)$ the set of labels of an edge $\langle a, b \rangle$.*

Important to note is that, although the tableau algorithm allows us to check ontology consistency, the algorithm doesn't provide us any information regarding the axioms causing the inconsistency, neither does it suggest solutions to overcome the inconsistency. In the remainder of this section and the next section (see Section 6.2), we discuss how we can overcome these shortcomings. We discuss the different aspects of the approach by means of a small example. We consider for our example the following TBox $\mathcal{T}$ consisting of the following axioms:

$$\begin{aligned} PhDStudent &\sqsubseteq \exists enrolledIn.Course \\ \exists enrolledIn.Course &\sqsubseteq Undergraduate \\ Undergraduate &\sqsubseteq \neg PhDStudent \\ PhDStudent\_CS &\sqsubseteq PhDStudent \end{aligned}$$

6.1.3 Axiom Transformations

To increase the performance of the tableau algorithm drastically, most reasoners apply a number of transformations to the axioms of an ontology as a pre-processing optimization step. These axiom transformations are performed before the actual tableau algorithm is ran. We give an overview of the different kind of transformations that occur during the preprocessing step of the tableau algorithm below:

- **Normalization:** The performance of the tableau algorithm can be greatly improved if a contradiction between two concepts can be detected based on the syntactical equivalence between the first concept and the negation of the second concept. To realize this, axioms are transformed into a syntactic normal form. E.g., a contradiction between $C \sqcap D$ and $\neg C \sqcup \neg D$ can be directly detected by transforming the second concept into $\neg(C \sqcap D)$. In [4], a complete set of normalization functions is given. For our example, this means that $PhDStudent \sqsubseteq \exists enrolledIn.Course$ is transformed to $PhDStudent \sqsubseteq \neg\forall enrolledIn.\neg Course$. Other forms of normalization (e.g., negation normal form) can be treated in a similar way.

- **Internalization:** Another task in the preprocessing step concerns the transformation of axioms to support General Concept Inclusion (GCI) of the form $C \sqsubseteq D$ where C and D are complex concepts. In contrast

to subsumption relations between atomic concepts ($A \sqsubseteq B$), which are simply unfolded by the appropriate tableau expansion rule, this is not possible with GCI. To support GCI, $C \sqsubseteq D$ must first be transformed into $\top \sqsubseteq \neg C \sqcup D$ (meaning that any individual must belong to $\neg C \sqcup D$). In our example, $\exists enrolledIn.Course \sqsubseteq Undergraduate$ is transformed to $\top \sqsubseteq Undergraduate \sqcup \forall enrolledIn.\neg Course$.

- **Absorption:** The problem with GCI axioms is that they are time-expensive to reason with due to the high-degree of non-determinism that they introduce [4]. They may degrade the performance of the tableau algorithm to the extent that it becomes in practice non-terminating. The solution of this problem is to eliminate GCI axioms whenever possible. In general, a distinction between *concept absorption* [42] and *role absorption* [87] is made. Concept absorption is a technique that tries to absorb GCI axioms into primitive axiom definitions. E.g., the GCI axiom $A \sqcap \exists R.C \sqsubseteq \exists S.D$ can be transformed and absorbed into the primitive definition of A (also using the normalization functions as described in the first item) so that $A \sqsubseteq \neg\forall S.\neg D \sqcup \neg(\neg\forall R.\neg C)$. Role absorption is only useful for those reasoners directly supporting domain and range axioms. This technique transforms GCI axioms of the form $\top \sqsubseteq \forall R.C$ and $\exists R.C \sqsubseteq \top$ into domain and range axioms respectively. This technique of role absorption can be extended to support a wider range of axioms. An axiom of the form $\exists R.C \sqsubseteq D$ can be absorbed into a range axiom $Range(R, D \sqcup \neg(\neg\forall R.\neg C))$ as the axiom can be rewritten first as $\exists R.\top \sqsubseteq D \sqcup \neg(\neg\forall R.\neg C)$. Similarly, an axiom of the form $D \sqsubseteq \forall R.C$ can be absorbed into a domain axiom $Domain(R, \neg D \sqcup \forall R.C)$. Note that not all reasoners have support for domain and range axioms (e.g., the Fact reasoner has no support for such axioms).

- **Axiom composition:** different axioms can be composed together into one axiom. E.g., the axioms $C \sqsubseteq A$ and $C \sqsubseteq B$ can be transformed to $C \sqsubseteq A \sqcap B$. Axiom composition is often also seen as part of the absorption technique described in the previous item. Assume that in the example used in the previous item, the primitive definition of A was $A \sqsubseteq B$. In this case, the resulting axiom would be $A \sqsubseteq B \sqcap (\exists S.D \sqcup \neg(\neg\forall R.\neg C))$.

- **Axiom decomposition:** an axiom of the form $C \equiv D$ is decomposed into two axioms of the form $C \sqsubseteq D$ and $D \sqsubseteq C$.

We introduce the notion of an *Axiom Transformation Graph* (ATG) to keep track of the transformations that occur during the pre-processing step i.e. an ATG stores the step-by-step transformation of the original axiom (as defined by the ontology engineer) to their transformed form. When later on

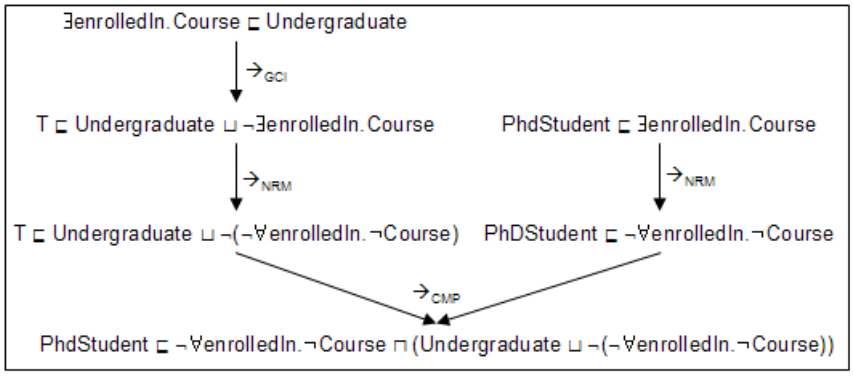

Figure 6.2: An Axiom Transformation Graph (ATG) for the given example

the tableau algorithm ends with a clash, the ATGs can be used to retrieve the original axioms by following the inverse transformations from the axioms causing the clash (as found by the tableau algorithm) to the original ones. We define an Axiom Transformation Graph as follows:

Definition 6.2 (Axiom Transformation Graph). *An Axiom Transformation Graph, notation ATG, is a directed acyclic graph starting from one or more axioms $\phi_1, \ldots, \phi_n$, and ending with a transformed axiom ϕ'. Each branch of the tree represents a transformation and is accordingly labeled as follows:*

- $\rightarrow_{\text{NRM}}$*: indicates a transformation into normal form;*
- $\rightarrow_{\text{GCI}}$*: indicates a transformation of a General Concept Inclusion axiom (GCI);*
- $\rightarrow_{\text{ABS}}$*: indicates an absorption of an axiom into a primitive axiom definition (including both concept and role absorptions);*
- $\rightarrow_{\text{CMP}}$*: indicates a composition of axioms into one single axiom;*
- $\rightarrow_{\text{DCM}}$*: indicates a decomposition of an axiom into two separate axioms.*

Figure 6.2 shows the ATG for the following axioms of our example[1]:

$$\exists enrolledIn.Course \sqsubseteq Undergraduate$$
$$PhDStudent \sqsubseteq \exists enrolledIn.Course$$

[1]Note that we didn't apply the role absorption technique to the $\exists enrolledIn.Course \sqsubseteq Undergraduate$ axiom

6.1.4 Concept Dependencies

The tableau algorithm itself reasons with the transformed axioms resulting from the pre-processing step described in the previous subsection. For our example, the set of transformed axioms is given below:

$$\begin{aligned} PhDStudent &\sqsubseteq \neg\forall enrolledIn.\neg Course \sqcap \\ &\quad (Undergraduate \sqcup \neg(\neg\forall enrolledIn.\neg Course)) \\ Undergraduate &\sqsubseteq \neg PhDStudent \sqcap \\ &\quad (Undergraduate \sqcup \neg(\neg\forall enrolledIn.\neg Course)) \\ PhDStudent\_CS &\sqsubseteq PhDStudent \sqcap \\ &\quad (Undergraduate \sqcup \neg(\neg\forall enrolledIn.\neg Course)) \end{aligned}$$

To test the satisfiability of a concept C, the set of tableau expansion rules are applied to expand this concept until either a clash occurs or no more rules are applicable. As we are interested in the axioms leading to a possible clash (as these axioms are involved in the cause of the unsatisfiability), we need to keep track of all concept axioms (or Class axioms in OWL terminology) used in the expansion of the concept C under investigation. Besides concept axioms, $\mathcal{SHOIN}(\mathbf{D})$ also introduces a number of role axioms (or Property axioms in OWL terminology) including role hierarchy axioms $R \sqsubseteq S$, transitive role axioms $Trans(R)$, inverse role axioms $R \equiv S^-$ and symmetric role axioms $R \equiv R^-$. Besides role axioms, also individual axioms exist including concept assertions $C(a)$, role assertions $R(a, b)$, individual equalities $a = b$ and inequalities $a \neq b$. We therefore also keep track of all role and individual axioms involved in the satisfiable checking of C.

To keep track of the axioms leading to a possible clash, we introduce an extended version of a tableau where each label associated with a node is annotated with the set of labels that were used by the tableau algorithm to add the annotated label to the node. The addition of a label l_1 to a node is caused by the appliance of one of the tableau expansion rules on a label l_2, possibly with the use of role and individual axioms. To express that a label l_1 was added to a node by applying a tableau expansion rule on a label l_2, we denote $causedBy(l_1, l_2, \mathbf{A})$ where $\mathbf{A}$ is the set of role or individual axioms used in the addition of l_1. We define an annotated tableau as follows:

Definition 6.3 (Annotated Tableau). *An annotated tableau for a concept C is a tuple $\langle \boldsymbol{V}, \boldsymbol{E} \rangle$ where $\boldsymbol{V}$ is the set of nodes and $\boldsymbol{E}$ is the set of edges. Each node $a \in \boldsymbol{V}$ is labeled with a set $L(a) \subseteq sub(C)$ of concepts and each edge $\langle a, b \rangle \in \boldsymbol{E}$ is labeled with a set $L(\langle a, b \rangle)$ of roles (including super roles) occurring in $sub(C)$. Each label $l \in L(a)$ of each node $a \in \boldsymbol{V}$ is annotated with a set $Ann(a, l) = \{(l_i, l_j, \boldsymbol{A}) \mid l_i, l_j \in sub(C) \wedge causedBy(l_j, l_i, \boldsymbol{A})\}$. The following conditions hold for an annotated tableau:*

- $\exists l_k \in sub(C).((l_k, l, \boldsymbol{A}) \in Ann(a, l))$

- $\exists l_k \in sub(C).((C, l_k, \boldsymbol{A}) \in Ann(a, l))$
- $\forall l_k \in sub(C).((l, l_k, \boldsymbol{A}) \notin Ann(a, l))$

When there exists $(l_i, l_j, \mathbf{A})$ and $(l_j, l_k, \mathbf{A}')$, both elements of $Ann(a, l)$, we say that l_i and l_k are dependent on each other, and denote this as $dependent(l_i, l_k)$. Note that the dependent relation has a transitive property i.e., if $dependent(l_i, l_k)$ and $dependent(l_k, l_n)$, then also $dependent(l_i, l_n)$.

To list all axioms (including both concept axioms as well as role and individual axioms) involved in the satisfiability checking of a concept C, we introduce the notion of a *Concept Dependency Tree* (CDT). Such a CDT is constructed whenever a clash is found in a node a of a tableau. The construction of a CDT is based on an annotated tableau we introduced in the previous definition. A Concept Dependency Tree itself is defined as follows:

Definition 6.4 (Concept Dependency Tree). *Assume a clash is found in a node a between the concepts D_1 and D_2. A Concept Dependency Tree for a given concept C is defined as an n-ary tree $CDT_C = \langle \boldsymbol{V}, \boldsymbol{E} \rangle$ where $\boldsymbol{V}$ is the set of nodes and $\boldsymbol{E}$ is the set of edges. A node $N \in \boldsymbol{V}$ is a tuple $\langle \phi, \boldsymbol{RIA} \rangle$ where ϕ is a concept axiom and $\boldsymbol{RIA}$ is a set of role and individual axioms. An edge between two nodes is a tuple $\langle N_i, N_j \rangle$ where $N_i, N_j \in \boldsymbol{V}$. The following conditions hold for a CDT:*

- *for each $(l_i, l_j, \boldsymbol{A}) \in Ann(a, D_1) \cup Ann(a, D_2)$ where l_i is an atomic concept, there exists a node $N = \langle \phi, \boldsymbol{RIA} \rangle$ where ϕ is the concept definition of l_i and*

$$\boldsymbol{RIA} = \bigcup_{\forall\ (l_m, l_n, \boldsymbol{A})} \boldsymbol{A} \quad \text{where } dependent(l_i, l_m)$$

- *for each node $N_i = \langle \phi, \boldsymbol{RIA} \rangle$, where ϕ is the concept definition of a label l_i, and node $N_j = \langle \psi, \boldsymbol{RIA'} \rangle$, where ψ is the concept definition of a label l_j, there exists an edge $\langle N_i, N_j \rangle$ iff $dependent(l_i, l_j)$;*

Furthermore, we say that a $child(N_i, N_j)$ relation exists between two nodes iff there exists an edge $\langle N_i, N_j \rangle \in \mathbf{E}$. Furthermore, we define *parent* as the inverse relation of *child*, and *child*$*$ and *parent*$*$ as the transitive counterparts of respectively *child* and *parent*.

The construction of the CDTs occurs during the execution of the tableau algorithm and is based on the use of an annotated tableau. For each label added to a tableau, the annotated tableau keeps track of the path of labels (together with possible role and individual axioms) leading to the addition of that label to the tableau. When a clash is found between two labels, the annotations associated with both labels are used to construct the CDT. For each concept axiom ϕ represented in an annotation, we add a new node N

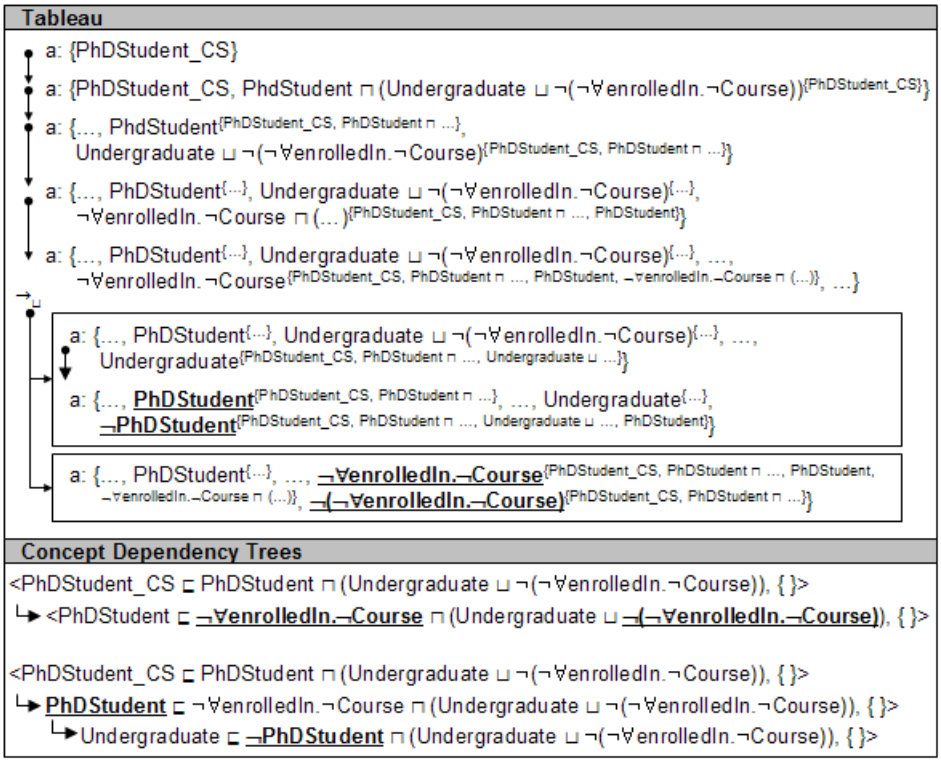

Figure 6.3: Example tableau and associated CDTs

to the CDT (unless such a node already exists) as child of the previous node (if any) so that $N = \langle \phi, \{\} \rangle$. When we encounter a role axiom or individual axiom ψ, we add it to the **RIA** set of the current node N of the CDT so that $\psi \in \mathbf{RIA}$ where $N = \langle \phi, \mathbf{RIA} \rangle$. When a non-deterministic expansion occurs in the tableau algorithm (e.g., as result of a disjunction $A \sqcup B$), this leads to different search paths. We construct copies of the CDT to represent the various search paths of the non-deterministic expansion. All these CDTs are equal until the moment of occurrence of the non-deterministic expansion. Furthermore, note that cyclic axioms (e.g., $C \sqsubseteq \forall R.C$) don't lead to the construction of an infinite CDT, as reasoners normally include some sort of cycle checking mechanism, such as blocking.

The result of the tableau algorithm testing the satisfiability of the concept *PhDStudent_CS* in our example is shown in Figure 6.3 at the top, while the CDTs are shown below[2]. The tableau consists of a single node called a that is gradually filled with labels added by applying a tableau expansion rule. Each appliance of a tableau expansion rule is indicated with

[2]Note that, as it concerns a fairly simple example, no role axioms are included in the example. A more complex example including role axioms will be elaborated in Section 6.3.

a small arrow shown on the left side of the tableau. Note that the appliance of the $\rightarrow_{\sqcup}$ tableau expansion rule results into two branches due to the non-determinism of the rule. Each label in the node of the tableau is annotated with the path of labels leading to the addition of that node (noted in superscript)[3]. The tableau algorithm terminates with the following two clashes:

$$\begin{aligned} PhDStudent &\Leftrightarrow \neg PhDStudent \\ \neg\forall enrolledIn.\neg Course &\Leftrightarrow \neg(\neg\forall enrolledIn.\neg Course) \end{aligned}$$

Note that the non-deterministic expansion in the tableau results into the creation of two CDTs, one for each search path. Both the CDTs contain the different axioms involved in the unsatisfiability of the concept $PhDStudent\_CS$.

Before discussing the interpretation of the Concept Dependency Trees in the following section, we first give the definition of the root node of a CDT.

Definition 6.5 (Root node). *For a given CDT_C, N_r is defined as the root node, notation $rootNode(N_r)$, iff $\neg\exists N_i.(parent(N_i, N_r))$ where N_i is a node of CDT_C.*

6.1.5 Interpretation of Concept Dependencies

In this section, we discuss which axioms listed in the CDTs are causing an ontology inconsistency. The interpretation of which axioms are causing an inconsistency depends on the task performed: the TBox Consistency Task or the ABox Consistency Task.

In the TBox Consistency Task, at least one CDT is constructed for each unsatisfiable concept C in the TBox. More than one CDT may be constructed for a single unsatisfiable concept when non-determinism occurs during the satisfiability checking of that concept. When a concept C is determined to be unsatisfiable, all the CDTs of that concept C contain clashing concepts.

To represent the outcome of the TBox Consistency Task, we introduce the notion of a *Concept Dependency Set for a TBox Consistency Task* (CDS_T). The definition is given as follows:

Definition 6.6 (CDS_T). *Assume $\boldsymbol{N}_{\mathcal{T}}$ to be the set of concept names of all unsatisfiable concepts of a TBox $\mathcal{T}$. A Concept Dependency Set for a TBox Consistency Task, notation CDS_T, is defined as a finite set so that*

$$\forall\sigma \in \boldsymbol{N}_{\mathcal{T}}, \exists! c \in CDS_T.(c = \{CDT_{\sigma,1}, \ldots, CDT_{\sigma,n}\})$$

where $CDT_{\sigma,1}, \ldots, CDT_{\sigma,n}$ are the CDTs of σ resulting from possible different search paths due to non-determinism.

[3]Note that the annotations shown in Figure 6.3 are a simplified representation of the annotations as defined in Definition 6.3 for reasons of clarity.

During the ABox Consistency Task, a reasoner constructs at least one CDT for each concept assertion $C(a)$ in the ABox until a clash is detected. The ABox is determined to be inconsistent when a clash exists between concepts of two different CDTs. Note that it is impossible that one CDT contains both concepts involved in a clash as this would reveal an inconsistent TBox. However, an inconsistent TBox cannot occur at this moment in time as the TBox Consistency Task is performed before the ABox Consistency Task and should have resolved all possible TBox inconsistencies.

The result of the ABox Consistency Task is a set of tuples of two CDTs that represent the clashes detected (the CDTs not representing a clash are omitted). To represent the outcome of the ABox Consistency Task, we introduce the notion of a *Concept Dependency Set for an ABox Consistency Task* (CDS_A). The definition is as follows:

Definition 6.7 (CDS_A). *Assume a clash exists between the concepts C and D. A Concept Dependency Set for an ABox Consistency Task, notation CDS_A, is defined as a finite set*

$$CDS_A = \{\langle CDT_{C,1}, CDT_{D,1}\rangle, \ldots, \langle CDT_{C,n}, CDT_{E,n}\rangle\}$$

where each element of CDS_A lists all the combinations of CDTs that represent a detected clash. Different combinations correspond to the different search paths due to non-determinism in the reasoning process.

In the remainder of this subsection, we discuss the different interpretations of the CDT for the TBox and ABox Consistency Task.

TBox Consistency Task

For a given Concept Dependency Tree CDT_C, all the axioms contained in CDT_C together form the cause of the clash found. are the axioms along the two paths starting from the top node of CDT_C and ending at the node with the axiom containing one of the concepts involved in the detected clash. Removing one of the axioms along the paths, which start from the top node of CDT_C and end at the node with the axiom containing one of the concepts involved in the detected clash, ensures that the particular clash will be avoided. The clash will be avoided because removing an axiom (including concept-, role-, and individual axioms) will interrupt at least one of the paths leading to the clash.

Although removing one of the axioms of a CDT will remove the contradiction represented by the detected clash, we consider it in general bad practice to take all the axioms of the path into consideration to remove the contradiction. We will explain this by means of an example. Assume a TBox populated with the following axioms:

$$\{C \sqsubseteq B, B \sqsubseteq \neg\forall R.\neg D, D \sqsubseteq E, E \sqsubseteq A \sqcap F, F \sqsubseteq \neg A\}$$

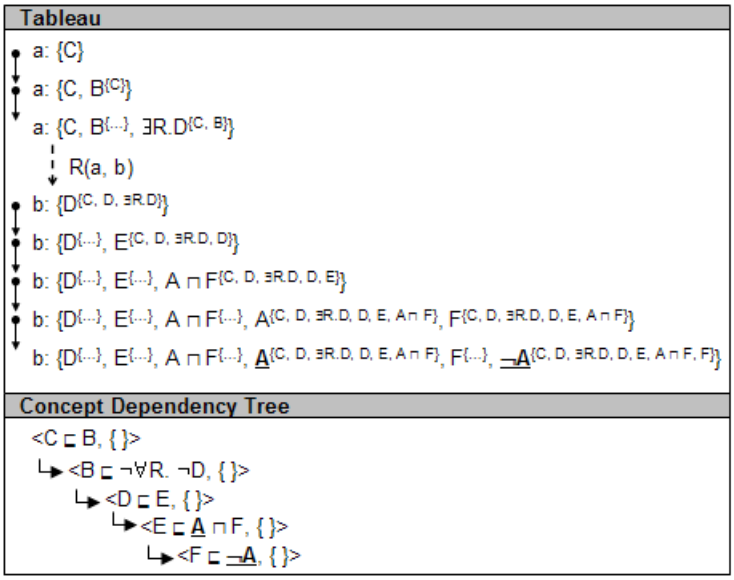

Figure 6.4: Example of a CDT in the TBox Consistency Task

Checking the satisfiability of C will reveal that it is unsatisfiable due to a clash between $A \Leftrightarrow \neg A$. The top part of Figure 6.4 shows the tableau, the bottom part the associated CDT. The tableau shows two nodes a and b, and one edge $\langle a, b \rangle$ (indicated with a dashed arrow). Although removing for example the axiom $C \sqsubseteq B$ will resolve the unsatisfiability of C, this change fails to address the true cause of the unsatisfiability as the overall TBox remains inconsistent (e.g., D is and remains unsatisfiable). The only reason that C is unsatisfiable, is because C depends on a concept that in its turn is also unsatisfiable. In our example, the reason that C is unsatisfiable is because the concept E is unsatisfiable. In other words, the unsatisfiability of C is a *victim* of the unsatisfiability of E.

To decide whether the unsatisfiability of a concept is a direct cause of contradictory concept definitions or simply because the concept depends on another unsatisfiable concept, we take the following procedure. A concept is considered unsatisfiable if a clash is found (in the case of non-deterministic branches, a clash needs to be found in each non-deterministic branch of the tableau). The axioms containing the concepts involved in a clash must have a common parent in the CDT. Otherwise, it would be impossible that a clash was found between these concepts (in our example the concepts A and $\neg A$). Only the first common parent of these axioms and the axioms along the paths from this first common parent to the clashes are directly involved in the unsatisfiability problem. Removing or adapting axioms leading to this common parent (e.g. $C \sqsubseteq B$ or $D \sqsubseteq E$) may resolve the unsatisfiability of the concept under investigation, but this doesn't tackle the true cause

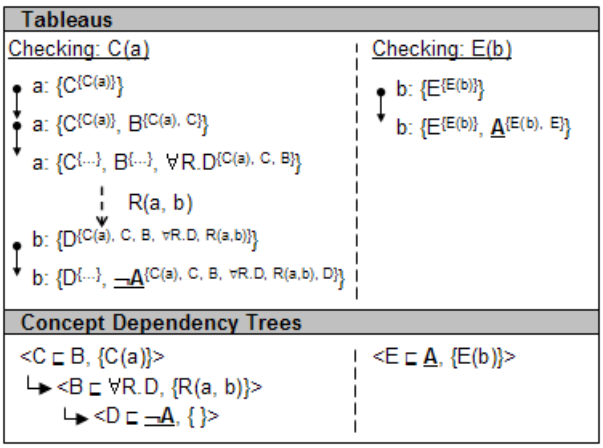

Figure 6.5: Example of CDTs in the ABox Consistency Task

as the actual contradiction remains. We therefore introduce the notion of a *FirstCommonParent* for a CDT, and define it as follows:

Definition 6.8 (FirstCommonParent). *We define N_c as the first common parent of two nodes N_i and N_j, notation $FirstCommonParent(N_c, N_i, N_j)$, iff $parent*(N_c, N_i) \wedge parent*(N_c, N_j) \wedge \neg\exists N_k.(parent*(N_k, N_i) \wedge parent*(N_k, N_j) \wedge child*(N_k, N_c) \wedge N_k \neq N_c)$.*

To select the axioms forming the true cause of the inconsistency, we take the union of all axioms of the *FirstCommonParent* N_c of the nodes containing the concepts involved in the detected clash and all axioms of all the child nodes of N_c. In our example, the node containing the axiom $E \sqsubseteq A \sqcap F$ is the first common parent for the nodes with axioms containing the concepts A and $\neg A$ involved in the clash. We therefore restrict the set of axioms causing the inconsistency to the following set: $\{E \sqsubseteq A \sqcap F, F \sqsubseteq \neg A\}$.

ABox Consistency Task

The interpretation of a CDT for the ABox Consistency Task is different from the interpretation used for the TBox Consistency Task we discussed previously. Consider the following example with TBox: $\{C \sqsubseteq B, B \sqsubseteq \forall R.D, E \sqsubseteq A, D \sqsubseteq \neg A\}$ and ABox: $\{C(a), E(b)\}$. Note that the TBox doesn't contain any unsatisfiable concepts (as the TBox Consistency Task was performed previously and possible contradictions should have been resolved). Adding the assertion $R(a, b)$ to the ABox will result in an inconsistent ABox as a clash occurs between $A(b) \Leftrightarrow \neg A(b)$. Figure 6.5 shows for this example the tableaus at the top (one for each assertion $C(a)$ and $E(b)$) and the associated CDTs at the bottom. The tableaus consists of two nodes a and b, and one edge $\langle a, b\rangle$ (indicated with a dashed arrow).

As shown in the figure, checking ABox consistency for our example results in two CDTs, one for each concept assertion of the ABox (i.e., $C(a)$ and $E(b)$). The individual axioms $C(a)$, $R(a, b)$ and $E(b)$ are added to the CDTs as elements of the **RIA** set of the CDT nodes where appropriate. Note that we only consider individual axioms that were present in the original ABox. Individuals that were added by the tableau algorithm to direct reasoning are not taken into account. Consistency can be restored by interrupting one of the paths leading to the concepts involved in the clash. Therefore, the axioms causing the inconsistency are the axioms, including both concept axioms as well as the axioms of the **RIA** set, along the paths from the top node of the CDTs to the node with the axiom containing the concept involved in the clash. For our example, this results into the following set: $\{C \sqsubseteq B, B \sqsubseteq \forall R.D, E \sqsubseteq A, D \sqsubseteq \neg A, C(a), E(b), R(a, b)\}$.

6.1.6 Axiom Selection

In this section, we give an overview of the overall algorithm to determine the axioms causing an inconsistency based on the interpretations of the CDTs given in the previous section. Note that the axioms that will be considered will differ for the TBox and ABox Consistency Task. We therefore discuss two distinct algorithms, one for each task. Before explaining both algorithms, we fist need to address the following issues:

- **Mark axioms:** It is not necessarily true that a complete concept axiom is in its entirety the cause of an inconsistency. Instead, in most cases, only a part of the axiom will form the true cause of an inconsistency. E.g., for an axiom $C \sqsubseteq A \sqcap \forall R.C \sqcap \ldots \sqcap \neg A$, only the concepts A and $\neg A$ are involved in the unsatisfiability of C, while the remaining concepts (i.e., $\forall R.C$, $\ldots$) in the axiom are not involved. A concept is involved in the cause of an inconsistency either because it is directly involved in the detected clash, or because it is indirectly leading to a concept involved in the detected clash. The algorithm therefore marks the concepts of the axioms involved in the cause of an inconsistency. In order to do so, we assume a `markAllParents(N)` function that marks all concepts in the concept axiom of all the parent nodes of a node N.

- **Non-inconsistency-revealing clashes:** The clashes found between transformed axioms by the tableau algorithm may not always indicate real contradictions between the original axioms as defined by the ontology engineer. Figure 6.3 (see Section 6.1.4) illustrates this. The first detected clash:

$$\neg\forall enrolledIn.\neg Course \Leftrightarrow \neg(\neg\forall enrolledIn.\neg Course)$$

seems to reveal a contradiction at first sight, but when we transform the axioms back to their original form using the reversed transformations of the ATG, it is clear that both concepts involved in the clash are actually the same concept and no contradiction is possible. Consider as example the axiom:

$$PhDStudent \sqsubseteq \underline{\neg\forall enrolledIn.\neg Course} \sqcap (Undergraduate \sqcup \underline{\neg(\neg\forall enrolledIn.\neg Course)})$$

Following the reversed transformations of the ATG, we recover that this axiom originates from the following two axioms as defined by the ontology engineer:

$$\begin{aligned} PhDStudent &\sqsubseteq \underline{\exists enrolledIn.Course} \\ \underline{\exists enrolledIn.Course} &\sqsubseteq Undergraduate \end{aligned}$$

After the transformation to the original axioms, it becomes clear that the two concepts involved in the clash are actually both the same concept i.e., $\exists enrolledIn.Course$. The detected clash guided the tableau algorithm, rather than revealing an actual inconsistency.

As we will see in Section 6.2, a detected inconsistency is resolved by weakening the axioms contained in a CDT in order to solve the contradiction. However, for CDTs representing a non-inconsistency-revealing clash, there is no contradiction to resolve. Nevertheless, the CDT represents a set of axioms that are leading to a clash found in another non-deterministic branch of the tableau (representing a consistency-revealing clash). So, removing one of the axioms of the CDT will resolve the inconsistency as the path to a consistency-revealing clash (represented by another clash) will be broken. In our example, the previously mentioned CDT indicates that $\exists enrolledIn.Course$ was used to expand from $PhDStudent$ to $Undergraduate$. Removing one of the axioms of the CDT resolves the inconsistency.

TBox Consistency Task

To select the axioms causing an inconsistency in the TBox Consistency Task, we don't take all CDTs into account. As described in Section 6.1.5, not all axioms contained in a CDT are always involved in the true cause of the detected inconsistency. Therefore, when the root node of a CDT and the first common parent of the nodes including the concepts involved in the clash found are not the same node, the CDT is not further considered. For the remaining CDTs, we transfer the contained axioms back to their original form using the inverse transformations specified in the ATG.

A more detailed description of the algorithm for the TBox Consistency Task to select the axioms causing an inconsistency is given as follows:

1. For each $CDT_{C,i}$ of a Concept Dependency Set CDS_T, representing a clash $X \Leftrightarrow Y$, lookup the nodes $N_X = \langle \phi_X, \mathbf{RIA} \rangle$ and $N_Y = \langle \phi_Y, \mathbf{RIA}' \rangle$ with respectively a concept axiom ϕ_X containing the concept X involved in the clash and a concept axiom ϕ_Y containing the concept Y involved in the clash. Mark these concepts as being directly involved in the clash.

2. For each $CDT_{C,i}$ of CDS_T, verify whether the root node of $CDT_{C,i}$ is also the first common parent i.e., $rootNode(N_r) \wedge firstCommonParent(N_r, N_X, N_Y)$. If this is not the case, remove $CDT_{C,i}$ from the set CDS_T. According to the interpretation of the CDTs, as discussed in Section 6.1.5, we may remove the CDTs where the root node and first common parent don't fall together because there exists another CDT (where the root node and first common parent do fall together) that narrows the number of axioms responsible for the detected inconsistency further down to those axioms forming the true cause of the inconsistency.

3. For each marked node[4] N, mark all parent nodes using the `markAllParents` function to mark the concepts that are leading to a concept directly involved in the clash.

4. For each $CDT_{C,i}$ of CDS_T, create a set $\mathbf{S}_{C,i}$ listing all axioms (including concept-, role- and individual axioms) contained in the nodes of $CDT_{C,i}$ so that:

$$\mathbf{S}_{C,i} = \bigcup_{\forall\ \langle \phi, \mathbf{RIA} \rangle \in\ CDT_{C,i}} \{\psi\ | \psi = \phi \vee \psi \in \mathbf{RIA}\}$$

5. For each set $\mathbf{S}_{C,i}$, replace all axioms $\psi \in \mathbf{S}_{C,i}$ with their original counterparts by applying the inverse transformations as specified in the ATG.

ABox Consistency Task

For each combination of CDTs found in the ABox Consistency Task, we select all axioms (including concept, role and individual axioms) listed in these two CDTs. We then transform them back to their original form using the transformations specified in the ATG.

A more detailed description of the algorithm for the ABox Consistency Task to select the axioms causing an inconsistency is given as follows:

1. For each tuple of CDTs $\langle CDT_{C,i}, CDT_{D,i} \rangle \in CDS_A$, lookup the nodes $N_X = \langle \phi_X, \mathbf{RIA} \rangle$ in $CDT_{C,i}$ and $N_Y = \langle \phi_Y, \mathbf{RIA}' \rangle$ in $CDT_{D,i}$ with

[4] We consider a node $N = \langle \phi, \mathbf{RIA} \rangle$ to be marked if ϕ contains a marked concept.

respectively a concept axiom ϕ_X containing the concept X involved in the clash and a concept axiom ϕ_Y containing the concept Y involved in the clash. Mark these concepts as being directly involved in the clash.

2. For each marked node N, mark all parent nodes using the `markAllParents` function to mark the concepts that are leading to a concept directly involved in the clash.

3. For each tuple of CDTs $\langle CDT_{C,i}, CDT_{D,i} \rangle$ of a Concept Dependency Set CDS_A, create a set $\mathbf{S}_{C,D,i}$ listing all axioms contained in the nodes of $CDT_{C,i} \cup CDT_{D,i}$ so that:

$$\mathbf{S}_{C,D,i} = \bigcup_{\forall\ \langle \phi, \mathbf{RIA} \rangle \in\ CDT_{C,i} \cup CDT_{D,i}} \{\psi\ | \psi = \phi \vee \psi \in \mathbf{RIA}\}$$

4. For each set $\mathbf{S}_{C,D,i}$, replace all axioms $\psi \in \mathbf{S}_{C,D,i}$ with their original counterparts by applying the inverse transformations as specified in the ATG.

Example

Applying the algorithm of the TBox Consistency Task to our example introduced in Section 6.1.2 results in two sets: $\mathbf{S}_{PhDStudent,1}$ and $\mathbf{S}_{PhDStudent,2}$ (note that the last set represents a non-inconsistency revealing clash). We use the following convention: single-underlined concepts are the concepts marked by the `markAllParents` algorithm, double-underlined concepts are the concepts directly involved in the consistency-revealing clash.

$$\mathbf{S}_{PhDStudent,1} = \{\ \underline{\underline{PhDStudent}} \sqsubseteq \exists enrolledIn.Course,\ \exists enrolledIn.Course \sqsubseteq \underline{Undergraduate},\ Undergraduate \sqsubseteq \underline{\underline{\neg PhDStudent}}\ \}$$

$$\mathbf{S}_{PhDStudent,2} = \{\ PhDStudent \sqsubseteq \underline{\underline{\exists enrolledIn.Course}},\ \underline{\underline{\exists enrolledIn.Course}} \sqsubseteq Undergraduate\ \}$$

6.1.7 Completeness of Axiom Selections

Under some circumstances, the axioms selected by the algorithm for the TBox Consistency Task discussed in the previous section, may result into an incomplete set of axioms. When checking the satisfiability of a concept C, a reasoner stops the reasoning process for that concept either as soon as it encounters a clash, or when no tableau expansion rules are applicable. This means that, in the case where there exists more than one reason for the unsatisfiability of a concept, only the axioms for one reason will be selected

as the reasoner halts after the first clash found. Consider as an example the following set of axioms: $\{C \sqsubseteq A \sqcap B, A \sqsubseteq \neg A, B \sqsubseteq \neg A\}$. It is clear that C is unsatisfiable because of two reasons: first, any instance of C must be an instance of both B and its complement; secondly, any instance of C must be an instance of A and its complement. Nevertheless, the selection algorithm will only select either $\{C \sqsubseteq A \sqcap B, A \sqsubseteq \neg B\}$ or $\{C \sqsubseteq A \sqcap B, B \sqsubseteq \neg A\}$ (depending on the decision made by the reasoner).

An incomplete axiom selection is not dramatically because the remaining reasons for the unsatisfiability of a concept will be dealt with in the next iteration of the ontology consistency checking process. Nevertheless, it may be disturbing for certain users not knowing all reasons for an unsatisfiability at once. The authors of [69] describe an approach to reveal all clashes for an unsatisfiable concept. The approach extends the tableau algorithm with an additional tableau expansion rule, the *clash continue* rule, which allows for the tableau expansion to continue even after a clash has been detected. In this way, the tableau algorithm reveals all possible clashes. The drawback of wanting to reveal all possible clashes is the negative influence it has on the performance of the tableau algorithm.

6.2 Inconsistency Resolving

The reason for an inconsistent ontology is that the overall set of axioms of the ontology is too restrictive in the sense that axioms are contradicting each other. A straightforward solution to overcome an inconsistency would be to simply remove one on the axioms selected by our algorithm discussed in the previous section. As simply removing axioms is often a too drastic solution, we propose a solution of weakening the restrictions imposed by the axioms in order to resolve inconsistencies. In this section, we present a set of rules that ontology engineers can use to weaken the set of axioms in order to overcome the ontology inconsistency detected.

In the previous section (see Section 6.1.5), we have specified two algorithms to determine the set of axioms causing an inconsistency based on the task performed: TBox Consistency Task or ABox Consistency Task:

- The result of the **TBox Consistency Task** is one or more sets $\mathbf{S}_{C,i}$ filled with axioms causing the unsatisfiability of C where i indicates the i^{th} set of axioms for the concept C. More than one set of axioms for the same concept occurs due to non-determinism in the reasoning process. The unsatisfiability of a concept C can be solved by changing an axiom of one of the sets $\mathbf{S}_{C,i}$ applying the set of rules we present in this section.

- The result of the **ABox Consistency Task** is one or more sets $\mathbf{S}_{C,D,i}$ filled with axioms causing the ABox inconsistency where i indicates

the i^{th} set of axioms. As with the TBox Consistency Task, more than one set of axioms may occur due to non-determinism in the reasoning process. The ABox inconsistency detected can be solved by changing an axiom of one of the sets $\mathbf{S}_{C,D,i}$ applying the set of rules we present in this section.

Before introducing the set of rules to solve an ontology inconsistency, we first define a number of auxiliary definitions. We call $\mathcal{H}_c$ the concept hierarchy of all concepts present in the axioms of a given set $\mathbf{S}$[5] and $\mathcal{H}_r$ the role hierarchy of all roles present in the axioms of a given set $\mathbf{S}$. Note that these hierarchies don't include concepts or roles not present in $\mathbf{S}$. We define both hierarchies as follows:

Definition 6.9 (Concept & Role Hierarchy). *For a given set $\boldsymbol{S}$, the concept hierarchy $\mathcal{H}_c$ is defined as a finite set so that $\mathcal{H}_c = \{\langle C, D\rangle \mid C \sqsubseteq D \wedge \neg\exists Z.(C \sqsubseteq Z \wedge Z \sqsubseteq D \wedge Z \neq C \wedge Z \neq D)\}$ where C, D and Z are all concepts marked in the axioms of $\boldsymbol{S}$.*
For a given set $\boldsymbol{S}$, the role hierarchy $\mathcal{H}_r$ is defined as a finite set so that $\mathcal{H}_c = \{\langle R, S\rangle \mid R \sqsubseteq S \wedge \neg\exists T.(R \sqsubseteq T \wedge T \sqsubseteq S \wedge T \neq R \wedge T \neq S)\}$ where R, S and T are all roles used in the axioms of $\boldsymbol{S}$.

We call a concept C_t a top concept of a concept C for a concept hierarchy $\mathcal{H}_c$ and a role R_t a top role of a role R for a role hierarchy $\mathcal{H}_r$ according to the following definition:

Definition 6.10 (Top Concept & Role). *A concept C_t is called a top concept of a concept C for a concept hierarchy $\mathcal{H}_c$, notation $top(C_t, C, \mathcal{H}_c)$, iff $C_t \sqsubseteq C \wedge \neg\exists \langle C_i, C_j\rangle \in \mathcal{H}_c.(C_j = C_t)$.*
A role R_t is called a top role of a role R for a role hierarchy $\mathcal{H}_r$, notation $top(R_t, R, \mathcal{H}_r)$, iff $R_t \sqsubseteq R \wedge \neg\exists \langle R_i, R_j\rangle \in \mathcal{H}_r.(R_j = R_t)$.

In contrary to a top concept and top role, we call a concept C_l a bottom concept of a concept C for a concept hierarchy $\mathcal{H}_c$ and a role R_l a bottom role of a role R for a role hierarchy $\mathcal{H}_r$ according to the following definition:

Definition 6.11 (Bottom Concept & Role). *A concept C_l is defined as bottom concept of a concept C for a concept hierarchy $\mathcal{H}_c$, notation $bottom-(C_l, C, \mathcal{H}_c)$, iff $C \sqsubseteq C_l \wedge \neg\exists(C_i, C_j) \in \mathcal{H}_c.(C_i = C_l)$.*
A role R_l is defined as bottom role of a role R for a role hierarchy $\mathcal{H}_r$, notation $bottom(R_l, R, \mathcal{H}_r)$, iff $R \sqsubseteq R_l \wedge \neg\exists(R_i, R_j) \in \mathcal{H}_r.(R_i = R_l)$.

In the remainder of this section, we present a collection of rules that guides the ontology engineer towards a solution to overcome the inconsistency. A rule either calls another rule or requests a change to an axiom. Note that it remains the responsibility of the ontology engineer to decide which

[5] We simply use $\mathbf{S}$ to refer to either a set $\mathbf{S}_{C,i}$ or a set $\mathbf{S}_{C,D,i}$.

axiom of a set **S** he desires to change. First, we define a set of rules that handle the different types of axioms (i.e. $C \equiv D$, $C \sqsubseteq D$, $R \sqsubseteq S$, $Trans(R)$, $C(a)$, $R(a,b)$, $a = b$ and $a \neq b$). Secondly, we define the necessary rules to weaken or strengthen the different types of concepts.

Note that axioms can always be weakened by removing the axiom. We therefore won't mention this option explicitly in the rules below. The rules for weakening axioms are as follows:

- **Concept Definition Axiom:** A concept definition axiom $C \equiv D$ can be weakened either by strengthening C or weakening D, or by weakening C or strengthening D (depending on the direction in which the axiom was used in the reasoning process leading to the found clash[6]). The first rule listed below corresponds to the direction $C \sqsubseteq D$, the second rule corresponds to the direction $D \sqsubseteq C$:

$$weaken(C \equiv D) \Rightarrow strengthen(C) \vee weaken(D) \tag{6.1}$$

$$weaken(C \equiv D) \Rightarrow weaken(C) \vee strengthen(D) \tag{6.2}$$

- **Concept Inclusion Axiom:** A concept inclusion axiom $C \sqsubseteq D$ can be weakened by strengthening C or weakening D:

$$weaken(C \sqsubseteq D) \Rightarrow strengthen(C) \vee weaken(D) \tag{6.3}$$

- **Role Inclusion Axiom:** A role inclusion axiom $R \sqsubseteq S$ can be weakened by strengthening R or weakening S:

$$weaken(R \sqsubseteq S) \Rightarrow strengthen(R) \vee weaken(S) \tag{6.4}$$

- **Transitive Axiom:** A transitive axiom $Trans(R)$ can only be weakened by removing the transitivity property of R:

$$weaken(Trans(R)) \Rightarrow deleteTransitive(R) \tag{6.5}$$

- **Inverse Axiom:** An inverse axiom $R \equiv S^-$ can only be weakened by removing the inverse axiom:

$$weaken(R \equiv S^-) \Rightarrow deleteInverseOf(R,S) \tag{6.6}$$

- **Symmetric Axiom:** A symmetric axiom $R \equiv R^-$ can only be weakened by removing the symmetric axiom:

$$weaken(R \equiv R^-) \Rightarrow deleteSymmetric(R) \tag{6.7}$$

[6]Because $C \equiv D$ is transformed to $C \sqsubseteq D$ and $D \sqsubseteq C$ in the pre-processing optimization step, the direction in which a concept definition axiom is used can be derived from whether $C \sqsubseteq D$ or $D \sqsubseteq C$ was used in the CDT.

- **Concept Assertion:** A concept assertion $C(a)$ can be weakened by replacing C with a superclass of C:

$$weaken(C(a)) \Rightarrow changeInstanceOf(a, C, D) \tag{6.8}$$

where $\exists C_l.(C_l \sqsubseteq D \wedge leaf(C_l, C, \mathcal{H}_c))$

- **Role Assertion:** A role assertion $R(a, b)$ can be weakened by replacing R with a superproperty of R:

$$weaken(R(a, b)) \Rightarrow changePropertyOfPropertyValue(a, b, R, S) \tag{6.9}$$

where $\exists R_l.(R_l \sqsubseteq S \wedge leaf(R_l, C, \mathcal{H}_c))$

- **Individual Equality & Inequality:** An individual equality $a = b$ and individual inequality $a \neq b$ can only be weakened by removing the axiom:

$$weaken(a = b) \Rightarrow deleteSameAs(a, b) \tag{6.10}$$

$$weaken(a \neq b) \Rightarrow deleteDifferentFrom(a, b) \tag{6.11}$$

The second part of rules deal with the weakening and strengthening of concepts. A concept can be always be weakened by removing the concept. We therefore won't mention this option explicitly in the rules below. When the rules to weaken a concept are similar to the rules to strengthen a concept, we omit these last rules. The rules for weakening and strengthening concepts are defined as follows:

- **Conjunction:** A conjunction $C \sqcap D$ can be weakened (strengthened) by weakening (strengthening) either C or D. The rules for weakening are given below; the rules for strengthening are analogous:

  if $Marked(C)$ **then**

$$weaken(C \sqcap D) \Rightarrow weaken(C) \tag{6.12}$$

  if $Marked(D)$ **then**

$$weaken(C \sqcap D) \Rightarrow weaken(D) \tag{6.13}$$

  if $Marked(C) \wedge Marked(D)$ **then**

$$weaken(C \sqcap D) \Rightarrow weaken(C) \vee weaken(D) \tag{6.14}$$

- **Disjunction:** A disjunction $C \sqcup D$ can be weakened (strengthened) by weakening (strengthening) either C or D. The rules for weakening are given below; the rules for strengthening are analogous:

if $Marked(C)$ **then**

$$weaken(C \sqcup D) \Rightarrow weaken(C) \tag{6.15}$$

if $Marked(D)$ **then**

$$weaken(C \sqcup D) \Rightarrow weaken(D) \tag{6.16}$$

if $Marked(C) \wedge Marked(D)$ **then**

$$weaken(C \sqcup D) \Rightarrow weaken(C) \vee weaken(D) \tag{6.17}$$

- **Existential Quantifier:** An existential quantification $\exists R.C$ can be weakened and strengthened in two manners as it represents both a cardinality restriction ("at least one") and a value restriction. To weaken $\exists R.C$, we either remove $\exists R.C$ if it concerns a cardinality restriction violation, or we weaken C if it concerns a value restriction violation. To strengthen $\exists R.C$, we either add a minimum cardinality restriction if it concerns a cardinality restriction violation, or we strengthen C if it concerns a value restriction violation:

  if $Marked(R)$ **then**

  $$weaken(\exists R.C) \Rightarrow del.SomeValuesFromRestr.(\exists R.C) \tag{6.18}$$

  if $Marked(C)$ **then**

  $$weaken(\exists R.C) \Rightarrow weaken(C) \tag{6.19}$$

  if $Marked(R)$ **then**

  $$strengthen(\exists R.C) \Rightarrow addMinCardinalityRestriction(R, 2) \tag{6.20}$$

  if $Marked(C)$ **then**

  $$strengthen(\exists R.C) \Rightarrow strengthen(C) \tag{6.21}$$

- **Universal Quantifier:** A universal quantification $\forall R.C$ can be weakened (strengthened) by weakening (strengthening) C. The rule for weakening is given below; the rule for strengthening is analogous:

  $$weaken(\forall R.C) \Rightarrow weaken(C) \tag{6.22}$$

- **Maximum Cardinality:** A maximum cardinality restriction $(\leq n\ R)$ can be weakened either by raising n or by removing the cardinality restriction altogether. To strengthen $(\leq n\ R)$, we can lower n:

$$weaken((\leq nR)) \Rightarrow changeCardinality((\leq n\ R), m) \tag{6.23}$$

where $m \geq 1$ if $(\leq n\ R)$ conflicts with $\exists R.C$, or $m \geq \alpha$ if $(\leq n\ R)$ conflicts with $(\geq \alpha\ R)$

$$strengthen((\leq n\ R)) \Rightarrow changeCardinality((\leq n\ R), m) \tag{6.24}$$

where $m = 0$ if $(\leq n\ R)$ conflicts with $\exists R.C$, or $m \leq \alpha$ if $(\leq n\ R)$ conflicts with $(\geq \alpha\ R)$

- **Minimum Cardinality:** A minimum cardinality restriction $(\geq n\ R)$ can be weakened by either lowering n or by removing the cardinality restriction altogether. To strengthen $(\geq n\ R)$, we can raise n:

$$weaken((\geq nR)) \Rightarrow changeCardinality((\geq n\ R), m) \tag{6.25}$$

where $m \leq \alpha$ if $(\geq n\ R)$ conflicts with $(\leq \alpha\ R)$

$$strengthen((\geq n\ R)) \Rightarrow changeCardinality((\leq n\ R), m) \tag{6.26}$$

where $m \geq \alpha$ if $(\geq n\ R)$ conflicts with $(\leq \alpha\ R)$

- **Negation:** A negation $\neg C$ is weakened by either removing $\neg C$ or by strengthening C. To strengthen $\neg C$, we need to weaken C:

$$weaken(\neg C) \Rightarrow strengthen(C) \tag{6.27}$$

$$strengthen(\neg C) \Rightarrow weaken(C) \tag{6.28}$$

- **Atomic Concept:** An atomic concept A is weakened either by removing the concept or by replacing it with a superclass of A. To strengthen an atom concept A, we replace it with a subclass of A. When no (appropriate) sub- or superclass exists, the ontology engineer may create one first:

$$weaken(A) \Rightarrow changeClass(A, C) \tag{6.29}$$

where $\exists C_l.(C_l \sqsubseteq C \wedge leaf(C_l, A, \mathcal{H}_c))$

$$strengthen(A) \Rightarrow changeClass(A, C) \tag{6.30}$$

where $\exists C_t.(C \sqsubseteq C_t \wedge top(C_t, A, \mathcal{H}_c))$

- **Role:** A role R is weakened either by removing the role or by replacing it with a superproperty of R. To strengthen a role R, we replace it with a subproperty of R. When no (appropriate) sub- or superproperty exists, the ontology engineer may create one first:

$$weaken(R) \Rightarrow changeProperty(R, S) \tag{6.31}$$

where $\exists R_l.(R_l \sqsubseteq S \wedge leaf(R_l, R, \mathcal{H}_r))$

$$strengthen(R) \Rightarrow changeProperty(R, S) \tag{6.32}$$

where $\exists R_t.(S \sqsubseteq R_t \wedge top(R_t, R, \mathcal{H}_r))$

Before we apply the set of rules to our example, first consider the following remarks regarding this set of rules:

- In the set of rules, several cases exist where more than one rule can be considered for appliance (e.g., in rule 6.3, a concept inclusion axiom $C \sqsubseteq D$ can be weakened by either weakening C or D) or where the same rule can be applied in different ways (e.g., in rule 6.31, a role R can be weakened by replacing it with a role S that holds to the condition imposed by the rule; in general, several possibilities for S are valid). In these cases, it is the responsibility of the engineer to make a decision best suiting his needs or beliefs.

- Note that in the precondition of rule 6.17, the two concepts C and D of a disjunction $C \sqcup D$ are marked. At first sight however, one would think think this rule will never get triggered as two inconsistency revealing clashes in both search paths of the disjunction would result into two sets **S**: one where C is marked and one where D is marked. The only situation where this may occur is when the disjunction actually represents a conjunction when transformed into NNF i.e., $\neg(\neg C \sqcup \neg D) = C \sqcap D$.

We conclude this section with our example. Remember that the TBox Consistency Task resulted in two sets. Assume that the ontology engineer selects the set **S** containing the following axioms:

$$\begin{aligned}\mathbf{S}_{PhDStudent,1} = \{ \quad & \underline{\underline{PhDStudent}} \sqsubseteq \underline{\exists enrolledIn.Course}, \\ & \underline{\exists enrolledIn.Course} \sqsubseteq \underline{Undergraduate}, \\ & Undergraduate \sqsubseteq \underline{\underline{\neg PhDStudent}} \qquad \}\end{aligned}$$

When for example an ontology engineer beliefs that the axiom:

$$\exists enrolledIn.Course \sqsubseteq Undergraduate$$

doesn't reflect the real word situation as it is too restrictive, he may decide to weaken the axiom by weakening the concept $Undergraduate$. The ontology engineer could change the axiom to $\exists enrolledIn.Course \sqsubseteq Student$, assuming $Undergraduate \sqsubseteq Student$, by following the rules 6.3 and 6.29.

6.3 Example

In this section, we elaborate a more complex example then the one used in the previous sections to illustrate the use of our approach. To check a new version of an ontology on logical consistency, we retrieve the latest version from the version log (using $\mathbf{S}(now)$) and use it as input of a reasoner that has implemented the modified tableau algorithm. The approach will select the axioms that are causing a detected inconsistency (in the occurrence of one) and the set of rules can be used to resolve the inconsistency.

Consider as example the following set of axioms in an OWL DL ontology O where A, B, C, D are concepts and P, R, S are roles:

$$C \sqsubseteq \exists R.A$$
$$C \sqsubseteq \forall R.B \sqcap D$$
$$A \sqsubseteq \exists S.\neg B$$
$$\exists P.\top \sqsubseteq D$$
$$S \sqsubseteq R$$
$$P \sqsubseteq S$$
$$Trans(R)$$
$$Trans(P)$$

Checking logical consistency will reveal a clash between $B \Leftrightarrow \neg B$ when testing the satisfiability of C. After the transformations in the optimization step of the reasoner, the set of axioms looks as follows:

$$C \sqsubseteq \neg\forall R.\neg A \sqcap \forall S.B \sqcap D$$
$$A \sqsubseteq \neg\forall S.B$$
$$domain(P, D)$$
$$S \sqsubseteq R$$
$$P \sqsubseteq S$$
$$Trans(R)$$
$$Trans(P)$$

Figure 6.6 shows the resulting tableau and the associated CDT of the satisfiability testing of C. The tableau consists of three nodes a, b and c, and two edges $\langle a, b\rangle$ and $\langle b, c\rangle$. Each node in the labels of the tableau is annotated with the path leading to the addition of that node (noted in superscript). Note that both the transitive feature of R and the role inclusion axiom $S \sqsubseteq R$ was required to encounter a clash between B and $\neg B$. The resulting CDT consists of four axioms (including the transitive feature $Trans(R)$ and the role inclusion axiom $S \sqsubseteq R$). Applying the algorithm for the TBox Consistency Task, results in the following set $\mathbf{S}_{C,1}$ listing the original axioms causing the unsatisfiability of C:

$$\mathbf{S}_{C,1} = \{C \sqsubseteq \exists R.\underline{A}, C \sqsubseteq \forall R.\underline{\underline{B}} \sqcap D, Trans(R), S \sqsubseteq R, A \sqsubseteq \exists S.\underline{\underline{\neg B}}\}$$

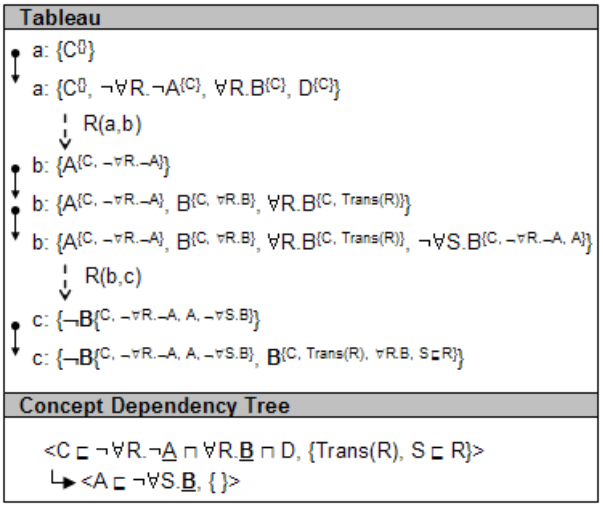

Figure 6.6: An example of logical consistency checking: tableau and associated CDT

Following the set of rules, the inconsistency could be resolved by for instance deleting the role axiom $S \sqsubseteq R$ or $Trans(R)$. Another possible solution could be the weakening of $\neg B$ in the axiom $A \sqsubseteq \exists S.\neg B$ by replacing B with a subclass (due to the negation) following the rules 6.3, 6.19, 6.27 and 6.30. Note that other solutions are certainly possible.

6.4 Backward Compatibility

The approach discussed in the previous sections to resolve ontology inconsistencies is used in the evolution process of an ontology to come from an inconsistent state to a consistent state. As we have seen in Section 6.2, there exists, in general, a number of ways to solve the same inconsistency. It is the responsibility of the ontology engineer to select one of these solutions. The knowledge of which of these solutions breaks the backward compatibility of the ontology for a certain depending artifact mangaged by the same ontology engineer[7] may influence his decision of which solution to take. Furthermore, the knowledge of whether the new version of an ontology is backward compatible with its predecessor or not, may have an influence on the decision of maintainers of depending artifacts to update to the latest version, an intermediate version or to not update at all.

In the context of databases, a database schema is considered backward compatible if all data that was accessible through the old version can still be accessed through the new version of the schema [78]. One refers to this form of backward compatibility as *data preservation*. In the context

[7] An ontology engineer can be a maintainer of a depending artifact at the same time.

of ontologies, the definition of backward compatibility is more complex as depending artifacts don't necessarily rely on instance data only, but may also rely on Classes and Properties of the ontology [51]. This form of backward compatibility is generally called *consequence preservation* and is defined as follows:

Definition 6.12 (Consequence Preservation). *For an ontology O, consequence preservation is defined as a form of backward compatibility where all the facts that could be inferred from the old version of an ontology O_{old} can still be inferred from the new version O_{new}.*

Note that this form of backward compatibility is extremely restrictive when applied to an ontology in its entirety, as a new ontology version is only backward compatible with its predecessor if all the facts that can be inferred from the new version is a superset of all the facts that could be inferred from the old version. This means that a lot of changes will break the backward compatibility of an ontology. In practice however, ontology depending artifacts (mostly) don't rely on all concepts of an ontology, but rather abide by a finite set of ontology concepts [51]. It makes therefore more sense to define backward compatibility for only a set of concepts of an ontology. Which concepts of the ontology it concerns depends on the ontology-depending artifacts in use.

The remainder of this section is structured as follows. In Section 6.4.1, we discuss how maintainers of depending artifacts can express requirements, called *compatibility requirements*, that an ontology should fulfill to be considered backward compatible. These compatibility requirements express which facts that could be inferred from the old version of an ontology must still be inferable from the new version of the ontology. We extend the Change Definition Language we introduced in Section 4.3 to express compatibility requirements. We illustrate this with a few examples. In Section 6.4.2, we explain how the compatibility requirements can be applied to check for backward compatibility of an ontology and how this fits into the overall ontology evolution framework.

6.4.1 Compatibility Requirements

Compatibility requirements allow maintainers of depending artifacts to express which facts that could be inferred from the old version of an ontology must still be inferable from the new version. Backward compatibility can be seen as a stronger version of ontology consistency. A new version of an ontology is backward compatible with an old version for a given depending artifact if the depending artifact remains consistent with the new version and all compatibility requirements are met. Note that backward compatibility of an ontology is always specified for a specific depending artifact. As a consequence, an ontology that is not backward compatible for one depending

artifact may be backward compatible for another depending artifact.

As a compatibility requirement in fact only expresses that a certain axiom that held in the last used version of an ontology must still hold in the new version of that ontology, it can be expressed in terms of the Change Definition Language. For reasons of convenience, we extend the Change Definition Language to include support for the implication operator[8]. We therefore extend the syntax of an expression as shown below where an expression can be an implication.

```
expression        ::=  implication
implication       ::=  term ('->' term)?
term              ::=  factor ('OR' factor)*
factor            ::=  secondary ('AND' secondary)*
secondary         ::=  (primary | 'NOT' primary)
primary           ::=  (statement |
                        parenExpression |
                        tempExp |
                        nativeFunction)
```

We explain the use of the (extended) Change Definition Language to express compatibility requirements by means of a number of examples. Consider as example a depending artifact that relies on the different subclasses of a Class *Individual* of an ontology O. In other words, a new version of the ontology O_{new} is backward compatible for that depending artifact if all the subclasses of *Individual* are a superset of the subclasses of *Individual* from the old version of the ontology O_{old}. We express this compatibility requirement as follows:

```
requirement(?x) :=
    <T(old_version)>(subClassOf*(?x, Individual)) ->
    subClassOf*(?x, Individual);
```

The compatibility requirement expresses that if `?x` was a subclass of *Individual* in the old version of the ontology, it still must be a subclass of *Individual* in the current version. The parameter $old\_version \in \mathbf{T}$ indirectly refers to the old version of the ontology used by the depending artifact (by using the moment in time the version was valid as parameter of the `T` tense operator). The new version of the ontology is backward compatible with the old version if the requirement holds for all concepts of the ontology. Note that this requirement doesn't prohibit that changes are made to the class hierarchy without breaking backward compatibility as is illustrated in Figure 6.7. In the old version of the ontology, there exists two subclasses

[8]The extension isn't an absolute requirement to express compatibility requirements as $\phi \rightarrow \psi$ can be written as $\psi \vee \neg\phi$

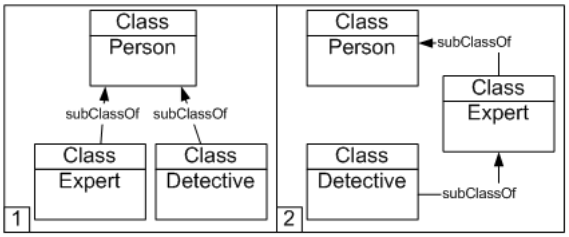

Figure 6.7: First example of a compatibility requirement

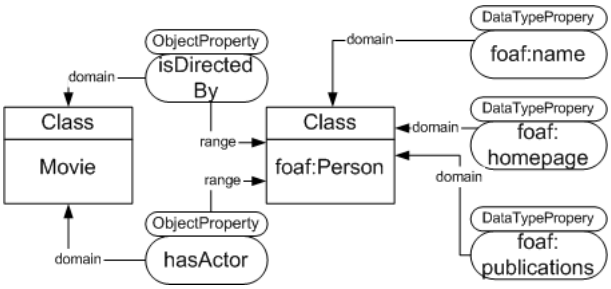

Figure 6.8: Second example of a compatibility requirement

of *Individual* i.e., the Classes *Expert* and *Detective*. In the new version of the ontology, it was decided to treat the Class *Detective* as a subclass of *Expert* as a detective is an expert in investigation. This change does obey the compatibility requirement as all the subclasses of *Individual* in the old version are still subclasses of *Individual* in the new version.

In a second example, the depending artifact is an ontology O that depends on the 'Friend of a Friend' (FOAF) ontology[9]. The ontology O describes concepts and properties of the movie domain (including movies and its actors, director, ...). The ontology uses the *Person* Class of the FOAF ontology to describe the different persons involved in the production of a movie. Figure 6.8 shows a small part of the ontology O. A movie has actors (*hasActor*) and a director (*isDirectedBy*) where both Properties have as domain *Person* of the FOAF ontology. The concept *Person* has a number of Properties among which *name*, *homepage* and *publications*. For the movie ontology, only the Properties *name* and *homepage* associated with *Person* in the FOAF ontology are of interest, while Properties as *publications* are in the movie context completely irrelevant. In other words, as long as *name* and *homepage* are associated with *Person*, the FOAF ontology suits the needs of the movie ontology O.

The maintainer of ontology O may therefore define the following compatibility requirements where the domain of the Properties *name* and *homepage* must still be *Person* in a new version of the FOAF ontology to be considered

[9]See `http://www.foaf-project.org/`

backward compatible for the movie ontology O:

```
requirement1() :=
    <T(old_version)>(domain(foaf:name, foaf:Person)) ->
    domain(foaf:name, foaf:Person);
requirement2() :=
    <T(old_version)>(domain(foaf:homepage, foaf:Person)) ->
    domain(foaf:homepage, foaf:Person);
```

Note that other concepts of the FOAF ontology, besides the *name* and *homepage* Properties, can't break the backward compatibility of the movie ontology O.

The third and last example is a little different from the previous examples as backward compatibility depends on a particular instantiation of ontology concepts. The ontology in this example describes the political situation of several countries (current government, prime minister, president, ...). The dependent artifact is a website providing political news. The website uses the political ontology to annotate the content of the website. An example of such annotation is a group photo of all the ministers of a government on one of the web pages that is annotated with an instantiation of the *Government* Class. A government consists of a number of ministers expressed by the *hasMinister* Property and *Minister* Class. One may argue that the image of the government is a correct representation of the actual government as long as no ministers are replaced. The maintainer of the website could formulate a compatibility requirement as follows (assuming $VHFII$, which stands for 'Verhofstadt II', is an instance of *Government* used to annotate the image):

```
requirement(?m) :=
    <T(old_version)>(hasPropertyValue(VHFII, ?pv) AND
      ofProperty(?pv, hasMinister) AND
      object(?pv, ?m)) ->
    hasPropertyValue(VHFII, ?pv) AND
      ofProperty(?pv, hasMinister) AND
      object(?pv, ?m));
```

6.4.2 Checking Backward Compatibility

We consider an ontology to be backward compatible for a given depending artifact when the depending artifact remains consistent and all its compatibility requirements hold for all its ontology concepts. When one of the compatibility requirements fails for one of the concepts, backward compatibility is no longer guaranteed. E.g., when we consider the compatibility requirement of the last example, all the instances of the Class *Minister* that were member of the 'VHFII'-government in the old version, must still be

instances of the Class *Minister* and must still be member of the same government in the new version.

Remember that checking of backward compatibility is used in two different scenarios in our ontology evolution framework. In the first scenario, the ontology engineer requests a change and is interested whether the requested change and possible deduced changes to restore consistency keep the resulting version of the ontology backward compatible with the old version for the depending artifacts that the ontology engineer manages. In the second scenario, a maintainer of a depending artifact is interested which of the versions (including intermediary versions) coming after the version currently in use, is still backward compatible. In the remainder of this section, we discuss both scenarios.

In the first scenario, the ontology engineer formulates a change request in the Change Request phase that leads to one or more new versions in the version log with status set to 'requested'. In the following phase, the Consistency Maintenance phase, possible deduced changes are added to the change request to restore the ontology inconsistency caused by the requested change. In their turn, the deduced changes also result in one or more new versions in the version log with status set to 'requested'. At the end of the Consistency Maintenance phase, implementing the changes of the change request (including deduced changes) will transform the ontology into another consistent version. At this point, we check whether the changes break backward compatibility of the ontology for the known depending artifacts of the ontology engineer. Based on the outcome of the backward compatibility checking, the ontology engineer may decide to go on with the requested changes or to cancel them. The former means that the status of the added versions is changed from 'requested' to 'confirmed' indicating that these new versions are confirmed for implementation. The latter means that either the ontology engineer undoes the deduced changes and chooses another solution (if one exists) to resolve the inconsistency, or he cancels his change request completely forcing all added versions to be undone.

In the second scenario, a maintainer of a depending artifact of an ontology O is interested in upgrading from an old version of the ontology O to the new version of that ontology. This situation is illustrated in Figure 3.6 on Page 55. One of the things the maintainer of the depending artifact may be interested in, is which of the versions between the old version and the new version (the new version included) is the latest backward compatible version. In other words, what is the latest version of the ontology for which all the compatibility requirements hold? This activity is part of the Cost of Evolution phase in the ontology evolution framework. Checking whether a version is backward compatible involves setting the *now* variable temporarily to the time point of that version before evaluating the compatibility requirements. The end result is a time point $t \in \mathbf{T}$ indicating the time point of the latest backward compatible version. Using this information, the maintainer of a

depending artifact may decide to update to the latest version (which may not be a backward compatible version), the latest backward compatible version or to not update at all. The exact procedure how to update is discussed in the next chapter (see Section 7).

6.5 Summary

In this chapter, we have discussed two facets that conduct the ontology evolution process: ontology consistency and backward compatibility. Concerning ontology consistency, the focus in this dissertation is on logical consistency. An ontology O (composed of a TBox and ABox) is considered to be logically consistent if all concepts of the TBox are satisfiable and the ABox is consistent w.r.t. the TBox. To check logical consistency of an OWL DL ontology, we can rely on existing DL reasoners. However, the disadvantage of current reasoners is that they don't provide any information about the cause of an inconsistency and don't offer any solutions to overcome inconsistencies. To overcome this disadvantage, we have extended the tableau algorithm on which most of the reasoners rely, to reveal the cause of an inconsistency. We keep track of the internal transformations on the axioms by means of an Axiom Transformation Graph (ATG) and pinpoint the axioms that are leading to a clash by means of a Concept Dependency Tree (CDT). Both the ATG and CDT are used to determine a set of axioms forming the cause of a detected inconsistency. Furthermore, we discussed a set of rules that ontology engineers can apply on the axioms that form the cause of an inconsistency, in order to resolve an inconsistency. The idea behind the set of rules is to weaken the axioms so that contradictions are resolved.

The second facet concerns backward compatibility. An ontology version is considered to be backward compatible with a previous version for a given depending artifact if the depending artifact remains consistent and a set of compatibility requirements hold. Compatibility requirements allow maintainers of depending artifacts to express which facts that could be inferred from the old version of an ontology must still be inferable from the new version. All compatibility requirements must be met for an ontology to be considered backward compatible for that specific depending artifact. The compatibility requirements are specified in terms of the Change Definition Language.

Chapter 7

Evolution in a Decentralized Environment

In the previous chapters, we discussed an approach to support the evolution of a single ontology in isolation. In Chapter 4, we explained how a version log describing the history of an ontology is constructed, introduced a Change Definition Language aimed at defining changes and meta-changes, and introduced an evolution log to describe the evolution of an ontology in terms of requested, deduced and detected changes. In Chapter 5, we illustrated how the Change Definition Language is used to express the different types of changes and how these definitions are evaluated either to request or detect changes. In Chapter 6, we presented an approach to both check the consistency of a new ontology version and resolve inconsistencies when needed. Furthermore, we discussed a manner to verify whether a new ontology version is backward compatible with an older version for a given depending artifact.

In this chapter, we abandon the track of considering only a single ontology but rather focus on the problems that arise when dealing with a *network* of ontologies. As ontologies are defined as a specification of a shared conceptualization, ontologies are intended to be reused by several actors, possibly for different purposes. Moreover, ontologies may reuse and extend the conceptualizations specified by other ontologies. As we focus in this dissertation on the Web Ontology Language OWL, the environment these OWL ontologies normally reside in is the World Wide Web (WWW). One of the main characteristics of the WWW is that it is a decentralized environment, which means that there is not a central authority being in control. In this chapter, we discuss the influence of this decentralized characteristic on the process of ontology evolution. The key principle turns out to be *tolerance*.

This chapter is structured as follows. In Section 7.1, we discuss the problems and limitations that come with a decentralized environment. In Section 7.2, we revise the version log we introduced in Section 4.1 on Page

59 to be able to deal with ontologies that use and extend other ontologies. In Section 7.3, we present the effects of dealing with multiple ontologies on the ontology evolution framework. In Section 7.4, we discuss how a depending ontology can be updated to a newer version of the ontology it depends on and how consistency with this new version can be assured. In Section 7.5, we focus on the problem where a refusal of an ontology engineer to update an ontology may cause all depending ontologies to be blocked. We conclude this chapter with a brief summary in Section 7.6.

7.1 Overview

As is clearly illustrated by the stack of Semantic Web layers (see Figure 2.1 on Page 17), OWL is built on top of the URI layer. This URI layer provides means for identifying resources in the Semantic Web i.e., any resource can be referred to using a URI. Based on this principle of URIs, ontologies can use and extend other ontologies by referring to concepts defined in other ontologies. OWL supports a number of constructs to enhance the use and extension of other ontologies. It allows to include XML namespace declarations to offer a means to unambiguously interpret identifiers and make the ontology more readable. Furthermore, it provides a construct to import other ontologies. Importing another ontology brings the entire set of axioms defined in that ontology into the importing ontology. Note that import statements are transitive i.e., if ontology A imports B, and B imports C, then A imports both B and C.

To better understand the problems associated with a decentralized environment, we introduce an example. The example environment is depicted in Figure 7.1. The environment consists of three ontologies. The base ontology is the 'friend-of-a-friend' (*foaf*) ontology describing people, the relations between them and the things they create and do (Figure 7.1 only shows a small part of this ontology). A second ontology, called *emp*, describes certain aspects of the business domain. This ontology expresses for example that an employee works for a company and that a company executes projects. The *emp* ontology bases its concept definitions on concepts defined in the *foaf* ontology. E.g., the *emp* Class *Employee* is defined as a subclass of the *foaf* Class *Person*. A third ontology describes concepts commonly associated with a university. We refer to this ontology as the *uni* ontology. The *uni* ontology depends in its turn on the *emp* ontology[1]. The ontology introduces a Class *Academic Staff* and a Class *University*, both defined as subclass of respectively the *emp* Classes *Employee* and *Company*. Furthermore, it adds an additional restriction on the *worksFor* Property of the *emp* ontology requiring all values to be instances of the Class *University*. We

[1]Note that the *uni* ontology indirectly depends on the *foaf* ontology through the *emp* ontology.

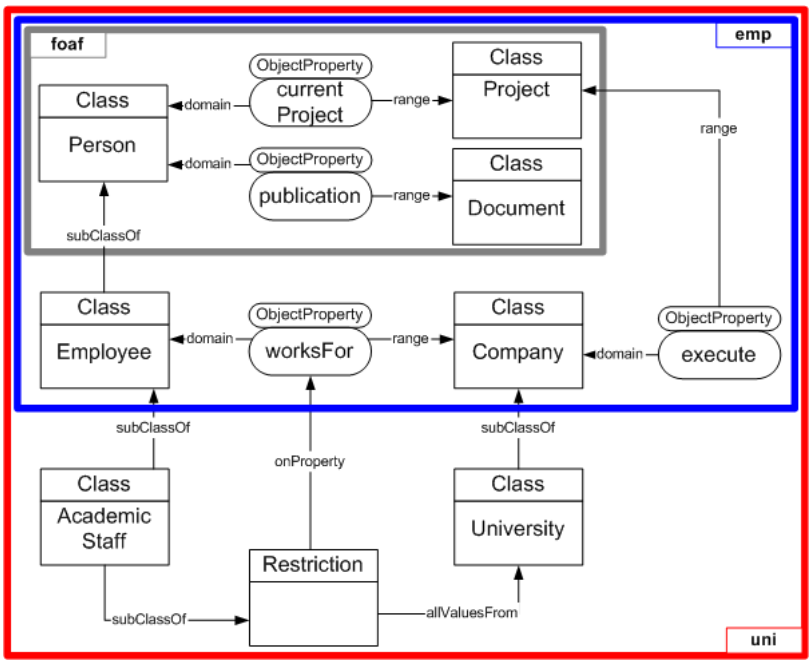

Figure 7.1: Example of ontology extensions

say that ontology *emp* depends on ontology *foaf*, and that the ontology *uni* depends on both the ontology *emp* and the ontology *foaf*.

It is clear from Figure 7.1 that changes to an ontology may have (besides consequences for the changed ontology itself) consequences for depending ontologies. E.g., the decision to remove the Class *Project* from the *foaf* ontology would turn both the ontologies *emp* and *uni* into an inconsistent state. The ontology *emp* becomes inconsistent because the range of the *execute* Property points to a non-existing Class, and the ontology *uni* becomes inconsistent because it imports an inconsistent ontology *emp*.

Current solutions often rely on a technique called *change propagation* where changes made to an ontology are further propagated to depending ontologies in order to resolve the inconsistencies of these depending ontologies [60]. In our example, the propagation of the deletion of the *Project* Class in the *foaf* ontology could lead to the deletion of the *execute* Property in the *emp* ontology, thereby restoring the consistency of this depending ontology. While this approach might work in a centralized controlled environment, the applicability of this technique in a decentralized environment as the WWW is questionable at least. It is unfeasible to force ontology engineers managing depending ontologies to update their ontologies on demand. Moreover, in a setting as the WWW, it is in the first place impossible to know which ontologies are depending on a given ontology as linking is a unidirectional operation.

We distinguish three types of dependencies between depending artifacts and ontologies: *intra dependencies*, *managed inter dependencies* and *unmanaged inter dependencies*. They are defined as follows:

- **Intra dependencies:** these are dependencies between concepts defined within one ontology. An intra dependency between two concepts exists when both concepts are interconnected. E.g., the Class *Document* is defined in the *foaf* ontology in Figure 7.1 as being the range of the *publication* Property. We say that there exists an intra dependency between *publication* and *Document*.

- **Managed inter dependencies:** these are dependencies between concepts defined in two different ontologies, but both ontologies are managed by the same authority. E.g., in our example shown in Figure 7.1, a managed inter dependency may exist between the Classes *AcademicStaff* in *uni* and *Employee* in *emp* assuming that both ontologies are controlled by the same ontology engineer.

- **Unmanaged inter dependencies:** these are dependencies between concepts defined in two different ontologies, and both ontologies are managed by different authorities. E.g., in our example shown in Figure 7.1, an unmanaged inter dependency may exist between the Classes *Employee* in *emp* and *Person* in *foaf* assuming that both ontologies are **not** controlled by the same ontology engineer.

The technique of change propagation may be appropriate for both intra dependencies and managed inter dependencies (where in both cases one has sufficient permissions to make changes to the depending ontology), it becomes unusable for unmanaged inter dependencies.

To overcome this problem, an approach of *ontology replication*, or sometimes called *reuse through replication*, is often introduced [83]. In this approach, ontologies are not used and extended by directly referring to the original ontology. Instead, a replication of the original ontology is made and stored locally, and ontologies use and extend this replica instead of the original ontology. The benefit is that changes to the original ontology don't directly affect the depending ontologies as they rely on a replica of the original ontology that remains unchanged. As a consequence, ontology engineers of depending artifacts are not forced to update, and they may update at their own pace. Reuse through replication has a number of advantages. A first advantage concerns the reliability of a system. Reuse through replication ensures a low coupling between a depending ontology and the ontologies it depends on. This prevents a failure of one ontology server to cause failure of all depending ontology servers. A second advantage concerns the performance of a system. Ontology replication avoids a huge communication overhead between different ontology servers. Although for some types of

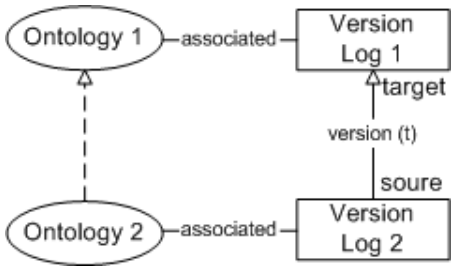

Figure 7.2: Using and extending ontologies by means of an associated version log

applications, these advantages may justify the appliance of ontology replication, we consider it in general an unsatisfying approach for two reasons. First of all, ontology replication opposes the main principle of the WWW as a distributed and decentralized environment. Imagine that every link that is created on the Web would result into a replication of the target resource of that link! Secondly, in order to correctly replicate an ontology, it demands replicating all the ontologies it depends on, plus replicating all the ontologies these ontologies depend on, etc. When taking into account that ontologies themselves may grow fairly large and that dependencies between ontologies may become numerous, reuse through replication becomes impracticable.

In this chapter, we propose an approach that overcomes the disadvantages of both the approach where ontologies are directly linked to other ontologies, and the approach of ontology replication. Instead of linking directly to an ontology, we link to a given version of an ontology by making use of the version log associated with the different ontologies. Changes applied to an ontology won't affect depending ontologies as these depending ontologies rely on a specific version that can be reconstructed by means of the associated version log. Furthermore, it overcomes the necessity to replicate all ontologies an ontology directly or indirectly depends on. Figure 7.2 visualizes the proposed approach. Dependencies between two ontologies are now represented by dependencies between two versions of these ontologies. The source of a dependency is always the latest version of the associated ontology, the target of a dependency is the version explicitly specified by a time point t of the associated ontology. In order to use this approach, we need to extend the version log as introduced in Section 4.1. We discuss this extension in the next section.

7.2 Revised Version Log

In order to express dependencies between ontologies in terms of dependencies between ontology versions using the version logs associated with the respective ontologies, we need to extend the version log in order to support

dependencies between ontologies because the version log as it was defined in Section 4.1 only supports a single ontology. We first define how to express a dependency on an ontology with a given time point indicating the version of the ontology to be used, before extending the definition of a version log. A dependency also indicates whether the ontology needs to be imported or not. A dependency is therefore defined as follows:

Definition 7.1 (Dependency). *Assume* $\boldsymbol{U}$ *to be the set of all possible URIs and* $\boldsymbol{S}$ *to be the set of all possible strings. A dependency* d *on an ontology* O *with associated version log* Ω *is defined as a five-tuple, so that* $d = \langle \mathrm{NS}, i, \mathrm{URI}_O, \mathrm{URI}_\Omega, t \rangle$ *where* $\mathrm{NS} \in \boldsymbol{S}$ *is the namespace assigned to the ontology* O*,* $i \in \{true, false\}$ *indicates whether the ontology* O *needs to be imported or not,* $\mathrm{URI}_O \in \boldsymbol{U}$ *is the URI of the ontology* O*,* $\mathrm{URI}_\Omega \in \boldsymbol{U}$ *is the URI of the version log* Ω *associated with* O*, and* $t \in \boldsymbol{T}$ *is a time point of the timeline associated with the version log* Ω *to indicate the version to be used.*

For each ontology O that a given ontology O_d depends on, we add a dependency d to the version log Ω associated with O_d. The namespace NS of the dependency represents the namespace that is assigned to the ontology O used in O_d. When no namespace is assigned for O in O_d, the empty string is assigned to NS. Important to mention is that the i of a dependency, which indicates whether the ontology O needs to be imported or not, greatly affects the semantics of the ontology O_d. Therefore, the decision whether to import an ontology or not exerts a great influence on the consistency checking process [28]. Importing an ontology O means an inclusion of all the axioms defined in that ontology. These added axioms are taken into account when checking ontology consistency of O_d. When deciding not to import an ontology O, nothing of the semantics of this ontology is brought into the depending ontology O_d, which means that the axioms of O are not taken into account when checking ontology consistency of O_d. Furthermore, an ontology that is specified to depend on itself is considered a null-operation. Finally, when creating a dependency of O_d on O, the time point t of the dependency refers, in general, to the latest time point of O (i.e., to the *now* time point of O).

To keep track of the dependencies of an ontology, we need to extend the version log that is associated with that ontology. As with ontology concepts, dependencies may change over time. We therefore introduce, analogously to concept versions, the notion of dependency versions. A dependency version expresses the validity in time of a dependency by keeping track of a start and end time point, and stores the state of the dependency (i.e., either 'pending', 'confirmed' or 'implemented'). The semantics of these states is the same as with concept versions. A dependency version is defined as follows:

Definition 7.2 (Dependency Version). *A dependency version* ω *is a tuple* $\langle d, s, t_s, t_e \rangle$ *where* d *is a dependency,* $s \in \{$ *'pending', 'confirmed', 'implemented'* $\}$ *is the state of the dependency version, and* t_s, $t_e \in \boldsymbol{T}$ *are*

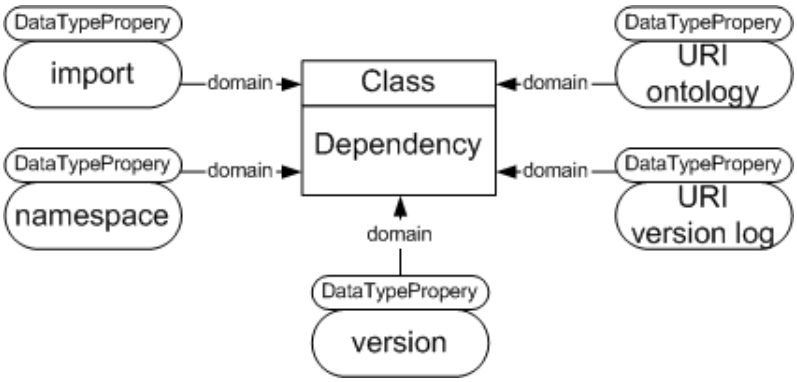

Figure 7.3: Properties of a Dependency

respectively the start- and end time of the dependency version. The start- and end time depict a closed interval $[t_s, t_e]$.

In Section 4.1, Definition 4.15 defined a version log as a tuple $\langle \mathbf{O}, \mathbf{V} \rangle$ where $\mathbf{O}$ is the associated ontology and $\mathbf{V}$ is the set of concept evolutions. In the following definition, we extend the original definition of a version log to be able to express that the associated ontology $\mathbf{O}$ depends on one or more other ontologies. We realize this by extending the tuple representing a version log to include a set of dependency versions. The definition of an extended version log is given as follows:

Definition 7.3 (Extended Version Log). *Assume N to be the set of all used concept names. An extended version log* Ω *is defined as the tuple* $\langle \boldsymbol{O}, \boldsymbol{D}, \boldsymbol{V} \rangle$ *where* $\boldsymbol{O}$ *is the associated ontology,* $\boldsymbol{D}$ *is a set of dependency versions, and* $\boldsymbol{V}$ *is the set of concept evolutions, so that*

$$\boldsymbol{V} = \bigcup_{\forall \sigma \in \boldsymbol{N}} \boldsymbol{E}_\sigma$$

As is the case with regular ontology concepts (i.e., Classes, Properties and Individuals), dependencies can also be added to an ontology, modified or removed from an ontology. This requires that ontology engineers must be able to request changes to dependencies, and that changes to dependencies must be detectable by the detection process of our approach. To realize this, we first need to extend the OWL meta-schema we introduced in Section 4.3.1 to be able to express change definitions involving dependencies. Figure 7.3 depicts the extension to the OWL meta-schema to include dependencies. It introduces a Class *Dependency* as the domain of five Properties *namespace*, *import*, *URI_ontology*, *URI_versionlog*, and *version*. For the interested reader, the complete meta-schema can be found at `http://wise.vub.ac.be/ontologies/OWLmeta.owl`.

Now that we have extended the OWL meta-schema, we are able to extend the set of primitive changes that we introduced in Section 5.4 (see Table 5.1 and 5.2) in order to add and delete dependencies. For this purpose, we add the following two primitive changes: *addDependency* and *deleteDependency*. Both their change definitions are given as follows:

- **Name:** addDependency(?n, ?i, ?o, ?l, ?v)
  Description: *Represents the change where a new dependency is added to the ontology with a namespace* `?n`*, a URI to an ontology* `?o`*, a URI to a version log* `?l`*, and a version* `?v`*. The parameter* `?i` *indicates whether the ontology needs to be imported or not.*
  Definition:

```
addDependency(?n, ?i, ?o, ?l, ?v) :=
    NOT <PREVIOUS>(Dependency(?d)) AND
    Dependency(?d) AND
      namespace(?d, ?n) AND import(?d, ?i)
      URI_ontology(?d, ?o) AND
      URI_versionlog(?d, ?l) AND version(?d, ?v);
```

- **Name:** deleteDependency(?d)
  Description: *Represents the change where a dependency is deleted from the ontology.*
  Definition:

```
deleteDependency(?d) :=
    <PREVIOUS>(Dependency(?d)) AND
    NOT Dependency(?d);
```

In the same way, complex changes can be defined to modify the different aspects of a dependency (e.g., changing the namespace of a dependency). However, we omit the change definitions of 'modify'-changes concerning dependencies from the discussion here. These change definitions are evaluated in the same way as discussed in Chapter 5.

With these aforementioned extensions to the version log and the introduction of two additional primitive changes, ontology engineers can express a request to add and delete dependencies, and changes to dependencies can automatically be detected by the ontology evolution framework. The only remaining shortcoming is that we don't have a manner in the version log to actually refer to concepts defined in ontologies we depend upon. We therefore change the definition of a concept name given in Section 4.1.2. While in the former definition, a concept name was a simple string, the new definition defines a concept name as the combination of a URI and a string. The URI is either the URI of the ontology associated with the version log, or one of the ontology URIs of the declared dependencies. We therefore revise the definition of a concept name as follows:

Definition 7.4 (Extended concept name). *Assume* $\boldsymbol{S}$ *to be the set of all possible strings and* $\boldsymbol{U}$ *to be the set of all possible URIs. We then define the set of Class names* $\boldsymbol{CN}$ *as* $\boldsymbol{CN} \subseteq \boldsymbol{U} \times \boldsymbol{S}$*, the set of Property names* $\boldsymbol{PN}$ *as* $\boldsymbol{PN} \subseteq \boldsymbol{U} \times \boldsymbol{S}$*, and the set of Individual names* $\boldsymbol{IN}$ *as* $\boldsymbol{IN} \subseteq \boldsymbol{U} \times \boldsymbol{S}$*. We now define the set of all concept names* $\boldsymbol{N} = \boldsymbol{CN} \cup \boldsymbol{PN} \cup \boldsymbol{IN}$*. Furthermore, we require that* $\boldsymbol{CN} \cap \boldsymbol{PN} \cap \boldsymbol{IN} = \emptyset$*.*

As defined above, concepts names consist of the name of the concept itself and the URI of the ontology where the concept is defined. Important to note is that this ontology URI should **never** be used directly when following the reference to a concept as the ontology may have changed in the mean time, possibly breaking the reference. Instead, the URI of the version log (URI_Ω) and the associated version (t) should be resolved from the dependency $\langle \mathrm{NS}, i, \mathrm{URI}_O, \mathrm{URI}_\Omega, t \rangle$ where URI_O is the ontology URI. Therefore, following a reference to a concept in an ontology means following a reference to a concept in a reconstructed version of the ontology. A version of an ontology is reconstructed by means of $\mathbf{S}(t)$ where t indicates the version in a given version log.

7.3 Framework Effects

When only considering a single ontology, the axioms to consider for the consistency checking, backward compatibility checking, change detection and inconsistency resolving task are only those axioms defined in that particular ontology. When we proceed to a more complex situation, where ontologies can import other ontologies, the situation becomes different.

In this section, we discuss the effects that the imports of ontologies have on these different tasks of our ontology evolution framework. The effects on most of these tasks are rather limited as we will see in Section 7.3.1, which discusses the consistency & backward compatibility checking task, and in Section 7.3.2, which discusses the change detection task. However, the effects on the inconsistency resolving task will be more profound as we will see in Section 7.3.3.

7.3.1 Consistency & Backward Compatibility Checking

Remember that importing an ontology means that the semantics defined by that ontology must be included in the importing ontology. In other words, the axioms to consider for an ontology O_1 that imports an ontology O_2 are the axioms defined in both O_1 and O_2. As a consequence, the ontology evolution framework needs to take all the axioms of O_1 and all the axioms of the imported ontologies into consideration when checking the consistency of the ontology O_1 as no contradictions must exist between the axioms defined in O_1 and the axioms defined in the imported ontologies. The same holds

for the backward compatibility checking task. When checking whether a new version of the ontology O_1 remains backward compatible for a certain depending artifact, the ontology evolution framework needs to take all the axioms of O_1 and all the axioms imported by O_1 into account.

7.3.2 Change Detection

The effects on the change detection task slightly differ from the consistency & backward compatibility checking task discussed in the previous subsection. When detecting changes that have occurred in an ontology, we are only interested in the changes that have occurred in that particular ontology, but not in changes that have occurred in ontologies it depends on. The evolution log that results from the change detection task describes the evolution in terms of change occurrences of exactly one ontology. We therefore restrict the change detection task to the detection of changes to axioms defined in one ontology, although axioms defined in imported ontologies may be taken into account in order to satisfy change definitions.

7.3.3 Inconsistency Resolving

As discussed in Chapter 6, the extended tableau algorithm allows determining a set of axioms that are responsible for a detected inconsistency. An ontology engineer can resolve an inconsistency within one single ontology by applying a set of rules on one of the axioms forming the cause of the inconsistency. Note that an ontology engineer can opt to change any of these axioms. The reason for this is that dependencies within an ontology are all intra dependencies which are controlled by the same ontology engineer i.e., he has sufficient permissions to change all axioms. When considering imported ontologies, inconsistency resolving becomes more complex as the set of axioms causing the inconsistency may be composed of axioms defined in different ontologies. The solution to restore consistency depends on the type of dependency that exists between ontologies.

We first introduce a small example that we will use in this section to explain the different cases. Consider an ontology O_1 consisting of the following axioms: $C \sqsubseteq A$ and $B \sqsubseteq \neg A$. A second ontology O_2 imports ontology O_1. The ontology engineer of O_2 decides to add the following axiom to O_2: $O_1 : C \sqsubseteq O_1 : B$. Note that we use the syntax $O_1 : C$ to refer to concepts defined in an imported ontology where O_1 represents the URI of the ontology and C represents a concept. Before the requested change can be implemented, it needs to be verified whether the ontology O_2 remains consistent or not. Checking consistency of ontology O_2 means checking the consistency of the axioms of both O_1 and O_2 i.e., $O_1 : C \sqsubseteq O_1 : A$, $O_1 : C \sqsubseteq O_1 : B$ and $O_1 : B \sqsubseteq \neg O_1 : A$. Consistency checking reveals a clash between the concepts $O_1 : A \Leftrightarrow \neg O_1 : A$. The resulting set $\mathbf{S}$, representing the set of

axioms causing the inconsistency, contains for our example all three axioms i.e., $\mathbf{S} = \{O_1 : C \sqsubseteq O_1 : A, O_1 : C \sqsubseteq O_1 : B, O_1 : B \sqsubseteq \neg O_1 : A\}$.

Managed Inter Dependency

We first consider the case of a managed inter dependency between two ontologies where one imports the other. A managed inter dependency between two ontologies means that both ontologies are managed by the same ontology engineer. As a consequence, this ontology engineer has the necessary permissions to change any axiom of both ontologies. In the case of a managed inter dependency between two ontologies, both ontologies can be treated as one single ontology for the purpose of inconsistency resolving. So when we assume a managed inter dependency between the ontologies O_2 and O_2, any axiom of the set $\mathbf{S}$ can be changed to resolve the inconsistency in O_2.

Unmanaged Inter Dependency

The situation becomes different when it concerns an unmanaged inter dependency between two ontologies where one imports the other. An unmanaged inter dependency between two ontologies means that the ontology engineer of the ontology that imports the second ontology doesn't have sufficient permissions to change axioms defined in the imported ontology. As a consequence, this ontology engineer can only restore the inconsistency by changing axioms that are defined in his own ontology. Imagine in our example that the dependency between O_2 and O_1 concerns an unmanaged inter dependency. In that case, the ontology engineer of O_2 doesn't have the sufficient permissions to make changes to the axioms of $\mathbf{S}$ that do not fall under his authority. Therefore, the only axiom of the set $\mathbf{S}$ that he can change is the singleton $\{O_1 : C \sqsubseteq O_1 : B\}$ as this is the only axiom defined in O_2.

The problem is that changing only axioms defined in the ontology itself doesn't always result in a satisfying solution. It might be the case that the axioms defined in an ontology form a correct reflection of the real world situation from one ontology engineer's viewpoint, but that his view conflicts with the one described by an ontology it depends on. As a consequence, it becomes impossible to preserve the dependency between both ontologies as they represent conflicting views. Before removing a dependency between two ontologies, the imported axioms need to be duplicated in the importing ontology. As the axioms originally defined in the imported ontology now became part of the importing ontology, all axioms contained in $\mathbf{S}$, listing all axioms causing the inconsistency, can be considered to resolve the detected inconsistency.

When we consider our example, there are two possibilities. It is possible that the inconsistency of O_2 was introduced by the added axiom $O_1 : C \sqsubseteq O_1 : B$ to O_2 because it doesn't reflect the real world situation and is too

restrictive causing a contradiction with the axioms defined in O_1. In this situation, the set **S** of axioms is for example weakened by changing the axiom $O_1 : C \sqsubseteq O_1 : B$ (defined in ontology O_2) to $O_1 : C \sqsubseteq O_2 : D$ where $O_1 : B \sqsubseteq O_2 : D$. The dependency between both ontologies remains as the changed axiom is defined in O_2. A second possibility is that the ontology engineer may be convinced that the added axiom is correct and reflects the real world situation. As a consequence, both views of O_2 and O_1 clearly contradict. To turn ontology O_2 back into a consistent state, the only option is to remove the dependency on ontology O_1 and duplicate the axioms of O_1 to O_2. Ontology O_2 now consists of the following set of axioms: $\{C \sqsubseteq B, C \sqsubseteq A, B \sqsubseteq \neg A\}$ without depending any longer on O_1. All axioms originally listed in **S** may now be considered when resolving the inconsistency. For example, the ontology engineer may belief that the disjointness between A and B is inappropriate. Removing $B \sqsubseteq \neg A$ will resolve the detected inconsistency.

Range of Duplication

The previous paragraphs discussed the solution to inconsistency resolving in the case of unmanaged inter dependencies. In the situation of two incompatible ontologies, the axioms of an imported ontology are duplicated to the importing ontology and the dependency between both is removed. Note however that the duplication of axioms is not necessarily restricted to axioms of directly imported ontologies, also axioms of indirectly imported ontologies may need to be duplicated. Imagine that the ontology O_1 in our example also imports another ontology O. Whether we also need to duplicate the axioms of O when duplicating the axioms of O_1 to O_2 depends on two factors. First of all, when no axioms of O are involved in the detected inconsistency, there is absolutely no need to duplicate the axioms of O to O_2. Note that whenthe dependency between O_2 and O_1 is removed, a new dependency should be created between O_2 and O. Secondly, when axioms of O are involved in the detected inconsistency and the ontology engineer of 0_2 beliefs that the axioms of 0 don't reflect any longer the real world situation, the axioms of O (and of O_1 as it is in between 0_2 and O) need to be duplicated to O_2.

7.4 Version Consistency

In this section, we discuss the last phase of our ontology evolution framework, the *version consistency* phase. The purpose of this phase is to keep ontologies consistent whenever ontologies they depend on evolve. When a new version of an ontology is released, ontology engineers of depending ontologies are confronted with the question whether to update or not. To make a well-considered decision, the ontology evolution framework must be able

to provide the ontology engineer with information about the changes that have occurred since the version currently used (by means of an evolution log), the latest backward compatible (intermediate) version, and possible inconsistencies that occur when updating to a non-backward compatible version.

Because of the decentralized nature of the Web, ontology engineers cannot be forced to update to the latest version of an ontology they depend on. As a consequence, an ontology engineer has three different options:

- An ontology engineer can decide to not update to the latest version of an ontology, or at least not to update at this moment in time;
- An ontology engineer can decide to update to the a backward compatible version of an ontology;
- An ontology engineer can decide to update to the a non-backward compatible version of an ontology.

No update

When an ontology engineer decides not to update (or at least not to update at this moment in time), the depending ontology doesn't become inconsistent with the changed ontology it depends on. The depending ontology depends on a specific version of an ontology, and remains to depend on that version even though a newer version was released. Remember that references to concepts defined in another ontology are always resolved to a specific version of that ontology by means of a version log. It is however important for users of an ontology to ascertain themselves of which versions are being used of the ontologies this ontology depends on. For example, when querying an ontology O, all queries involving concepts defined in an ontology O_i that O depends on must be conform to the version of O_i used by O.

Update to a backward compatible version

When an ontology engineer decides to update an ontology O to a backward compatible version, the old version of the ontology can easily be replaced by the newer backward compatible version. This involves modifying the corresponding dependency in the version log, where the previous version t used is replaced by the backward compatible version. Nothing needs to be changed to the actual ontology. Note that although nothing changed for ontology O, the update to a backward compatible version may still have consequences for other ontologies that depend on O. A new version of an ontology can be considered backward compatible with an older version for one particular ontology, but this doesn't mean that the new version is also backward compatible for another ontology as each ontology defines its own backward compatibility requirements.

Update to a non-backward compatible version

Updating an ontology O to a non-backward compatible version of an ontology O_i that O depends on, is more troublesome as it requires changes to the ontology O. Changes are necessary to resolve possible inconsistencies that may occur with the new version and/or to take care of the changes in the new version of O_i that caused it to be no longer backward compatible. Consistency & backward compatibility checking and inconsistency resolving can be realized as described in respectively Section 7.3.1 and Section 7.3.3. Furthermore, it is the responsibility of the ontology engineer to deal adequately with changes in the new version of O_i that caused it to be no longer backward compatible.

Although the ontology evolution framework can verify whether a new version of an ontology O_i, that an ontology O depends on, introduces inconsistency in O or breaks backward compatibility for O, updates should always be handled with care as unintentional conceptualizations may arise that cannot be identified automatically by the ontology evolution framework. This problem arises when O and O_i evolve independently in the same way. Consider the example shown in Figure 7.1 where the ontology *uni* uses and extends the ontology *emp* (e.g., by defining a Class *AcademicStaff* as subclass of the *Employee* Class defined in *emp*). Assume that the ontology engineer of *emp* now decides to change his ontology in a similar way: creating different types of employees among which a Class *AcademicStaff*. When the *uni* ontology is updated to the latest version of *emp*, the Class *Employee* has among others two distinct subclasses *uni:AcademicStaff* and *emp:AcademicStaff*, although both concepts represent the same real world concept. It is however impossible for the ontology evolution framework to decide that both classes are equal. Solutions to overcome this problem includes removing the *uni:AcademicStaff* or stating that both concepts are actually the same concept (using the OWL `sameAs` Property), although this last option would turn the ontology into an OWL Full ontology.

7.5 Blocked Ontologies

In a decentralized environment as the WWW, ontology engineers cannot be forced to update their ontologies when an ontology they depend on changes. For diverge reasons, ontology engineers may decide to update at a later moment in time, or decide to update not at all. A problem is that the refusal of an ontology engineer to update his ontology prevents the ontology engineer of depending ontologies to update either. Consider as example the set of ontologies outlined in Figure 7.1. An ontology *uni* imports an ontology *emp* that in its turn imports an ontology *foaf*. The dependencies between the three ontologies are considered to be unmanaged inter dependencies. Assume that the ontology engineer of *foaf* releases a new version of its

ontology. The changes to this ontology since the last public version include the addition of numerous new Properties for the Class Person (e.g., first name, surname, homepage, ...) and the deletion of the *Project* Class and *currentProject* Property.

The ontology engineer of *emp* decides not to update his ontology to the latest version of *foaf* at this moment in time. Updating would break the consistency of *emp* as the range of the *execute* Property refers to the deleted Class *Project*. Instead, the ontology *emp* remains depending on the old version of *foaf*.

The ontology engineer of *uni* on the other hand is eager to update as he likes to make use of the new Properties added to the new version of *foaf*. However, the refusal of the ontology engineer of *emp* to update to the latest version of *foaf* blocks the update wishes of the *uni* ontology engineer. The ontology *uni* cannot directly update to the latest version of *foaf* as it only depends indirectly on *foaf* through *emp*. Unfortunately, *emp* still depends on the old version of *foaf*. The ontology engineer of *uni* cannot add the latest version of *foaf* as a direct dependency, as this would mean that the ontology depends on two, not necessarily consistent versions of the same ontology. We say that the ontology *uni* is *blocked*. A blocked ontology gets *unblocked* whenever the ontology it depends on is updated. So, the ontology *uni* gets unblocked whenever *emp* updates to the latest version of *foaf*.

Waiting until *emp* updates to the new version of *foaf* is in a lot of cases not a satisfiable solution. The ontology engineer of *uni* is probably unaware of when *emp* will be updated or if it will ever be updated in the first place. Even if it is known when the ontology *emp* will be updated, forcing the ontology engineer of *uni* to postpone updating is a severe limitation. This limitation is even more severe when we take into account that all ontologies depending on *uni* also get blocked. A naive solution is to duplicate all the axioms introduced in the new version of *foaf* that are of interest to the ontology engineer of *uni* to the *uni* ontology. In this way, it is no longer necessary to update to the latest version of *foaf* in order to make use of the new axioms that were introduced. This solution has a number of serious drawbacks. First of all, the solution introduces redundancy by duplicating the axioms. When the *emp* ontology eventually updates, the copied axioms are defined twice, once in ontology *foaf* and once in ontology *uni*. Secondly, the solution can only be applied when the duplicated axioms would not cause an inconsistency with axioms of *emp*. Another solution consists of removing the dependency between the ontologies *uni* and *emp*, and duplicating all axioms of *emp* to *uni*. This solution allows *uni* to update to the new version of *foaf* as it controls now the former axioms of *emp*. In this situation, we distinguish two cases. In a first case, the ontology engineer of *emp* never intends to update. The proposed solution is in this case a satisfiable solution as it doesn't make sense to maintain the dependency with an ontology that is no longer maintained. In a second case, the ontology engineer of *emp* in-

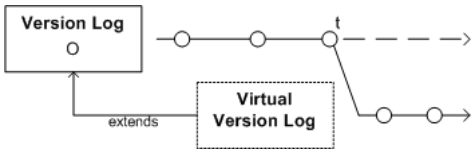

Figure 7.4: Extension of a version log

tends to update his ontology, but at a later moment in time. The drawback of the proposed solution is that the dependency between the two ontologies is cut and the ontology engineer is burdened with the responsibility of maintaining the axioms originating from *emp*, although an update of *emp* may follow shortly. The proposed solution doesn't handle blocked ontologies well because it immediately breaks the dependency between ontologies. The problem of course is that, in general, there is no way of knowing whether an ontology will ever be updated or not.

Virtual Version Log

We propose a new approach to manage blocked ontologies by introducing the notion of a *virtual version log*. Note that this approach doesn't prevent ontologies from getting blocked, instead it offers ontology engineers a chance to bypass a blocked ontology. When an ontology O is blocked because an ontology O_2 it depends on refuses at this moment in time to update to the latest version of O_1 it depends on, the ontology engineer of O can create a virtual version log that extends the current version log associated with O_2 and that depends on the latest version of O_1. The purpose of the virtual version log is to *update* the ontology O_2 to the latest version of O_1 and to resolve all possible inconsistencies that occur as result of the update. Note that the ontology engineer of O does not actually change the ontology O_2 (he also doesn't have the permissions to do so), instead the changes are simulated by creating a virtual version log that extends the original version log of O_2. We call this type of version log *virtual* as it describes the hypothetical evolution of the O_2 ontology i.e., the evolution of O_2 that the ontology engineer of O wished for in order to update to the latest version of O_1. The version log of O is changed to depend on the virtual version log instead of on the version log of O_2. From the view point of ontology O, ontology O_2 has updated to the latest version of O_1, although nothing has changed for ontology O_2 as it still depends on the old version of O_1.

A virtual version log is fairly similar to a *normal* version log, although there is a point of difference. A virtual version log is not directly associated with an ontology as it is the case with a normal version log. Instead, it extends another version log by referring to a version log and a time point. This time point indicates the moment in time of the extension. The virtual version log represents a sidetrack of the evolution of the ontology associated

with the version log it extends. Figure 7.4 illustrates this graphically. The virtual version log extends the version log associated with the ontology O. The virtual version log represents an alternative evolution of O after time point t. Note that the version log of O may describe an alternative evolution after time point t. The definition of a virtual version log is given as follows:

Definition 7.5 (Virtual Version Log). *We define a virtual version log* Ω_v *as the tuple* $\langle E, \boldsymbol{D}, \boldsymbol{V} \rangle$ *where* E *is a tuple representing the version log it extends so that* $E = \langle \Omega, t \rangle$ *where* Ω *is a version log and* t *is the time point of the extension,* $\boldsymbol{D}$ *is a set of dependency versions, and* $\boldsymbol{V}$ *is the set of concept evolutions.*

The purpose of a virtual version log is to bypass the problem of blocked ontologies by allowing ontology engineers to update a blocked ontology without having to wait until the ontology gets unblocked. A virtual version log is used to simulate the necessary changes required to resolve possible inconsistencies caused by an update. As is clear from Figure 7.4, the owner of a virtual version log describes with his virtual version log how he should evolve the ontology O in order to update to the latest version of an ontology that O depends on. Note that, when the ontology engineer of O eventually decides to update and unblocks the ontologies that depend on O, the real evolution of O may differ from the virtual one. We discuss the different possibilities that may occur in that case at the end of this section. The advantage of virtual version logs, compared to previously mentioned solutions, is that a link is maintained with the original version log and that no axioms need to be duplicated.

Figure 7.5 visualizes the different version logs of the example shown in Figure 7.1. At the top left of the figure, the version log associated with the *foaf* ontology is shown. The release of a new version of the *foaf* ontology (referred to as *foaf'*) is accompanied with an evolved version log (see 1), shown at the top right of the figure. The ontology *uni* is blocked for as long as the ontology engineer of *emp* doesn't update to the new version of *foaf* and the ontology *uni* depends on *emp*. In order for the ontology engineer of *uni* to bypass the ontology block, he creates a virtual version log that extends the version log of *emp* and depends on the latest version of *foaf* (see 2). The virtual version log Ω_v initially looks as follows (we use Ω_{emp} to refer to the version log of *emp* and d to represent the new dependency on the latest version of *foaf*):

$$\Omega_v = \langle E, \mathbf{D}, \mathbf{V} \rangle$$

where $E = \langle \Omega_{emp}, 28 \rangle$, $\mathbf{D} = \{\langle d, \text{'confirmed'}, 29, now \rangle\}$, and $\mathbf{V} = \{\}$.

The extension E declares that the virtual version log extends version log Ω_{emp} after time point 28 i.e., the evolution of *emp* before time point 28 is described by Ω_{emp}, a (possible) evolution after time point 28 is described

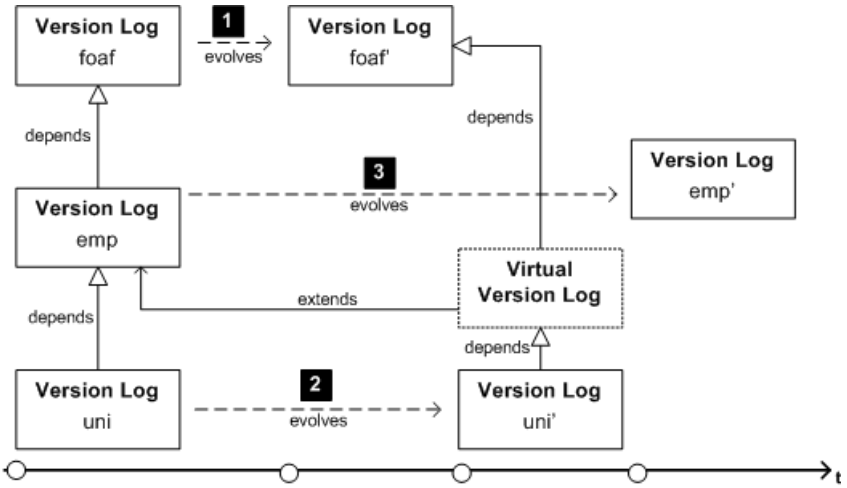

Figure 7.5: Example use of a virtual version log

by Ω_v[2]. The first change that is simulated by the Ω_v is a modification of the dependency on *emp* on time point 29, so that it now depends on the latest version of *foaf*. The set of concept versions $\mathbf{V}$ is initially empty as no concepts have been changed yet. Changing the dependency to the latest version of *foaf* may cause inconsistencies in the virtual ontology that the virtual version log represents. An actual ontology can be extracted from the virtual version log using $\mathbf{S}(now)$ (where *now* is the current time for the virtual version log). Consistency checking and inconsistency resolving for the extracted ontology can be done as discussed in Section 7.4. Resolving inconsistencies results in the addition of new concept versions to the set $\mathbf{V}$ of the virtual version log. Finally, the ontology *uni* needs to be changed so that the dependency on *emp* is based on the created virtual version log Ω_v, instead of the version log of *emp* i.e., Ω_{emp}.

Unblocking Ontologies

As already mentioned, an advantage of the appliance of virtual version logs is that a blocked ontology can be updated awaiting the update of the ontology it depends on. Figure 7.5 shows in step 3 an update of the *emp* ontology to the new version of the *foaf* ontology. Note that the updated version of the *emp* ontology may be different from the one represented by the virtual version log created by the ontology engineer of *uni* as they can have different views on the same domain.

After the *emp* ontology has been updated, the ontology engineer of *uni* needs to decide whether he wants to restore the dependency to the original *emp* ontology instead of using the virtual ontology he created. This decision

[2]The time point 28 is a random chosen time point for the sake of this example.

depends most likely on the fact whether his view on the evolution of *emp* can be brought in correspondence with the view of the ontology engineer of *emp* itself, and the cost of doing this. Replacing the virtual ontology with the original ontology can be seen as an evolution from the virtual ontology (which is seen as the old version) to the original one (which is seen as the new version). The ontology evolution framework is able to provide:

- detailed information about the changes that were applied to *emp* by means of the evolution log listing requested, deduced and detected changes and meta-changes;
- information about whether the updated version of the original ontology (i.e., the new version) is backward compatible with the virtual version (i.e., the old version) for the *uni* ontology;
- information about possible inconsistencies in the *uni* ontology that get introduced when going back to the original ontology.

Based on this information, the ontology engineer of *uni* needs to take a decision whether to restore the original dependency or not. If he chooses to restore the original dependency, the version log of the dependency on ontology *emp* is changed from the virtual version log to the version log of *emp*. Possible inconsistencies need to be resolved as described in Section 7.4. Note that the loss of changes introduced by the virtual version log is no loss of information as they were only applied to resolve inconsistencies due to the update to the new version of *foaf*. When the ontology engineer of *uni* decides not to change the dependency, the dependency with the virtual version log can be maintained or he can extract an ontology from the virtual version log and change the former dependency from an unmanaged inter dependency to a managed inter dependency.

Depending on Virtual Ontologies

Consider for our example a fourth ontology that depends on the *uni* ontology. We refer to this fourth ontology as the *vub* ontology. This ontology introduces additional concepts which are not present in the *uni* ontology but are needed for a particular university. In this case, it concerns the 'Vrije Universiteit Brussel' (the 'Free University of Brussels' in English). As previously seen, the *uni* ontology is updated by means of a virtual version log that extends the version log of *emp* in order to use and extend the latest version of the *foaf* ontology. From the point of view of the ontology engineer of *vub*, it seems that both the ontologies *uni* and *emp* have been updated to the latest version of *foaf*, although the update of *emp* was simulated by the ontology engineer of *uni* by using a virtual version log. When the ontology engineer of *vub* agrees with the applied changes, he can update to the new

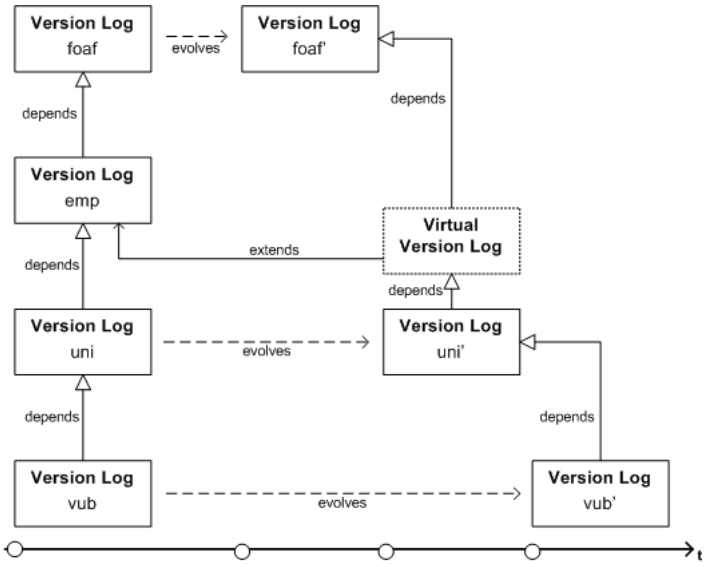

Figure 7.6: Example of an ontology depending on a virtual version log

version of *uni* by changing its dependency on the *uni* ontology to the latest version (and possibly apply other changes in order to maintain consistency). Figure 7.6 visualizes this situation. Note that the *vub* ontology actually depends on a virtual version of the *emp* ontology as its version log indirectly depends on a virtual version log extending the version log of *emp*.

Important to note is that an ontology engineer is not restricted to one single virtual version log to bypass a blocked ontology. Consider the example given in Figure 7.7. Once more, the ontology engineer of *foaf* has released a new version of its ontology. In contrast to previous examples, both the ontology engineers of the *emp* and *uni* ontologies are not willing to update immediately. However, the ontology engineer of the *vub* ontology is very interested in using the new version of the *foaf* ontology. Note that the *vub* ontology is blocked as (in this case two) ontologies it depends on do not update at this moment in time. To bypass the blocking, the ontology engineer of *vub* first creates a virtual version log that extends the version log of the *emp* ontology and depends on the new version of the *foaf* ontology. The purpose of the virtual version log is to resolve possible inconsistencies in the *emp* ontology as result of the update to the new version of the *foaf* ontology. Subsequently, he creates a second virtual version log that extends the version log of the *uni* ontology and depends on the previously created virtual version log. The purpose of this second virtual version log is to resolve possible inconsistencies in the *uni* ontology as result of the update to the virtual version of the *emp* ontology. Finally, the *vub* ontology is changed so that it depends on the virtual ontology represented by the second virtual version log instead of the original *uni* ontology.

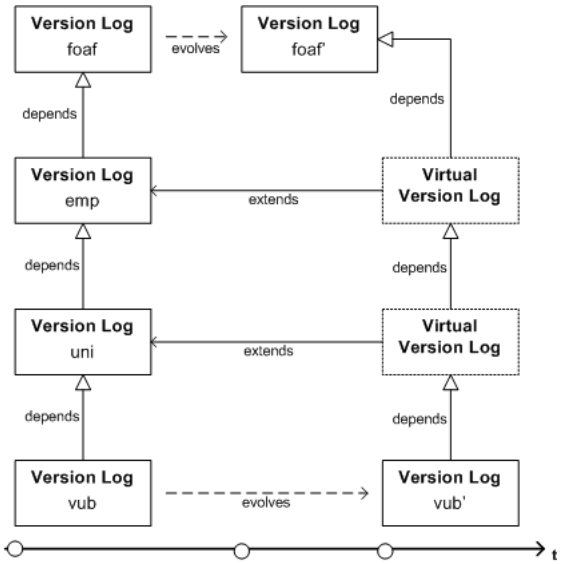

Figure 7.7: Example use of multiple virtual version logs

7.6 Summary

In this Chapter, we discussed the problem of evolution of multiple ontologies in a decentralized environment such as the Semantic Web. The problem is that changes to one single ontology may turn depending artifacts into an inconsistent state. The technique of change propagation, where changes are propagated to depending artifacts to restore inconsistency, no longer holds in a decentralized environment as maintainers of depending artifacts cannot be forced to update. As a solution, we redefined dependencies between depending artifacts and ontologies to dependencies between depending artifacts and ontologies *at a given moment in time* i.e., a depending artifact depends on a specific version of an ontology. When an ontology changes, its depending artifacts remains consistent as they still dependent on the old version. Its maintainers can update at their own pace, or decide not to update to the new version.

Just as ontologies can evolve, so are dependencies between depending artifacts and ontologies subject to possible changes. In order to deal with multiple ontologies and to represent the evolution of dependencies, we revised the version log that we introduced in Section 4.1. We introduced the notion of a *Dependency Version* to describe the different versions of a dependency over time. Furthermore, we also extended the set of primitive change definitions to support changes on dependencies. Moreover, we also discussed the effect of multiple ontologies on the consistency & backward compatibility checking task, change detection task and inconsistency resolving task of the ontology evolution framework.

Finally, we also presented the problem of *blocked ontologies*. In a decentralized environment, ontology engineers cannot be forced to update their ontologies when an ontology they depend on changes. For diverge reasons, ontology engineers may decide to update at a later moment in time, or decide to not update at all. A problem is that the refusal of an ontology engineer to update his ontology prevents the ontology engineers of depending ontologies from updating either. We say that these ontologies are blocked. We provided an answer to the problem of blocked ontologies by introducing a *virtual version log*. A virtual version log allows a maintainer of a depending artifact to simulate a new version of an ontology it depends on without actually changing that ontology.

Chapter 8

Implementation

In the previous chapters, we have described a complete ontology evolution framework to support and guide ontology engineers and maintainers of depending artifacts in the evolution process of ontologies in order to maintain consistency of both ontologies and depending artifacts. Chapter 4 discussed the foundations of the ontology evolution framework. Chapter 5 explained how changes and meta-changes can be formally defined and how conceptual change definitions are evaluated for both the purpose of change requests and change detection. Chapter 6 presented an approach to determine the cause of an inconsistency and to offer solutions to the ontology engineer to resolve inconsistencies. Furthermore, a method was proposed to verify the backward compatibility of an ontology w.r.t. a given depending artifact. Finally, Chapter 7 discussed the problems associated with ontology evolution in a decentralized environment such as the Web.

In this chapter, we discuss a number of prototype implementations that serve as proof of concept for the feasibility of the main ideas presented in this dissertation. We have opted to implement our different software artifacts as an extension to the Protégé ontology editor[1] [65]. This chapter is structured as follows. Section 8.1 discusses a plug-in for the Protégé ontology editor to automatically create a version log representing the evolution of an ontology. Section 8.2 discusses the implementation of the Change Definition Language and its evaluation to automatically detect changes. Section 8.3 discusses the extensions that were implemented to the FaCT++ Description Logic reasoner [88] to retrieve the axioms of an ontology causing a detected inconsistency. Finally, Section 8.4 provides a summary of the chapter.

8.1 Version Log Generation

In this section, we discuss the implementation of a plug-in for the Protégé ontology editor to automatically generate a version log. The plug-in inter-

[1] See `http://protege.stanford.edu/`

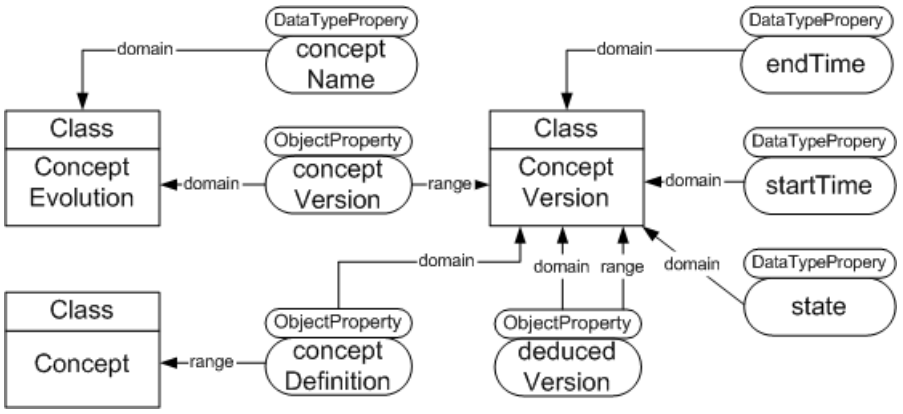

Figure 8.1: Properties of the ConceptEvolution and ConceptVersion Classes

cepts the changes that an ontology engineer applies to the ontology loaded into Protégé, and generates a version log describing the evolution of that ontology. The version log is expressed in terms of a *version ontology*, describing all necessary concepts to express a version log. In Section 8.1.1, we give an overview of the representation of this version ontology. In Section 8.1.2, we discuss the overall architecture and functioning of the plug-in.

8.1.1 Representation

Version logs are represented in terms of instances of the concepts of this version ontology. Remember that a version log describes the state of each concept ever created in an associated ontology at the different moments in time. In Section 4.3.1 on page 75, we introduced a meta-schema of the OWL ontology language. As a version log is intended to describe the various states of concepts over time, we take this meta-schema as basis for the version ontology and extend it to be able to represent different versions of the same concept.

Figure 8.1 gives an overview of the Classes and Properties defined in the evolution ontology to represent different versions of concepts. The Classes *ConceptEvolution* and *ConceptVersion* reflect the definitions of respectively **Concept Evolution** and **Concept Version** introduced in Section 4.1.2 on page 62. A *ConceptEvolution* represents the evolution of a particular concept and keeps track of the current concept name (*conceptName*) and of the different versions of that concept (*conceptVersion*). A *ConceptVersion* stores the definition of a concept (*conceptDefinition*), possible deduced versions (*deducedVersion*), the state of the version (*state*) (pending, confirmed or implemented), and the start and end time of the version (*startTime* and *endTime*).

Furthermore, parts of the meta-model we introduced in Section 4.3.1

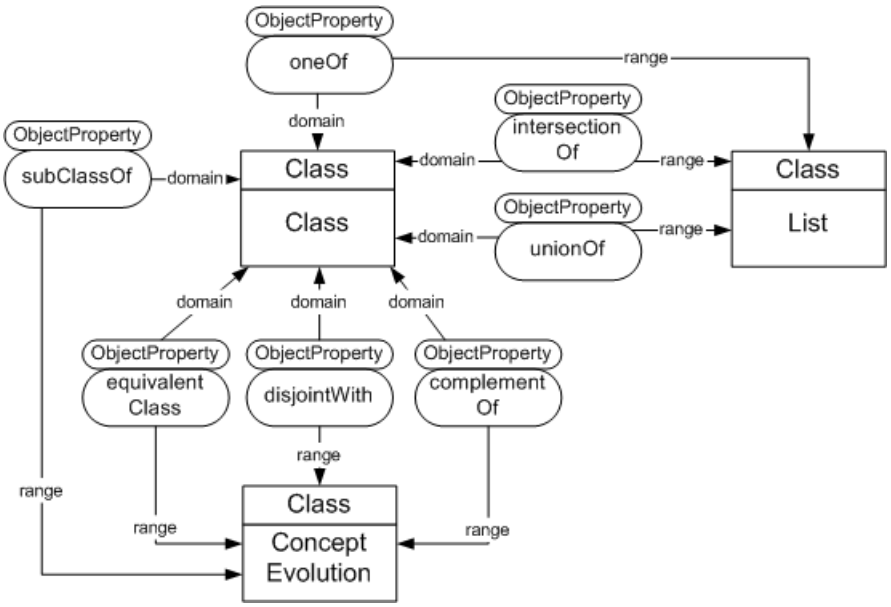

Figure 8.2: Concepts to represent a version of a Class

have to be adapted slightly to be able to correctly represent different versions of concept definitions. The reason for the adaption is that the object of an object Property does not refer to a specific version of that object as this object itself may change. Figure 8.2 gives an overview of the required changes for representing a version of a Class. Instead of linking concepts directly to other concepts, concepts are linked to concept evolutions describing the evolution of a particular concept. E.g., a *Class* is the subclass of a *ConceptEvolution* describing the evolution of another *Class*. The parts of the meta-schema concerned with Properties, Individuals and Restrictions are adapted in a similar way.

8.1.2 Architecture

In this section, we briefly discuss the general architecture and the functioning of the Protégé plug-in to automatically generate a version log. Figure 8.3 gives an overview of the different components of the architecture. Notice that we didn't implement the change request phase of our framework, but rather reused the functionality to modify an ontology already contained in Protégé. The functionality already contained in Protégé suffices to illustrate the generation of a version log and to implement the change detection mechanism (see Section 8.2).

We discuss the components of the architecture by means of Figure 8.3. Whenever an ontology is loaded in Protégé, either a new version log is created for the loaded ontology in the case that no associated version log already

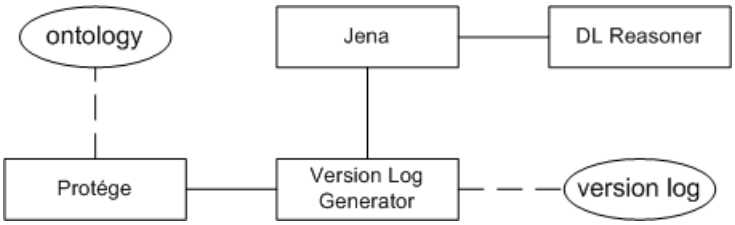

Figure 8.3: Architecture of the version log generator

exists, or otherwise, the associated version log is opened. When changes are applied to the ontology, the Version Log Generator catches the change events thrown by Protégé caused by modifications made to the ontology and updates the version log by creating the appropriate new *EvolutionConcepts* and *EvolutionVersions* representing the new state of the changed concepts. The Version Log Generator relies on the Jena Semantic Web Framework[2] to write the version log to file.

Changes to an ontology may cause previously inferred knowledge to be no longer valid and, the other way round, previously not inferred knowledge may become inferable after the change. As we also want to be able to detect changes to implicit knowledge, we explicitly store this inferred knowledge in the version log. To retrieve implicit knowledge from an ontology, we rely on an external Description Logic reasoner. As we rely on the 'standard' DIG interface[3] to communicate with the Description Logic reasoner, any reasoner supporting this DIG interface can be used for this purpose. Furthermore, we use the Jena framework to retrieve a model of inferable statements from the ontology by means of the plugged-in reasoner. Inferred statements that are not present in current concept versions in the version log are added to the version log, while statements in current concept versions in the version log that are not present in the model of inferred statements are removed.

Figure 8.4 shows a screenshot of the Protégé plug-in to generate a version log. The plug-in also offers a graphical user interface to browse through the different versions of the ontology concepts. The list on the left shows all the concepts ever created in the ontology, the list at the top shows the available versions of a selected concept, while the bottom list shows the details of a selected version.

8.2 Change Detection

In this section, we discuss the implementation of the Change Definition Language and its evaluation to support change detection. Similar to the version log generator, we have chosen to implement it as a plug-in for Protégé. The

[2]See `http://jena.sourceforge.net/`
[3]See `http://dig.sourceforge.net/`

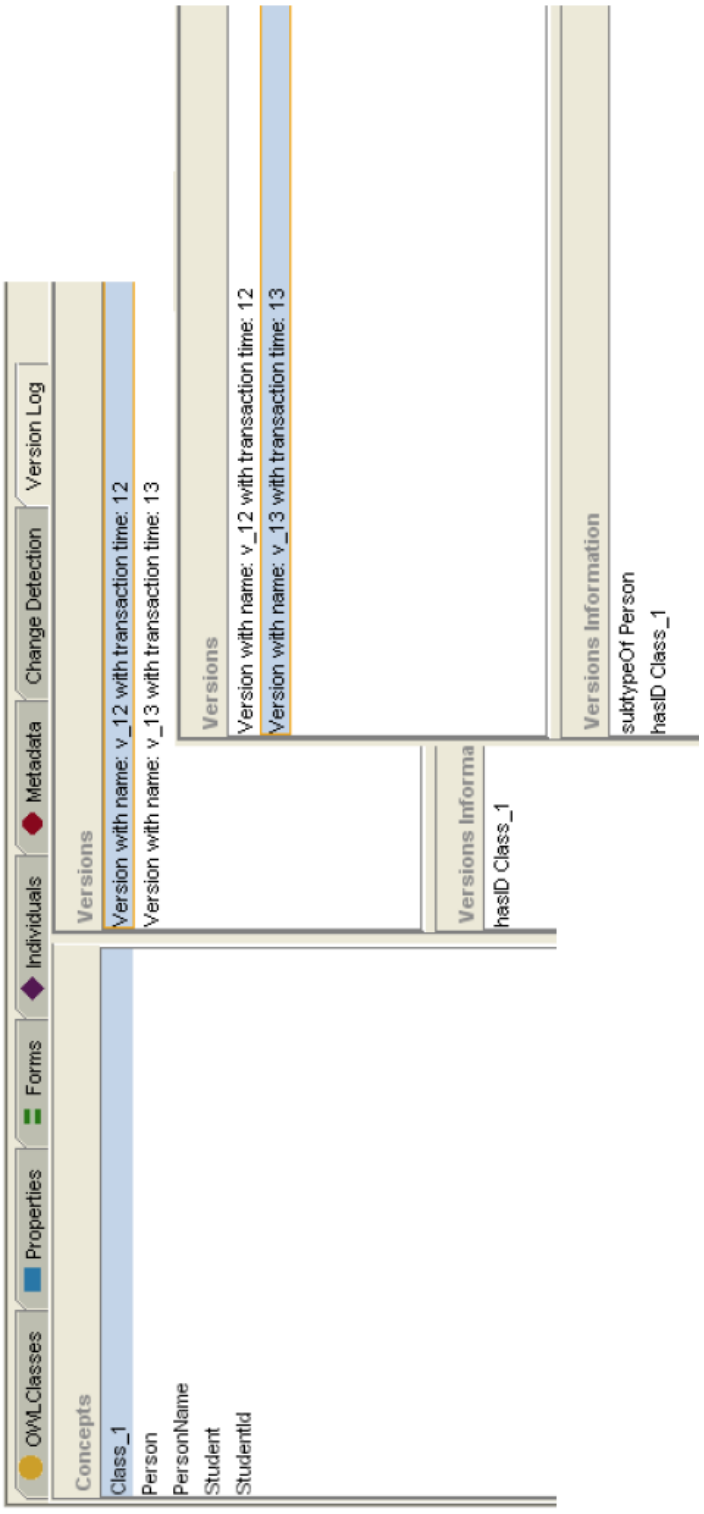

Figure 8.4: Screenshot of the version log plug-in

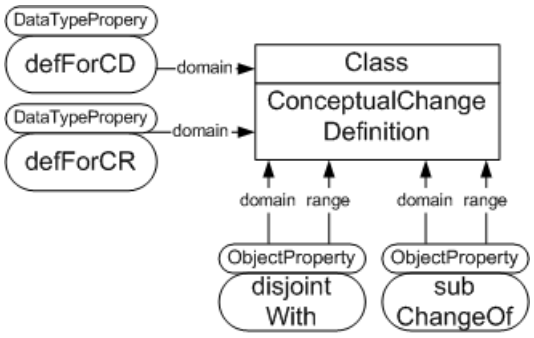

Figure 8.5: Concepts of the change definition ontology

plug-in takes as input a Change Definition Set, containing a set of conceptual change definitions, and a version log, representing the evolution of an ontology. The plug-in evaluates the given conceptual change definitions w.r.t. the given version log and builds up an evolution log. Similar to the version ontology for version logs, we have defined a *change definition ontology* to represent change definition sets and an *evolution ontology* to represent evolution logs. In Section 8.2.1, we give an overview of both ontologies. In Section 8.2.2, we discuss the general architecture of the plug-in.

8.2.1 Representation

In this section, we describe the concepts defined in both the change definition ontology and the evolution ontology. Figure 8.5 gives an overview of the concepts defined in the change definition ontology. A conceptual change definition (*ConceptualChangeDefinition*) is defined in terms of base change definitions for both the purpose of change requests (*defForCR*) and change detection (*defForCD*). Furthermore, a conceptual change definition can be specified to be a subtype of (*subChangeof*) or disjoint with (*disjointWith*) other conceptual change definitions.

Figure 8.6 gives an overview of the concepts defined in the evolution ontology. An occurrence of a change (*OccurrenceOfChange*) is an occurrence of a particular conceptual change definition (*ofConceptualChange*) and instantiates one of the base change definitions of this conceptual change definition (*instantiates*). Such an instantiation stores the header of the base change definition (*baseChangeHeader*) and possibly has a number of parameter bindings (*hasBinding*). A parameter binding consists of the name of the parameter (*parameter*) and a value (*value*).

8.2.2 Architecture

As already mentioned in the introduction of this section, the plug-in takes as input a Change Definition Set defined in terms of the Change Definition

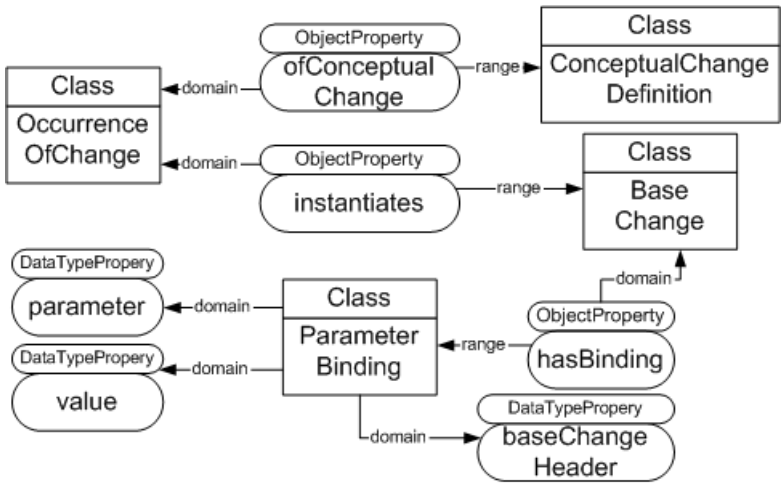

Figure 8.6: Concepts of the evolution ontology

Ontology and a version log. The plug-in builds up an evolution log by evaluating the change definitions in the Change Definition Set. For this purpose, it implements an evaluator for the Change Definition Language. Starting from the EBNF grammar of the Change Definition Language, a parser was generated using the ANTLR parser generator[4]. The parser is used to parse the base change definitions specified in the different conceptual change definitions listed to be used in the change detection phase. An evaluator for the Change Definition Language was implemented that evaluates the base change definitions as temporal queries on the given version log. The plug-in relies on the RDQL implementation of the Jena framework to do the actual querying of the version log, while the implementation of the tense operators is handled by the plug-in itself. The results of the query executions are used to add appropriate occurrences of change to the evolution log. The Jena framework is used to write the evolution log to file.

Figure 8.7 shows a screenshot of the Protégé plug-in implementing the change detection mechanism. The plug-in offers a graphical user interface to select a Change Definition Set and a version log. The list at the bottom gives an overview of the detected changes. The two dropdown boxes on top give the user the possibility to filter the detected changes in the list below by selecting a concept or conceptual change definition.

8.3 Consistency Checking

In this section, we discuss the extension of an existing Description Logic reasoner in order to reveal the axioms of an ontology causing a detected

[4]See `http://www.antlr.org`

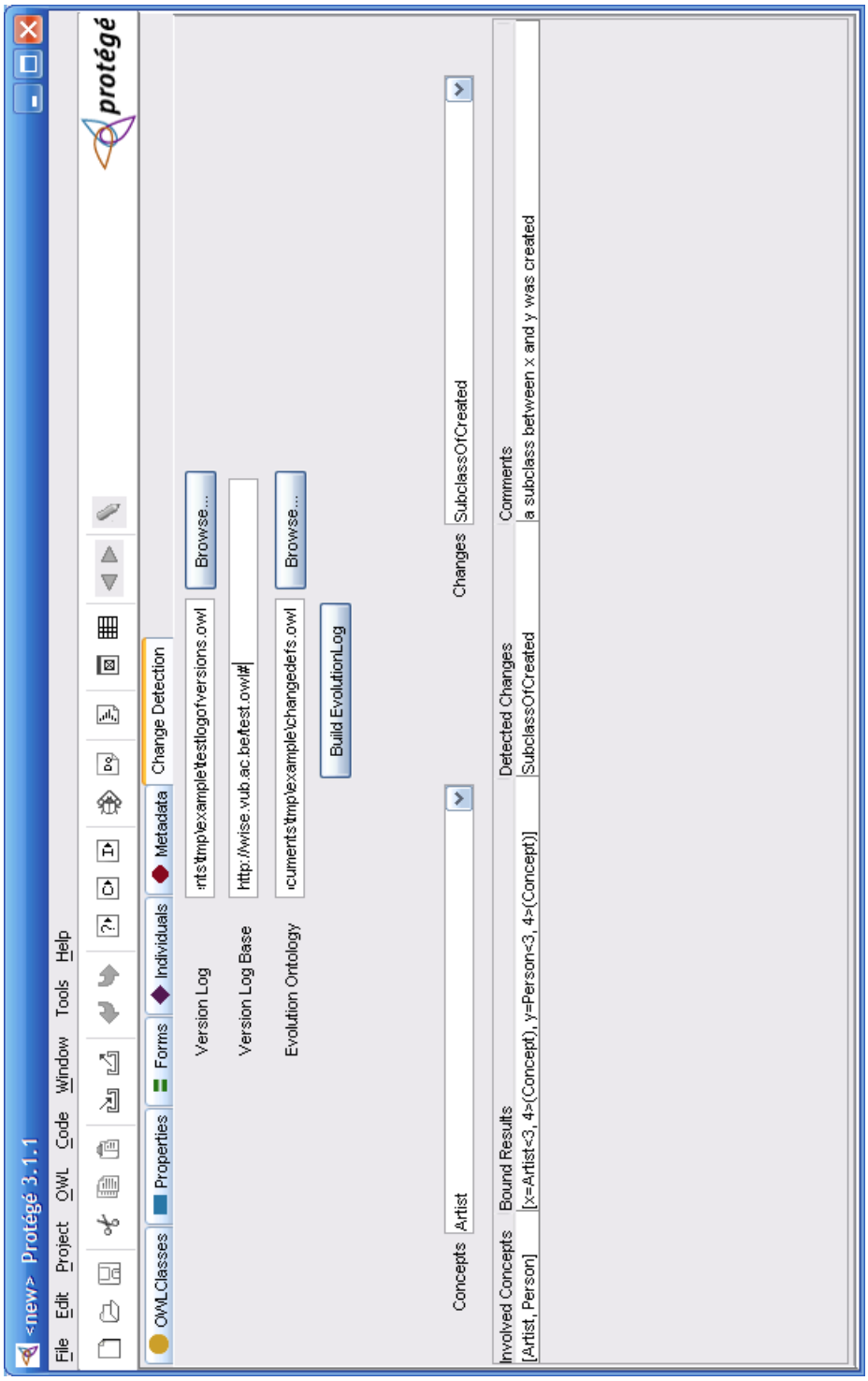

Figure 8.7: Screenshot of the change detection plug-in

inconsistency. We have opted to extend the open source FaCT++ reasoner[5], the successor of the FaCT reasoner. FaCT++ supports the $\mathcal{SHOIQ}(\mathbf{D})$ Description Logic variant which corresponds to OWL DL augmented with qualifying cardinality restrictions, and is written in C++.

As explained thoroughly in Chapter 6, the extended FaCT++ reasoner keeps track of the internal axiom transformations by maintaining an Axiom Transformation Graph. Furthermore, the nodes that are added to a tableau are traced in order to construct a Concept Dependency Tree. The Concept Dependency Trees forms the basis of the selection mechanism of axioms causing a detected inconsistency.

8.4 Summary

In this chapter, we discussed the implementation of the prototype tools that serve as proof of concept for the feasibility of the main ideas presented in this dissertation. Two plug-ins for the Protégé ontology editor were developed: one to automatically generate a version log based on the changes that are applied to an ontology in Protégé, a second plug-in implements a parser and evaluator for the Change Definition Language that allows querying a given version log. We discussed for both plug-ins the internal architecture and representations used. A version log is represented in terms of the version ontology, while a change definition set and an evolution log are represented in terms of respectively the change definition ontology and the evolution ontology. Furthermore, we also extended the existing FaCT++ reasoner in order to pinpoint a set of axioms that form the cause of a detected inconsistency.

[5] See `http://owl.man.ac.uk/factplusplus/`

Chapter 9

Conclusion

In the previous chapter, we have described a number of prototype implementations that serve as proof of concept for the feasibility of the main ideas presented in this dissertation. We discussed the implementation of two Protégé plug-ins and an extension of the FaCT++ Description Logic reasoner. A first Protégé plug-in implements the automatic generation of a version log that describes the evolution of an ontology, while a second plug-in implements a parser and evaluator of the Change Definition Language that allows querying a (generated) version log. Finally, the extension of the FaCT++ reasoner makes it possible to retrieve the exact cause of an inconsistency.

In this final chapter, we seize the opportunity to reflect on the achievements of this dissertation and to look forward to possible new research directions. This chapter is structured as follows. Section 9.1 summarizes the work presented in this dissertation. Section 9.2 discusses the main contributions and achievements of this dissertation, thereby reflecting on the problem statement as described in the introduction (see Section 1.2). Furthermore, section 9.3 lists a number of limitations of our approach. Finally, Section 9.4 presents possible future work.

9.1 Summary

In this dissertation, we have presented a new ontology evolution approach. This ontology evolution approach proposes a framework that consists of a number of phases, each having a specific purpose. The framework allows ontology engineers to request and apply changes to the ontology he manages. The framework assures that the ontology remains consistent after changes are applied so that the ontology evolves from one consistent state into another consistent state. Moreover, the framework also guarantees that the depending artifacts of an ontology remain consistent after that ontology has changed. Furthermore, the framework provides a detailed overview of

the changes that have occurred, supporting different levels of abstraction, different view points and different interpretations. This enhances the comprehensibility of the evolution of an ontology for the ontology engineers as well as for maintainers of depending artifacts.

Before we review the different phases of our ontology evolution framework, we first mention the cornerstones on which the framework is founded. A first key element of our approach is the notion of a version log. A version log has the purpose to describe the evolution of an ontology by keeping track of the different versions of all concepts ever created in an ontology. Important to note is that a version log simply registers the evolution of an ontology, but doesn't form an interpretation of the evolution. For this purpose, the framework offers an evolution log that describes the evolution of an ontology in terms of change definitions. This brings us to another key element of our approach: the Change Definition Language. This Change Definition Language allows users to explicitly capture the semantics of changes they are interested in by formulating conceptual change definitions. The Change Definition Language itself is based on a hybrid-logic approach that is midway between modal-logic and predicate-logic approaches. The tense operators that are included in the language allow users to define changes in terms of differences between current and past versions of concepts.

The conceptual change definitions play a double role in our approach. First of all, they are used by ontology engineers to request the changes they want to apply to an ontology. Ontology engineers request changes by specifying a change request in terms of change definitions to be applied. Secondly, the conceptual change definitions also play a key role in the change detection mechanism offered by our approach. This change detection mechanism allows automatically detecting changes that satisfy the change definitions specified. The detection of these changes is possible because the Change Definition Language is based on a temporal logic i.e., the change definitions can be evaluated as temporal queries on a version log. We discuss both roles of the conceptual change definitions in the following paragraphs.

When an ontology engineer specifies a change request, simply applying the requested changes listed in the change request to an ontology may turn the ontology into an inconsistent state. Our ontology evolution framework ensures that an ontology evolves from one consistent state into another consistent state. It therefore checks whether requested changes introduce inconsistencies. When inconsistencies are found, the approach offers the ontology engineer possible solutions to resolve the inconsistencies. To check for possible inconsistencies in an ontology, our framework relies on OWL reasoners. However, the problem with existing reasoners is that they are built to detect inconsistencies, but don't reveal the cause of the inconsistencies detected. Nevertheless, being able to retrieve the cause of an inconsistency is a necessary condition to be able to resolve the inconsistency. We therefore extended the tableau algorithm on which most state-of-the-art reasoners are based.

We keep track of the internal transformations that occur as a preprocessing step of the tableau algorithm by means of an Axiom Transformation Graph (ATG). Furthermore, we trace the axioms of an ontology that are used by the tableau algorithm to reveal a contradiction, and construct a Concept Dependency Tree (CDT) with these axioms. The CDT allows us to determine the axioms that are causing an inconsistency, while the ATG allows us to retrieve the axioms in their original form. To offer the ontology engineer solutions to resolve a detected inconsistency, we have defined a number of rules. The rules take as input the axioms that are found to cause an inconsistency by the extended tableau algorithm and result in the resolution of the inconsistency. Note that in a given situation, it is possible that more than one rule is applicable. In that case, it is the responsibility of the ontology engineer to select the rule that is most appropriate to solve the inconsistency.

When an ontology changes, this may not only lead to inconsistencies in the ontology itself, but may also cause inconsistencies in the depending artifacts. Due to the decentralized nature of the Web, it becomes not desirable and even impossible to propagate the changes made to an ontology to all its depending artifacts in order to maintain consistency. In the approach taken by our framework, we have redefined dependencies between depending artifacts and ontologies as dependencies between depending artifacts and *specific versions* of ontologies i.e., a depending artifact depends on a specific version of an ontology. Whenever an ontology changes, the consistency of depending artifacts is maintained as they remain depending on a previous version of the ontology. Previous versions of an ontology can be easily retrieved by means of the version log. This allows maintainers of depending artifacts to update at their own pace. However, we also indicated in this dissertation that postponing an update might prevent other depending artifacts from updating as well. We have proposed an approach based on the use of a virtual version log to circumvent this problem. A virtual version log allows a maintainer of a depending artifact to simulate a new version of an ontology it depends on without actually changing that ontology.

The second role of the conceptual change definitions is its use in the change detection mechanism provided by our framework. As mentioned earlier, an evolution log represents just one possible view and interpretation of an ontology evolution. In our approach, an evolution log is not solely populated with changes listed in change requests, instead it may also include changes detected based on the conceptual change definitions given. Due to this change detection mechanism, different users each can create their own evolution log, as different users may define their own set of conceptual change definitions. This provides the opportunity to have different views and interpretations of the same evolution.

Finally, to validate the main ideas presented in this dissertation, a number of prototype tools were developed. We have implemented a plug-in

for the Protégé ontology editor to automatically create a version log. The plug-in is able to capture all changes that are applied to an ontology and automatically generates a version log representing the evolution of the loaded ontology. In a second plug-in for Protégé, a parser and evaluator for the Change Definition Language is implemented. The plug-in is able to take as input a set of conceptual change definitions and a version log, to evaluate the change definitions w.r.t. the given version log and to generate an evolution log forming an interpretation of the ontology evolution. Finally, we have implemented an extension to the FaCT++ Description Logic reasoner in order to retrieve the axioms that are causing a detected inconsistency.

9.2 Contributions

In this section, we discuss the contributions and achievements that are the result of this dissertation by means of the problem statements we formulated in the introduction (see Chapter 1).

Problem 1: Comprehending changes Whenever an ontology evolves, maintainers of depending artifacts that depend on this ontology must be able to get a clear understanding of the changes that have occurred as their decision to update to a new version of the ontology depends for a great part on this understanding. To assist maintainers of depending artifacts in understanding changes, ontology evolution approaches should be able to give a complete overview of the changes that have occurred supporting different levels of abstraction, different viewpoints and even different interpretations.

To assist maintainers of depending artifacts in understanding changes, our approach offers them the possibility to (semi-)automatically generate an evolution log representing the evolution of an ontology in terms of change definitions. Because changes are in our approach formally defined using the Change Definition Language, it becomes possible to detect occurrences of change definitions as change definitions can be evaluated as temporal queries on a version log. The outcome of the change detection process is used to build an evolution log.

To support different levels of abstraction, the Change Definition Language allows to define both primitive and complex changes. Primitive changes give a fine-grained overview of the evolution, while complex changes rather give a more coarse-grained insight in the evolution of an ontology. Furthermore, the Change Definition Language also allows to define meta-changes, which describe the implications of a change rather then the change itself (e.g., *rangeWeakened*), and domain dependent changes, which describe changes in terms of domain concepts

instead of OWL constructs (e.g., *fireEmployee* instead of *deleteIndividual*). Finally, due to the change detection mechanism and the fact that maintainers of depending artifacts can define their own set of change definitions, different interpretations can be associated to the same ontology evolution by generating different evolution logs, each based on one set of change definitions.

Problem 2: Ontology consistency Changes to an ontology may turn the ontology into an inconsistent state. We have extended the tableau algorithm, which most state-of-the-art reasoners rely on to check for consistency, to be able to retrieve the exact axioms that are causing an inconsistency. Furthermore, we have defined a set of rules that ontology engineers can apply to resolve inconsistencies. The rules are applied to the axioms that are causing the inconsistency in order to remove the contradiction that lead to the inconsistency.

Problem 3: Decentralized authority The decentralized nature of the Web imposes a number of problems with respect to ontology evolution. First of all, a decentralized architecture means that changes to an ontology can not be propagated to depending artifacts to maintain consistency. Secondly, the refusal of depending artifacts to update to the latest version of an ontology, may prevent other depending artifacts from updating as well. We say that these depending artifacts are blocked.

To prevent depending artifacts from becoming inconsistent, we redefine dependencies between depending artifacts and ontologies to dependencies between depending artifacts and ontologies at a given moment in time i.e., a depending artifact depends on a specific version of an ontology. When an ontology changes, its depending artifacts remain consistent as they still depend on the old version. Its maintainers can update at a later moment in time, or decide not to update at all. Furthermore, we have provided an answer to the problem of blocked ontologies by introducing a virtual version log. A virtual version log allows a maintainer of a depending artifact to simulate a new version of an ontology it depends on without actually changing that ontology.

Problem 4: Depending artifact consistency In order for a maintainer of a depending artifact to decide whether he wants to update to the latest version of an ontology, or to an intermediate version, or to not update at all, it is important for him to know the consequences of updating to a certain version.

To provide an answer to this problem, we introduced the notion of compatibility requirements that express the requirements a new ontology version should fulfill to be considered backward compatible for

a given depending artifact. Using these compatibility requirements, our approach is able to determine the last backward compatible version of an ontology for a depending artifact. A maintainer knows that updating to a backward compatible version can be done without any consequences for the depending artifact. Finally, when a maintainer of a depending artifact decides to update to a non-backward compatible version, our approach is able to determine the consequences for the depending artifact (when the depending artifact is an OWL ontology). For this purpose, our approach depends on the extended reasoner mentioned in problem statement 2.

9.3 Limitations

As already mentioned in the introduction of this dissertation, the proposed ontology evolution approach does not solve all problems associated with ontology evolution. In this section, we discuss the boundaries and limitations that apply to the work:

- **Limitations of the Change Definition Language:** While the Change Definition Language allows to define a great variety of possible change definitions, its expressiveness is limited. For example, it is currently not feasible to reuse existing change definitions in other definitions, their is no support for explicit quantifiers, and advanced filter expressions (e.g., wildcard expressions) are not part of the Change Definition Language. Furthermore, the language itself consists of rather low-level constructs which may make it not obvious for ontology engineers to define their own change definitions. It might be interesting to look for patterns that are often used in change definitions (e.g., expressions to excluding transitive changes), and include these as more high-level language constructs in the Change Definition Language.

- **No complex dependencies:** In this dissertation, we considered only 'simple' dependencies between ontologies. A simple dependency is represented as a single Property between a concept defined in one ontology and a concept defined in another ontology. However, in various research domains such as ontology alignment and ontology merging there are often complex dependencies needed between ontologies. Complex dependencies are often represented in terms of a mapping ontology that represents a mapping between two or more concepts defined in two ore more different ontologies. The consistency checking and inconsistency resolving approach presented in this approach doesn't support complex dependencies.

- **No support for collaborative development:** We assume in this dissertation that ontologies are developed and maintained by a sin-

gle ontology engineer, or that at least a single ontology engineer has the last word in the management of an ontology. In practice, the development and maintenance of an ontology is often a collaborative activity between different users. Certainly when we take into account that an ontology is a shared conceptualization. The collaborative development and maintenance of ontologies raises issues such as change request negation, conflict resolution of requested changes and concurrency control which are not covered by our approach.

- **Limitations of the implementation:** The tools that have been developed for this dissertation serve as proof of concept for the feasibility of the main ideas presented. They don't cover all aspects of the ontology evolution approach and still have a number of limitations. Consequently, they can not be seen as a full-fledged implementation of the complete ontology evolution framework.

9.4 Future Work

The research presented in this dissertation yields several possibilities for future work. In this section, we discuss possible future work that we envisage. Some future work concerns straightforward elaborations or extensions to the current ontology evolution framework, and mostly arise from the limitations listed in the previous section. Other further work is concerned with applying the ideas presented here to other research areas possibly resulting in new research (sub-)areas.

As already mentioned as a limitation, the presented ontology evolution framework does not support complex dependencies. Complex dependencies between ontologies are often represented by means of a mapping ontology (e.g., the MAFRA (MApping FRAmework for distributed ontologies) ontology [59]). The problem is that this mapping ontology is situated on a different level then the ontologies it interconnects as it doesn't describe a particular domain but rather serves as an extension to an ontology language in order to express complex mappings. As a consequence, such a mapping ontology cannot be regarded the same way as the ontologies it interconnects when maintaining ontology consistency. Instead, the consistency checking and inconsistency resolving approach presented in this dissertation need to be extended to take into account the semantics of the complex mappings.

Somewhat related to the problem of complex dependencies is the problem of importing ontologies. Our ontology evolution approach only supports importing ontologies as a whole, but doesn't allow to import only parts of an ontology. This is not a limitation inherent to our approach as this limitation is the result of the OWL language. OWL only allows importing complete ontologies, not parts of an ontology. Unfortunately, only allowing to import ontologies as a whole has a negative effect on the performance of

reasoning activities as they take all concepts defined in the imported ontology into account. Recently, research has focused on formalisms that allow ontology engineers to partially import concepts from other ontologies. For this purpose, the authors of [55] have proposed the ε-connections framework. Furthermore, the authors of [28] have proposed an extension of the ε-connections formalism to integrate it into OWL. Extending our ontology evolution framework to cope with partial imports by means of ε-connections seems to be appealing future work.

A rather straightforward extension of the ontology evolution framework consists of support for ontology versioning instead of solely ontology evolution. Ontology versioning differs from ontology evolution in the sense that the former deals with creating and managing explicit versions of an ontology, while the latter deals with the adaptation of an ontology without maintaining explicit versions. The implementation phase can be easily extended to create an explicit new version of the ontology for the requested changes. Furthermore, a version log can be associated with each explicit version of an ontology describing the changes since the previous ontology version. To reveal the modifications that have occurred since earlier versions, different version logs must be combined. Moreover, the change detection mechanism should be adapted so that it can work with combinations of version logs.

The work presented in this dissertation also lends itself perfectly to be combined with other research areas. Some interesting future work is presented in the remainder of this section.

The work presented in this dissertation doesn't have to be restricted to the Semantic Web in particular, but may be interesting for the research domain of hypermedia systems in general. Possible future work in this domain concerns the maintenance of link integrity. Hypermedia systems are in general liable to frequent changes: content is often added and removed, the structure of the system may alter and also the presentation is often subject to changes. As a consequence of these changes, links in hypermedia systems quickly become dangling links or point to the wrong information. An interesting research topic would be to investigate whether keeping track of the evolution of a hypermedia system could provide added value, e.g., to allow reducing the number of dangling and wrong-pointing links. A possible option could be to redirect outdated links in the case that information was moved to a different location in the system. Moreover, nodes of the hypermedia system could also be reconstructed in case the requested information is no longer available in the current version of a system.

Other possible future work concerns the domain of adaptive hypermedia systems. Brusilovsky [11] defines adaptive hypermedia systems as all hypertext and hypermedia systems which reflect some features of the user in the user model and apply this model to adapt various visible aspects of the system to the user. As adaptive hypermedia systems are subject to frequent changes, support for evolution is even more insistent then it is the case for

non-adaptive systems. However, research in adaptive hypermedia systems has mainly focused on techniques to monitor user behavior in order to continuously update a user model and adapt the presentation accordingly. As far as we are aware of, no research has been done concerning evolution in adaptive hypermedia systems. Nevertheless, several problems come to mind. One of the problems is that the adaptive behavior of the system can easily lead to an improperly working system (e.g., some information becomes impossible to reach). Another problem is that changes made to the system by its maintainer may cause conditions that were once met to be no longer true. When these conditions have triggered an adaptation, the question arises what should happen to the adapted system. Should the adaptation be reverted? Or does it suffice that a condition was once met to maintain the adaptation?

Furthermore, we also vision interesting future work in the domain of model driven development. Model driven development has gained much attention recently due to OMG's Model Driven Architecture (MDA) initiative[1]. An example of model driven development are model-based hypermedia systems that represent the system in terms of models that capture different aspects of the system (e.g., content, navigation, presentation). WSDM [15], OOHDM [80] and Hera [43] are a few examples of well-known model-based hypermedia systems. When changes are applied to models of the hypermedia system (either applied by the designer or by adaptive behavior), the models of the system run the risk of becoming inconsistent, resulting in a broken hypermedia system. To maintain consistency of the various models, the approach presented in this dissertation could be taken as a starting point. This is especially the case when we take into account that a lot of these model-based hypermedia systems rely on ontologies to represent their models.

As a last possible future work, we envisage a more intelligent ontology evolution approach that is able to learn from actions taken by ontology engineers. In the current approach, the framework offers the ontology engineers a set of rules they can use to resolve inconsistencies. The framework could learn from the rules applied by an ontology engineer in different situations in order to draw up resolution strategies. These resolution strategies can be used by the framework to offer ontology engineers complete and adequate solutions, or even resolve inconsistencies automatically. Another possibility is the scenario where the framework learns patterns in the sequence of changes that are applied by an ontology engineer. For patterns of changes that are often applied, the framework could then automatically define complex changes that define these patterns, and offer them to the ontology engineer.

[1] See `http://www.omg.org/mda/`

Appendix A

Syntax Change Definition Language

```
changeDefinition  ::=  identifier
                       '(' (parameterListH)? ')' body ';'
parameterListH    ::=  parameterH (',' parameterH)*
parameterH        ::=  ('[' role ']')? '?' IDENT
role              ::=  ('subject' | 'old' | 'new')

body              ::=  (':=' | ':<') condition
condition         ::=  expression

expression        ::=  term
term              ::=  factor ('OR' factor)*
factor            ::=  secondary ('AND' secondary)*
secondary         ::=  (primary | 'NOT' primary)
primary           ::=  (statement |
                        parenExpression |
                        tempExp |
                        nativeFunction)
parenExpression   ::=  '(' expression ')'

statement         ::=  identifier ('*')? '(' subject
                       (',' object)? ')'
subject           ::=  (parameter | identifier)
object            ::=  (parameter | identifier | value)
parameter         ::=  '?' IDENT
identifier        ::=  IDENT
value             ::=  '"' IDENT '"'

tempExp           ::=  (unaryTempExp | binaryTempExp)
```

```
unaryTempExp      ::=  '<' (unaryTenseOp | timeRef) '>'
                       parenExpression
binaryTempExp     ::=  '<' binaryTenseOp '>'
                       '(' expression ',' expression ')'
unaryTenseOp      ::=  ('ALWAYS' |
                        'SOMETIME' |
                        'PREVIOUS')
                       ( '(' parameter |
                             identifier ')' )?
binaryTenseOp     ::=  'SINCE' | 'AFTER'
timeRef           ::=  'T' '(' INTEGER ')'

nativeFunction    ::=  nativeID '('
                       nativeArg ','
                       nativeArg ')'
nativeArg         ::=  (parameter | identifier | value)
nativeID          ::=  'equal' | 'lt' | 'gt'

INTEGER           ::=  ('0'..'9')+
IDENT             ::=  ('a'..'z' | 'A'..'Z')
                       ('a'..'z' | 'A'..'Z' | '_' |
                       '0'..'9')*
```

Bibliography

[1] J. Allen. Time and time again: the many ways to represent time. *International journal of intelligent systems*, 6(4):341–356, 1991.

[2] C. Areces, P. Blackburn, and M. Marx. Hybrid logics: characterization, interpolation and complexity. *The Journal of Symbolic Logic*, 66(3):977–1010, 2001.

[3] A. Artale, E. Franconi, and F. Mandreoli. Description logics for modelling dynamic information. In J. Chomicki, R. van der Meyden, and G. Saake, editors, *Proceedings of an outcome of a Dagstuhl seminar on Logics for Emerging Applications of Databases*, pages 239–275. Springer, 2003.

[4] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors. *The Description Logic handbook: Theory, implementation, and applications*. Cambridge University Press, 2003.

[5] F. Baader, I. Horrocks, and U. Sattler. Description logics for the semantic web. *KI – Künstliche Intelligenz*, 16(4):57–59, 2002.

[6] K. Baclawski, C. Matheus, M. Kokar, J. Letkowski, and P. Kogut. Towards a symptom ontology for semantic web applications. In S. McIlraith, D. Plexousakis, and F. van Harmelen, editors, *Proceedings of 3rd International Semantic Web Conference (ISWC '04)*, volume 3298 of *Lecture Notes in Computer Science*, pages 650–667. Springer, 2004.

[7] J. Banerjee, W. Kim, H. Kim, and H. Korth. Semantics and implementation of schema evolution in object-oriented databases. In U. Dayal and I. Traiger, editors, *Proceedings of the 1987 ACM SIGMOD international conference on Management of data (SIGMOD '87)*, pages 311–322. ACM, 1987.

[8] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web: A new form of web content that is meaningful to computers will unleash a new revolution of possibilities. *Scientific American*, 284(5):34–43, 2001.

[9] A. Borgida, R. Brachman, D. McGuinness, and L. Resnick. CLASSIC: A structural data model for objects. In J. Clifford, B. Lindsay, and D. Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data (SIGMOD '89)*, pages 58–67. ACM Press, 1989.

[10] P. Brèche. Advanced principles for changing schemas of object databases. In P. Constantopoulos, J. Mylopoulos, and Y. Vassiliou, editors, *Proceedings of the 8th International Conference on Advances Information System Engineering (CAiSE '96)*, volume 1080 of *Lecture Notes in Computer Science*, pages 476–495. Springer, 1996.

[11] P. Brusilovsky. Methods and techniques of adaptive hypermedia. *User Modeling and User-Adapted Interaction*, 6(2-3):87–129, 1996.

[12] P. Chen. The entity-relationship model: toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[13] J. Chomickie and D. Toman. Temporal logic in information systems. *Logics for databases and information systems*, 98(1):31–70, 1998.

[14] C. Darwin. *The Origin of Species by Means of Natural Selection: Or the Preservation of Favoured Races in the Struggle for Life.* Penguin Books Ltd, July 1982.

[15] O. De Troyer and C. Leune. WSDM: A user centered design method for web sites. *Computer Networks*, 30(1-7):85–94, 1998.

[16] D. Dey, T. Barron, and V. Storey. A conceptual model for the logical design of temporal databases. *Decision support systems*, 15(4):305–321, 1995.

[17] C. Dyreson, F. Grandi, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J. Roddick, N. Sarda, M. Scalas, A. Segev, R. Snodgrass, M. Soo, A. Tansel, P. Tiberio, and G. Wiederhold. A consensus glossary of temporal database concepts. *SIGMOD Record*, 23(1):52–64, 1994.

[18] A. Farquhar, R. Fikes, and J. Rice. The ontolingua server: A tool for collaborative ontology construction. *International Journal of Human-computer Studies*, 46(6):707–727, 1997.

[19] D. Fensel. Relating ontology languages and web standards. In J. Ebert, editor, *Modelle und Modellierungssprachen in Informatik und Wirtschaftsinformatik, Modellierung 2000.* Verlag, 2000.

[20] D. Fensel, I. Horrocks, F. van Harmelen, S. Decker, M. Erdmann, and M. Klein. OIL in a nutshell. In R. Dieng and O. Corby, editors, *Proceedings of the 12th European Workshop on Knowledge Acquisition, Modeling, and Management (EKAW '00)*, volume 1937 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2000.

[21] M. Fernandez, A. Gomez-Perez, and N. Juristo. METHONTOLOGY: From ontological art towards ontological engineering. In *Working Notes of the AAAI Spring Symposium on Ontological Engineering, Stanford University*. AAAI Press, 1997.

[22] F. Ferrandina, T. Meyer, R. Zicari, G. Ferran, and J. Madec. Schema and database evolution in the O2 object database system. In U. Dayal, P. Gray, and S. Nishio, editors, *Proceedings of the 21th International Conference on Very Large Data Bases (VLDB '95)*, pages 170–181. Morgan Kaufmann, 1995.

[23] G. Flouris, D. Plexousakis, and G. Antoniou. On applying the AGM theory to DLs and OWL. In Y. Gil, E. Motta, V. Benjamins, and M. Musen, editors, *Proceedings of the 4th International Semantic Web Conference (ISWC '05)*, volume 3729 of *Lecture Notes in Computer Science*, pages 216–231. Springer, 2005.

[24] E. Franconi, F. Grandi, and F. Mandreoli. A semantic approach for schema evolution and versioning in object-oriented databases. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K. Lau, C. Palamidessi, L. Moniz Pereira, Y. Sagiv, and P. Stuckey, editors, *Proceedings of the 1st International Conference on Computational Logic (CL '00)*, volume 1861 of *Lecture Notes in Computer Science*, pages 1048–1062. Springer, 2000.

[25] M. Genesereth. Knowledge interchange format. In J. Allen, R. Fikes, and E. Sandewall, editors, *Proceedings of the Second International Conference on the Principles of Knowledge Representation and Reasoning (KR '91)*, pages 238–249. Morgan Kaufman, 1991.

[26] M. Genesereth and R. Fikes. Knowledge interchange format, version 3.0, reference manual. Technical Report Logic-92-1, Computer Science Department, Stanford University, 1992.

[27] M. Genesereth and N. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann, September 1987.

[28] B. Grau, B. Parsia, and E. Sirin. Working with multiple ontologies on the semantic web. In S. McIlraith, D. Plexousakis, and F. van Harmelen, editors, *Proceedings of the 3rd International Semantic Web Conference*

(ISWC '04), volume 3298 of *Lecture Notes in Computer Science*, pages 620–634. Springer, 2004.

[29] H. Gregersen and C. Jensen. Temporal entity-relationship models - a survey. *Knowledge and Data Engineering*, 11(3):464–497, 1999.

[30] T. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition, An International Journal of Knowledge Acquisition for Knowledge-Based Systems*, 5(2):199–220, 1993.

[31] P. Haase, A. Hotho, L. Schmidt-Thieme, and Y. Sure. Collaborative and usage-driven evolution of personal ontologies. In A. Gómez-Pérez and J. Euzenat, editors, *Proceedings of the 2nd European Semantic Web Conference (ESWC '05)*, volume 3532 of *Lecture Notes in Computer Science*, pages 486–499. Springer, 2005.

[32] P. Haase and L. Stojanovic. Consistent evolution of OWL ontologies. In A. Gómez-Pérez and J. Euzenat, editors, *Proceedings of the 2nd European Semantic Web Conference (ESWC '05)*, volume 3532 of *Lecture Notes in Computer Science*, pages 182–197. Springer, 2005.

[33] P. Haase, F. van Harmelen, Z. Huang, H. Stuckenschmidt, and Y. Sure. A framework for handling inconsistency in changing ontologies. In Y. Gil, E. Motta, V. Benjamins, and M. Musen, editors, *Proceedings of the 4th International Semantic Web Conference (ISWC '05)*, volume 3729 of *Lecture Notes in Computer Science*, pages 353–367. Springer, 2005.

[34] T. Halpin. *Information modeling and relational databases: from conceptual analysis to logical design.* The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, April 2001.

[35] T. Halpin. ORM 2. In R. Meersman, Z. Tari, and P. Herrero, editors, *Proceedings of On The Move to meaningful Internet systems 2005: OTM 2005 workshops*, volume 3762 of *Lecture Notes in Computer Science*, pages 676–687. Springer, 2005.

[36] J. Heflin and J. Hendler. Dynamic ontologies on the web. In H. Kautz and B. Porter, editors, *Proceedings of the 17th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence*, pages 443–449. AAAI Press / The MIT Press, 2000.

[37] I. Horrocks. The FaCT system. In H. de Swart, editor, *Proceedings of the International Conference Tableaux on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX '98)*, volume 1397 of *Lecture Notes in Computer Science*, pages 307–312. Springer, 1998.

[38] I. Horrocks. DAML+OIL: a description logic for the semantic web. *IEEE Data Engineering Bulletin*, 25(1):4–9, 2002.

[39] I. Horrocks, D. Fensel, J. Broekstra, S. Decker, M. Erdmann, C. Goble, F. van Harmelen, M. Klein, S. Staab, R. Studer, and E. Motta. OIL: The Ontology Inference Layer. Technical Report IR-479, Vrije Universiteit Amsterdam, Faculty of Sciences, 2000.

[40] I. Horrocks and P. Patel-Schneider. Reducing OWL entailment to description logic satisfiability. *Journal of Web Semantics*, 1(4):345–357, 2004.

[41] I. Horrocks, U. Sattler, and S. Tobies. Practical reasoning for expressive description logics. In H. Ganzinger, D. McAllester, and A. Voronkov, editors, *Proceedings of the 6th International Conference on Logic for Programming and Automated Reasoning (LPAR '99)*, volume 1705 of *Lecture Notes in Computer Science*, pages 161–180. Springer, 1999.

[42] I. Horrocks and S. Tobies. Reasoning with axioms: Theory and practice. In A. Cohn, F. Giunchiglia, and B. Selman, editors, *Proceedings of the 7th International Conference on Principles of Knowledge Representation and Reasoning (KR '00)*, pages 285–296. Morgan Kaufmann, 2000.

[43] G. Houben, P. Barna, F. Frasincar, and R. Vdovjak. Hera: Development of semantic web information systems. In J. Lovelle, B. Rodríguez, L. Aguilar, J. Gayo, and M. Ruíz, editors, *Proceedings of the International Conference on Web Engineering (ICWE '03)*, volume 2722 of *Lecture Notes in Computer Science*, pages 529–538. Springer, 2003.

[44] Z. Huang and H. Stuckenschmidt. Reasoning with multi-version ontologies: a temporal logic approach. In Y. Gil, E. Motta, V. Benjamins, and M. Musen, editors, *Proceedings of the 4th International Semantic Web Conference (ISWC '05)*, volume 3729 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 2005.

[45] E. Hyvonen. *Semantic Web kick-off in Finland - Vision, technologies, research, and applications*, volume 2002-001. HIIT Publications, Helsinki Institute for Information Technology, Heslinki, Finland, 2002.

[46] M. Jarrar and S. Heymans. Unsatisfiability reasoning in orm conceptual schemes. In A. Illarramendi and D. Srivastava, editors, *Proceeeding of International Conference on Semantics of a Networked World*, LNCS. Springer, 2005.

[47] M. Jarrar and R. Meersman. Formal ontology engineering in the DOGMA approach. In R. Meersman and Z. Tari, editors, *Proceedings of*

On the Move to Meaningful Internet Systems - DOA/CoopIS/ODBASE Confederated International Conferences DOA, CoopIS and ODBASE, volume 2519 of *Lecture Notes in Computer Science*, pages 1238–1254. Springer, 2002.

[48] A. Kalyanpur, B. Parsia, and E. Sirin. Black box techniques for debugging unsatisfiable concepts. In I. Horrocks, U. Sattler, and F. Wolter, editors, *Proceedings of the 2005 International Workshop on Description Logics (DL '05)*, volume 147 of *CEUR Workshop Proceedings*. CEUR-WS, 2005.

[49] J. Kamp. Tense logic and the theory of linear order. In *Ph.D. Thesis*, 1968.

[50] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42(4):741–843, 1995.

[51] M. Klein. Change management for distributed systems. In *Ph.D. Thesis*, 2004.

[52] M. Klein, D. Fensel, A. Kiryakov, and D. Ognyanov. Ontology versioning and change detection on the web. In A. Gómez-Pérez and V. Benjamins, editors, *Proceedings of the 13th International Conference on Knowledge Engineering and Knowledge Management (EKAW '02)*, volume 2473 of *Lecture Notes in Computer Science*, pages 197–212. Springer, 2002.

[53] M. Klein, A. Kiryakov, D. Ognyanov, and D. Fensel. Finding and characterizing changes in ontologies. In S. Spaccapietra, S. March, and Y. Kambayashi, editors, *Proceedings of the 21th International Conference on conceptual modeling (ER '02)*, volume 2503 of *Lecture Notes in Computer Science*, pages 79–89. Springer, 2002.

[54] M. Klein and N. Noy. A component-based framework for ontology evolution. In F. Giunchiglia, A. Gomez-Perez, A. Pease, H. Stuckenschmidt, Y. Sure, and S. Willmott, editors, *Proceedings of the Workshop on Ontologies and Distributed Systems (IJCAI '03)*. CEUR-WS, 2003.

[55] O. Kutz, F. Wolter, and M. Zakharyaschev. Connecting abstract description systems. In D. Fensel, F. Giunchiglia, D. McGuinness, and F. Williams, editors, *Proceedings of the 8th International Conference of Knowledge Representation and Reasoning (KR '02)*, pages 215–226. Morgan Kaufmann, 2002.

[56] M. Lehman and J. Ramil. Software evolution in the age of component-based software engineering. *IEE Proceedings - Software*, 147(6):249–255, 2000.

[57] D. Lenat and R. Guha. *Building large knowledge-based systems: Representation and inference in the Cyc project*. Addison-Wesley, January 1990.

[58] R. MacGregor and R. Bates. The loom knowledge representation language. Technical Report ISI/RS-87-188, University of Southern California, Information Science Institute, Marina del Rey (CA, USA), 1987.

[59] A. Maedche, B. Motik, N. Silva, and R. Volz. MAFRA - a mapping framework for distributed ontologies. In A. Gómez-Pérez and R. Benjamins, editors, *Proceedings of the 13th Conference on Knowledge Engineering and Knowledge Management, Ontologies and the Semantic Web (EKAW '02)*, volume 2473 of *Lecture Notes in Computer Science*, pages 235–250. Springer, 2002.

[60] A. Maedche, B. Motik, and L. Stojanovic. Managing multiple and distributed ontologies on the semantic web. *The VLDB journal - The International Journal on Very Large Data Bases*, 12(4):286–302, 2003.

[61] A. Maedche, B. Motik, L. Stojanovic, R. Studer, and R. Volz. An infrastructure for searching, reusing and evolving distributed ontologies. In *Proceedings of the 12th International World Wide Web Conference (WWW '03)*, pages 439–448. ACM, 2003.

[62] R. Möller and V. Haarslev. RACER system description. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proceedings of the 1st International Joint Conference on Automated Reasoning (IJCAR '01)*, pages 701–706. Springer, 2001.

[63] B. Motik, A. Maedche, and R. Volz. A conceptual modelling approach for building semantic-driven enterprise applications. In R. Meersman and Z. Tari, editors, *Proceedings of the 1st international conference on ontologies, databases and application of semantics (ODBASE '02)*, volume 2519 of *Lecture Notes in Computer Science*, pages 1082–1099. Springer, 2002.

[64] N. Noy and M. Klein. Ontology evolution: not the same as schema evolution. *Knowledge and information systems*, 6(4):428–440, 2003.

[65] N. Noy, M. Sintek, S. Decker, M. Crubezy, R. Fergerson, and M. Musen. Creating semantic web contents with protege-2000. *IEEE Intelligent Systems*, 16(2):60–71, 2001.

[66] D. Oliver. Change management and synchronization of local and shared versions of a controlled vocabulary. In *Ph.D. Thesis*, 2000.

[67] G. Ozsoyoglu and R. Snodgrass. Temporal and real-time databases: A survey. *Knowledge and Data Engineering*, 7(4):513–532, 1995.

[68] C. Parent, S. Spaccapietra, and E. Zimanyi. Spatio-temporal conceptual models: Data structures + space + time. In C. Bauzer-Medeiros, editor, *Proceedings of the 7th International Symposium on Advances in Geographic Information Systems (ACM-GIS '99)*, pages 26–33. ACM, 1999.

[69] B. Parsia, E. Sirin, and A. Kalyanpur. Debugging OWL ontologies. In A. Ellis and T. Hagino, editors, *Proceedings of the 14th international conference on World Wide Web (WWW '05)*, pages 633–640. ACM, 2005.

[70] J. Penney and J. Stein. Class modification in the GemStone object-oriented DBMS. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-oriented programming systems, languages and applications (OOPSLA '87)*, pages 111–117. ACM, 1987.

[71] R. Peters and M. Özsu. An axiomatic model of dynamic schema evolution in objectbase systems. *ACM Transactions on Database Systems*, 22(1):75–114, 1997.

[72] H. Pinto, C. Tempich, and S. Staab. DILIGENT: towards a fine-grained methodology for distributed, loosely-controlled and evolving engingeering of ontologies. In R. Lopez de Mantaras and L. Saitta, editors, *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI '04)*, pages 393–397, Valencia, Spain, 2004. IOS Press.

[73] P. Plessers and O. De Troyer. Ontology change detection using a version log. In Y. Gil, E. Motta, V. Benjamins, and M. Musen, editors, *Proceedings of the 4th International Semantic Web Conference (ISWC '05)*, volume 3729 of *Lecture Notes in Computer Science*, pages 578–592. Springer, 2005.

[74] P. Plessers and O. De Troyer. Resolving inconsistencies in evolving ontologies. In Y. Sure and J. Domingue, editors, *Proceedings of the 3rd European Semantic Web Conference (ESWC '06)*, volume 4011 of *Lecture Notes in Computer Science*, pages 200–214. Springer, 2006.

[75] P. Plessers, O. De Troyer, and S. Casteleyn. Event-based modeling of evolution for semantic-driven systems. In O. Pastor and J. Cunha, editors, *Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE '05)*, volume 3520 of *Lecture Notes in Computer Science*, pages 63–76. Springer, 2005.

[76] A. Prior. *Papers on Time and Tense*. Oxford university press, 1st edition, January 2003.

[77] A. Rector, N. Drummond, M. Horridge, J. Rogers, H. Knublauch, R. Stevens, H. Wang, and C. Wroe. OWL pizzas: Practical experience of teaching OWL-DL: common errors & common patterns. In E. Motta, N. Shadbolt, A. Stutt, and N. Gibbins, editors, *Proceedings of 14th International Conference on Knowledge Engineering and Knowledge Management (EKAW '04)*, volume 3257 of *Lecture Notes in Computer Science*, pages 63–81. Springer, 2004.

[78] J. Roddick. A survey of schema versioning issues for database systems. *Information and Software Technology*, 37(7):383–393, 1995.

[79] S. Schlobach and R. Cornet. Non-standard reasoning services for the debugging of description logic terminologies. In G. Gottlob and T. Walsh, editors, *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI '03)*, pages 355–362. Morgan Kaufmann, 2003.

[80] D. Schwabe, G. Rossi, and S. Barbosa. Systematic hypermedia application design with OOHDM. In *Proceedings of the 7th ACM Conference on Hypertext and Hypermedia*, pages 116–128. ACM, 1996.

[81] E. Sirin and B. Parsia. Pellet: An OWL DL reasoner. In R. Möller and V. Haarslev, editors, *Proceedings of the 2004 International Workshop on Description Logics (DL '04)*, volume 104 of *CEUR Workshop Proceedings*. CEUR-WS, 2004.

[82] R. Snodgrass and I. Ahn. A taxonomy of time databases. In *Proceedings of the 1985 ACM SIGMOD international conference on Management of data (SIGMOD '85)*, pages 236–246. ACM, 1985.

[83] L. Stojanovic. Methods and tools for ontology evolution. In *Ph.D. Thesis*. University of Karlsruhe, 2004.

[84] L. Stojanovic, A. Maedche, B. Motik, and N. Stojanovic. User-driven ontology evolution management. In A. Gómez-Pérez and V. Benjamins, editors, *Proceedings of the 13th International Conference on Knowledge Engineering and Knowledge Management. Ontologies and the Semantic Web (EKAW '02)*, volume 2473 of *Lecture Notes in Computer Science*, pages 285–300. Springer, 2002.

[85] L. Stojanovic, A. Maedche, N. Stojanovic, and R. Studer. Ontology evolution as reconfiguration-design problem solving. In J. Gennari, B. Porter, and Y. Gil, editors, *Proceedings of the 2nd international conference on Knowledge capture (K-CAP '03)*, pages 162–171. ACM, 2003.

[86] M. Tallis and Y. Gil. Designing scripts to guide users in modifying knowledge-based systems. In *Proceedings of the 16th National Conference on Artificial Intelligence (AAAI/IAAI)*, pages 242–249. AAAI Press / MIT Press, 1999.

[87] D. Tsarkov and I. Horrocks. Efficient reasoning with range and domain constraints. In V. Haarslev and R. Möller, editors, *Proceedings of the 2004 International Workshop on Description Logics (DL '04)*, volume 104 of *CEUR Workshop Proceedings*. CEUR-WS, 2004.

[88] D. Tsarkov and I. Horrocks. FaCT++ description logic reasoner: System description. In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR '06)*, 2006. To appear.

[89] H. Wang, M. Horridge, A. Rector, N. Drummond, and J. Seidenberg. Debugging OWL-DL ontologies: a heuristic approach. In Y. Gil, E. Motta, V. Benjamins, and M. Musen, editors, *Proceedings of the 4th International Semantic Web Conference (ISWC '05)*, volume 3729 of *Lecture Notes in Computer Science*, pages 745–757. Springer, 2005.