Reasoning with Multi-version Ontologies

Zhisheng Huang and Heiner Stuckenschmidt
(Vrije Universiteit Amsterdam)

Abstract.

EU-IST Integrated Project (IP) IST-2003-506826 SEKT

Deliverable D3.5.1(WP3.5)

In this document, we propose a framework for reasoning with multi-version ontology, in which a temporal logic is developed to serve as its semantic foundation. We show that the temporal logic approach can provide a solid semantic foundation which can support various requirements on multi-version ontology reasoning. We have implemented the prototype of MORE (Multi-version Ontology REasoner), which is based on the proposed framework. In this document, we present the design of the interfaces for MORE, and discuss its implementation.

Keyword list: ontology management, ontology versioning, ontology reasoning

SEKT Consortium

This document is part of a research project partially funded by the IST Programme of the Commission of the European Communities as project number IST-2003-506826.

British Telecommunications plc.
Orion 5/12, Adastral Park
Ipswich IP5 3RE
UK
Tel: +44 1473 609583, Fax: +44 1473 609832
Contact person: John Davies
E-mail: john.nj.davies@bt.com

Jozef Stefan Institute
Jamova 39
1000 Ljubljana
Slovenia
Tel: +386 1 4773 778, Fax: +386 1 4251 038
Contact person: Marko Grobelnik
E-mail: marko.grobelnik@ijs.si

University of Sheffield
Department of Computer Science
Regent Court, 211 Portobello St.
Sheffield S1 4DP
UK
Tel: +44 114 222 1891, Fax: +44 114 222 1810
Contact person: Hamish Cunningham
E-mail: hamish@dcs.shef.ac.uk

Intelligent Software Components S.A.
Pedro de Valdivia, 10
28006 Madrid
Spain
Tel: +34 913 349 797, Fax: +49 34 913 349 799
Contact person: Richard Benjamins
E-mail: rbenjamins@isoco.com

Ontoprise GmbH
Amalienbadstr. 36
76227 Karlsruhe
Germany
Tel: +49 721 50980912, Fax: +49 721 50980911
Contact person: Hans-Peter Schnurr
E-mail: schnurr@ontoprise.de

Vrije Universiteit Amsterdam (VUA)
Department of Computer Sciences
De Boelelaan 1081a
1081 HV Amsterdam
The Netherlands
Tel: +31 20 444 7731, Fax: +31 84 221 4294
Contact person: Frank van Harmelen
E-mail: frank.van.harmelen@cs.vu.nl

Empolis GmbH
Europaallee 10
67657 Kaiserslautern
Germany
Tel: +49 631 303 5540, Fax: +49 631 303 5507
Contact person: Ralph Traphöner
E-mail: ralph.traphoener@empolis.com

University of Karlsruhe, Institute AIFB
Englerstr. 28
D-76128 Karlsruhe
Germany
Tel: +49 721 608 6592, Fax: +49 721 608 6580
Contact person: York Sure
E-mail: sure@aifb.uni-karlsruhe.de

University of Innsbruck
Institute of Computer Science
Techikerstraße 13
6020 Innsbruck
Austria
Tel: +43 512 507 6475, Fax: +43 512 507 9872
Contact person: Jos de Bruijn
E-mail: jos.de-bruijn@deri.ie

Kea-pro GmbH
Tal
6464 Springen
Switzerland
Tel: +41 41 879 00, Fax: 41 41 879 00 13
Contact person: Tom Bösser
E-mail: tb@keapro.net

Sirma AI EAD, Ontotext Lab
135 Tsarigradsko Shose
Sofia 1784
Bulgaria
Tel: +359 2 9768 303, Fax: +359 2 9768 311
Contact person: Atanas Kiryakov
E-mail: naso@sirma.bg

Universitat Autonoma de Barcelona
Edifici B, Campus de la UAB
08193 Bellaterra (Cerdanyola del Vallès)
Barcelona
Spain
Tel: +34 93 581 22 35, Fax: +34 93 581 29 88
Contact person: Pompeu Casanovas Romeu
E-mail: pompeu.casanovas@uab.es

Executive Summary

Multiple versions of an ontology can be considered as a temporal sequence of change actions on an ontology.

In this document we will investigate how temporal logics serve as the semantic foundation of multi-version ontology reasoning. We propose a framework of reasoning with multi-version ontologies which is based on a temporal logic approach. We will show that the temporal logic can provide a solid semantic foundation which serve as an extended query language to detect the ontology changes and their consequences.

We have implemented the prototype of MORE (Multi-version Ontology REasoner), which extends existing systems for querying Description Logic Ontologies with temporal operators that support the maintenance of multiple versions of the same ontology. We discuss the implementation of the prototype of MORE.

Contents

Chapter 1

Introduction

When an ontology is changed, the ontology developers may want to keep the older versions of the ontology. Although maintaining multi-version ontologies increases the resource cost, it is still very useful because of the following benefits:

- **Change Recovery**. For ontology developers, the latest version of an ontology is usually less stable than the previous ones, because the new changes have been introduced on it, and those changes and their consequences have not yet been fully recognized and evaluated. Maintaining the previous versions of the ontology would allow the possibilities for the developers to withdraw or adjust the changes to avoid unintended impacts.

- **Compatibility** Ontology users may still want to use an earlier version of the ontology despite the new changes, because they may consider that the functionalities of the earlier version of the ontology are sufficient for their needs. Furthermore, multi-version ontologies may have different resource requirement. Ontology users may prefer an earlier version with less resource requirement to a newer version with higher resource requirement.

The list above is not complete. We are going to discuss more benefits in the next chapter. Those benefits justify to some extent that multi-version ontology management and reasoning systems are really useful. The change recovery requires the system to provide a facility to evaluate the consequences raising from ontology changes and a tool to compare multi-versions of the ontology. Selecting a compatible version needs a system which can support a query language for reasoning on a selected version of the ontology. This requires a query language which can express the temporal aspects of the ontology changes. Intuitively, multiple versions of an ontology can be considered as a temporal sequence of change actions on an ontology. That serves as our departure point in this document. In this document we will investigate how temporal logics serve as the semantic foundation of multi-version ontology reasoning. We propose a framework of reasoning with multi-version ontologies which is based on a temporal logic approach. We

will show that the temporal logic can provide a solid semantic foundation which serves as an extended query language to detect the ontology changes and their consequences.We have implemented the prototype of MORE (Multi-version Ontology REasoner), which extends existing systems for querying Description Logic Ontologies with temporal operators that support the maintenance of multiple versions of the same ontology. We discuss the implementation of the prototype of MORE.

This document is organized as follows: Chapter 2 provides a brief survey on the ontology evolution and versioning. Chapter 3 discusses the problem of multi-version ontology reasoning. Chapter 4 presents a temporal logic for reasoning with multi-version ontologies, shows how the proposed temporal logic can serve as a query language for reasoning with multi-version ontologies. Chapter 5 presents the interface design of MORE. Chapter 6 discusses the implementation issues of MORE, and Chapter 7 discusses further work, and concludes the document.

Chapter 2

Solved and Open Problems in Ontology Evolution

An important area related to the problem of ontology evolution is the area of database schema evolution. In this work a number of basic problems and principles with respect to evolution have been identified that are also applicable in the context of ontology evolution. A basic distinction made in this work is between schema evolution and schema versioning [Roddick1995]. Schema evolution deals with the problem of changing the schema and making sure that the data is still accessible after the change. This includes the propagation of changes to the underlying data in order to adapt them to the new model [Banerjee *et al.*1987]. Versioning, on the other hand aims at maintaining different versions of the schema in parallel and to make sure that the data can be accessed using any of the versions. This problem, however, can be reduced to evolving the schema and keeping the old one as well.

In the following, we summarize some of the basic requirements for schema evolution that have been stated in connection with the problem of schema evolution for object oriented databases that are most relevant for the problem of ontology evolution.

Evolvability   The basic requirement in connection with schema evolution is the availability of a suitable apparatus for evolving the schema. In particular, there are two main aspects of this apparatus. The first one is a set of **change operations** that can be used to modify the schema. In schema evolution, these operations are often specified in terms of an algebra that defines the semantics of individual and composed changes. The second important part is a **structure for representing changes** made to the schema. This representation is used to store information about changes performed and provides the basis for managing multiple versions of the schema, determine the impact of changes on the data and to invert changes if necessary.

Integrity An important aspect of schema evolution is to preserve the integrity of the database during change. As a result of a change operation different kinds of conflicts can appear that may harm integrity. **Syntactic conflicts** may occur for example due to multiply defined attribute names in the same class definition. Further, **semantic conflicts** can appear if changes to the schema break up referential integrity or if the modification of integrity constraints makes it incompatible with another one. In order to guarantee integrity, schema evolution has to provide means for detecting and for resolving syntactic and semantic conflicts that are the result of schema changes.

Compatibility While integrity only concerns the content of the database itself, another important issue is to ensure the compatibility of the database with applications using it even after the schema has changed. The literature mentions two aspects of compatibility: **downward compatibility** means that systems that were based on the old version of the schema can still use the database after the evolution. **Upward compatibility** means that system that are built on top of the new schema can still access the old data. The question of compatibility is tightly linked to versioning as the ability to access data through different versions of the same schema ensures compatibility.

2.1   Ontology Versioning and Evolution

In principle, the issues discussed above are also relevant for the problem of ontology evolution. In the following, we summarize recent work that addressed the different aspects mentioned above for the special case of ontologies.

2.1.1   Evolvability

The evolvability of ontologies has been addressed by different researchers by defining change operations and change representations for ontology languages. Change operations have been proposed for specific ontology languages. In particular change operations have been defined for OKBC, OWL[Klein2004] and for the KAON ontology language [Stojanovic2003]. All approaches distinguish between atomar and complex changes. While atomar changes can be detected on the syntactic level and significantly different between the different languages, complex changes define changes on the conceptual level of the ontology (i.e. insert subclass). The latter type of changes is quite similar across different languages. Different ways of representing ontological changes have been proposed: besides the obvious representation as a change log that contains a sequence of operations, authors have proposed to represent changes in terms of mappings between two versions of the same ontology [Noy and Musen2003]. The latter approach has the advantage that they can be used to access data in different versions.

2.1.2 Integrity

The problem of preserving integrity in the case of changes is also present for ontology evolution. On the one hand, the problem is harder here as ontologies are often encoded using a logical language where changes can quickly lead to logical inconsistency that cannot directly be determined by looking at the change operation. On the other hand, there are logical reasoners that can be used to detect inconsistencies both within the ontology and with respect to instance data. As this kind of reasoning is often costly, heuristic approaches for determining inconsistencies have been proposed [Klein2004, Stuckenschmidt and Klein2003]. While deciding whether an ontology is consistent or not can easily be done using existing technologies, repairing inconsistencies in ontologies is an open problem. However, there is some preliminary work on diagnosing the reasons for an inconsistency which is prerequisite for a successful repair [Schlobach and Cornet2003].

2.1.3 Compatibility

The problem of compatibility with applications that use an ontology has received little attention so far. The diffuclty is that the impact of a change in the ontology on the function of the system is hard to predict and strongly depends on the application that uses the ontology. Part of the problem is the fact that ontologies are often not just used as a fixed structure but as the basis for deductive reasoning. The functionality of the system often depends on the result of this deduction process and unwanted behavior can occur as a result of changes in the ontology. Some attempts have been made to characterize change and evolution multiple versions on a semantic level [Heflin and Pan2004]. This work provides the basis for analyzing compatibility which currently is an open problem.

2.2 Conclusions

We conclude that at the current state of research the problem of defining the basic apparatus for performing ontology evolution in terms of change operations and representation of changes is understood. Open questions with respect to ontology evolution mainly concern the problem of dealing with integrity problems and with ensuring compatibility of the ontology with existing applications. The basic problem that has to address in the context of both of these topic lies in the logical nature of many ontology specifications. The semantics of the underlying logic often causes side effects that are hard to predict because they can occur at any place in the model and can only be detected by deductive reasoning. It is unlikely that syntactic approaches to evolution that only work on the syntactic representation of the ontologies can deal with these problems. We therefore need methods that work at the semantic level and are aware of logical implications caused by the changes. The formal characterization of ontology evolution provided by Heflin is a step in

the right direction, but it does not provide any concrete methods for supporting evolution that are necessary to resolve existing problems with respect to dealing with inconsistency or determining compatibility.

Chapter 3

Multi-Version Reasoning: An Open Problem

The aim of this work is to provide basic support for solving the open problems in ontology evolution, in particular with respect to the problem of compatibility to existing applications. As argued above, in order to support compatibility an analysis of changes on a syntactic and structural level is not sufficient as the function of applications often depends on the result of reasoning processes.

A semantic portal might for example provide a function that lists all people working on a particular topic, say ontology evolution. This functionality can be provided by retrieving all instances of the concept person that are linked to ontology evolution by the works-on relation. Changes in the underlying ontology can significantly hamper this functionality. If for instance the class researcher is defined to no longer be a subclass of person, but of position, people classified as researchers will no longer be shown in the list. Further, if it is decided to no longer represent the relation between topics by the subclass relation but by a special subtopic-of relation for the sake of conceptual correctness, people that defined to work on the topic 'multi-version reasoning' will no longer be in the list of people, because the new relation does not support inheritance reasoning necessary to retrieve the corresponding instances.

Our goal is to provide ontology managers and users with a tool that helps to detect effects like the one described above after an ontology has changed. Another more ambitious goal for the future is to also provide support for predicting such effects before the ontology has actually been changed[Haase *et al.*2005]. In the following sections, we introduce the general idea of providing tool support for this purpose and identify relevant use cases for the technology.

3.1 Multi-Version Management

As mentioned above, versioning is the key to compatibility as it enables each application to use a version of the ontology that best fits its requirements. In the example given above, the portal would just continue to use the old version of the ontology and would still be able to provide the requested functionality. Our work therefore focuses on supporting the **management of different versions of the same ontology on a semantic level**. In particular, we want top provide functionality for answering queries about knowledge derivable from different versions. The corresponding approach has to provide two kinds of functionalities

- Ask questions about statements derivable from a certain version.

- Ask for a version that allows to derive certain statements

While the first kind of functionality can be used to inspect a given version of the ontology in order to find out whether important statements can or cannot be derived from it, the second kinds of functionality helps to find a version that is compatible with a given application because important statements can be derived from it.

Another issue is the scope of the approach, in particular the space of versions to be considered. There are several possible scenarios. In most relevant cases, we are concerned with a history of different versions of the same ontology where each version replaces the previous one. We call this the **retrospective** approach. As a result, we have a sequence of versions. There are also scenarios, in which different versions of the ontology co-evolve. This is mostly the case in scenarios where the development of the ontology is not controlled by an authority. In the following, we ignore this scenario which is less relevant for professional ontology development. The other question is whether future versions are considered to be part of the space of versions. Such future versions can be characterized in terms of changes to an existing version. As mentioned above, different proposals for change representations exist that could be used for this purpose. We call this the **prospective** approach. We believe that including future versions in the scope of the management approach has many benefits as it allows us to answer 'what if' questions that help the ontology engineer to better plan new versions.

In the following we describe application scenarios for the basic version management approach that only considers existing versions as well as for the extended approach that also considers future versions.

3.2 Application Scenarios

The development of our method is based on the assumption that different versions of an ontology are managed on a central server. In a commercial setting, ontologies are normally created and maintained on a development server. Stable versions of the ontology are moved to a production server which publishes the corresponding models and therefore plays the role of the central server. Further Compatangelo et al propose a blackboard architecture [Compatangelo *et al.*2004] that also allows the centralized management of different ontology versions in distributed environments and makes our approach applicable also in the distributed setting. Based on this general assumption, there are a number of quite relevant application scenarios for the version management technology sketched above. In the following, we provide a number of use cases for Multi-version Reasoning including typical relevant questions about the relation between statements in different versions of an ontology.

Use Case 1: Semantic Change Log

The ontology provider wants to inform the users of the ontology about changes in the new version. The idea is that the new version of the ontology is added to the system which automatically computes all changes with respect to a certain facts. A typical case would be that all subsumption relations are checked. The system outputs a list of obsolete subsumption relations and a list of new subsumption relations. Questions connected to this use case are:

1. Are all facts from the old version still derivable ?

2. What facts are not derivable any more ?

3. What new facts are derivable from the new version ?

4. Which parts of the ontology are backward compatible and which not ?

Use Case 2: Compatibility Check

The user of an ontologies is informed about the release of a new version of his ontology. He wants to check in a controlled setting whether the new version conforms to his application needs. The user specifies a set of queries. Each query is accompanied by a list of certain answers that the user wants to be in the result set. In a typical case, this query could be for all instances of a concept. For each query, the system outputs a list of certain answers that are not computed any more plus a set of answers that are results now but have bot been results in the previous version. Typical questions connected to this use case are:

1. What is the impact on the results of a certain query ?

2. Do we still get all the wanted answers for our data ?

3. Is the new ontology compatible with our data ?

Use Case 3: Version Selection

The user needs an ontology with particular properties for his application. He wants to know which version of ontology fits his specific requirements best. For this purpose, the user defines a number of statements that he wants to hold. The systems identifies the latest version of the ontology in which the required statements hold.

1. Which is the last version that can be used to derive certain facts ?

2. Which is the latest version compatible with my data ?

3. Is there a newer version compatible with my data ?

4. which is the latest version of my ontology that allows me to make a certain change without becoming inconsistent ?

Use Case 4: Evolution Planning

Based on customer feedback and requests, the ontology provider wants to determine useful and harmful changes to plan the future evolution of the ontology. This use case concerns the prospective aspect of multi-version reasoning and is out of the scope of this deliverable. Questions that will have to be answered in this setting are the following.

1. Will this change make it possible to derive a certain fact ?

2. What change will make it possible to derive a certain fact ?

3. We need a certain statement to hold, is this still the case after a proposed change ?

4. Which changes are not allowed if I want this fact to be still derivable ?

5. will the ontology be backward compatible after this change (will all the derivable facts still be derivable) ?

6. What changes can I make without becoming inconsistent/incompatible?

3.3   Outline of the Work

In this document, we make a contribution towards a general apparatus for supporting multi-version management on the semantic level. We will do this by defining and implementing a query language that is able to answer some of the relevant questions about multiple versions of the same ontology as identified in the usage scenarios. Rather than trying to directly support all of these queries, be identify a basic machinery that provides the expressive power needed to provide the required functionality, but still needs to be optimized for specific queries. Our solution is partial in the sense that it implements a basic mechanism that does not necessarily cover all of the use cases and questions mentioned above. This deliverable is devoted to the description of this basic mechanism. The approach will be evaluated against the general use cases mentioned above as well as with respect to the concrete use cases in the SEKT project. The results of this evaluation will be reported in a follow-up deliverable. The relation of our work to other work on consistent ontology evolution in SEKT are described in another upcoming deliverable [Haase *et al.*2005].

We base our method on a temporal logic over statements derivable from different versions of the ontology. Queries concerning the content derivable from versions can be stated in this temporal logic and are evaluated using model-checking techniques. In this work we focus on the retrospective approach to multi-version analysis and only sketch a possible extension to the prospective analysis. Concerning the representation of the ontologies being analyzed, our approach will in principle be independent of a particular language. The only requirement is the availability of a reasoner that computes logical implications from a single ontology. In practice, we have designed and implemented our methods to work on the basis of ontologies specified in OWL-DL.

With respect to derivable statements, we mainly consider subsumption between named classes. We want to stress, however, that the approach presented is independent from the representation of the ontologies and can easily be adapted to other representation languages. In the following we first introduce the temporal logic approach to managing multiple versions. Based on this, we define a minimal query language for multiple versions and describe a prototypical implementation of the approach. We conclude with examples of how the prototype can be used to answer queries about different versions of an example ontology.

Chapter 4

A Temporal Logic for Multi-version Ontology Reasoning

Temporal logics can be classified as two main classes with respect to two different time models: linear time model and branching time model. The linear time logics which express properties over a single sequence of states. This view is suitable for the retrospective approach to multi-ontology reasoning where we assume the existence of a sequence of versions. Branching time logics are express properties across different sequences of states. This feature would be needed for the prospective approach where we consider different possible sequences of changes in the future. The linear temporal logic **LTL** is a typical temporal logic for modeling linear time, whereas the computation tree logic **CTL** is a typical one for modeling branching time [Rescher and Urquhart1971, van Benthem1995, Clarke *et al.*1999].

Temporal logics are often future-oriented, because their operators are designed to be ones which involve the future states. Typical operators are: the operator $\mathbf{Future}\,\phi$ which states that 'ϕ holds sometimes in the future with respect to the current state', and the operator $\mathbf{Alwaysf}\,\phi$ which states that 'ϕ always holds in the future with respect to the current state', and the operator $\phi\,\mathbf{Until}\,\psi$ which states that 'ϕ always holds in the future until ψ holds'. For a discrete time model, the operator ϕ is introduced to state that ϕ holds at the next state with respect to the current state. For the retrospective reasoning, we only need a temporal logic that only talks about the past. Namely, it is one which can be used to compare the current state with some previous states in the past. It is natural to design the following past-oriented operators, which correspond with the counterparts of the future oriented temporal operators respectively:

- the previous operator states that a fact ϕ holds just one state before the current state.

- the the sometimes-in-the past operator states that a fact ϕ holds sometimes in the past with respect to the current state.

14

- the always-in-the-past operator states that ϕ holds always in the past with respect to the current state.

In this document, we use a linear temporal logic, denoted as **LTLm**, which actually is a restricted linear temporal logic **LTL** to past-oriented temporal operators.

4.1   Version Spaces and Temporal Models

In the following, we will define the formal semantics for the temporal operators by introducing an entailment relation between a semantic model (i.e., multi-version ontologies) and a temporal formula. We consider a version of an ontology to be a state in the semantic model. We do not restrict ontology specifications to a particular language (although OWL and its description logics are the languages we have in mind). In general, an ontology language can be considered to be a set of formulas that is generated by a set of syntactic rules in a logical language $\mathcal{L}$.

We consider multi-versions of an ontology as a sequence of ontologies which are connected with each other via change operations. Each of these ontologies has a unique name. This is different from the work by [Heflin and Pan2004], who consider that an ontology is one which contains the set of other ontologies which are backwards compatible with it. We have the following definition.

Definition 1 (Version Space) *A version space S over an ontology set Os is a set of ontology pairs, namely, $S \subseteq Os \times Os$.*

We use version spaces as a semantic model for our temporal logic, restricting our investigation to version spaces that present a linear sequence of ontologies:

Definition 2 (Linear Version Space) *A linear version space S on an ontology set Os is a version space which is a finite sequence of ontologies*

$$S = \{\langle o_1, o_2 \rangle, \langle o_2, o_3 \rangle, \cdots, \langle o_{n-1}, o_n \rangle\}$$

Alternatively we write the sequence S as follows:

$$S = (o_1, o_2, \cdots, o_n)$$

We use $S(i)$ to refer the i_th ontology o_i in the space. For a version space $S = (o_1, o_2, \cdots, o_n)$, We call the first ontology $S(1)$ in the space the *initial version of the version space*, and the last ontology $S(n)$ the *latest version of the version space* respectively.

We introduce an ordering $\prec_S$ with respect to a version space S as follows:

Definition 3 (Ordering on Version Space) *$o \prec_S o'$ iff o occurs prior to o' in the sequence S, i.e., $S = (\cdots, o, \cdots, o', \cdots)$.*

Proposition 4.1.1 (Prior version and Linear Ordering)
the prior version relation $\prec_S$ is a linear ordering, namely, $\prec_S$ is
(i) irreflexive, i.e., $(o \nprec_S o)$,
(ii) transitive, i.e., $o \prec_S o'$ and $o' \prec_S o'' \Rightarrow o \prec_S o''$,
(iii) asymmetry, i.e., $o \prec_S o' \Rightarrow o' \nprec_S o$,
(iv) comparable, i.e., either $o \prec_S o'$ or $o' \prec_S o$,
for any ontology o, o', o''.

4.2 Syntax and Semantics of LTLm

The Language $\mathcal{L}+$ for the temporal logic **LTLm** can be defined as an extension to the ontology language $\mathcal{L}$ with Boolean operators and the temporal operators as follows:

$q \in \mathcal{L} \Rightarrow q \in \mathcal{L}+$
$\phi \in \mathcal{L}+ \Rightarrow \neg\phi \in \mathcal{L}+$
$\phi, \psi \in \mathcal{L}+ \Rightarrow \phi \wedge \psi \in \mathcal{L}+$
$\phi \in \mathcal{L}+ \Rightarrow \mathbf{PreviousVersion}\,\phi \in \mathcal{L}+$
$\phi \in \mathcal{L}+ \Rightarrow \mathbf{AllPriorVersions}\,\phi \in \mathcal{L}+$
$\phi, \psi \in \mathcal{L}+ \Rightarrow \phi\,\mathbf{Since}\,\psi \in \mathcal{L}+$

Where the negation $\neg$ and the conjunction $\wedge$ must be new symbols that do not appear in the language $\mathcal{L}$ to avoid the ambiguities. Define the disjunction $\vee$, the implication $\rightarrow$, and the bi-conditional $\leftrightarrow$ in terms of the conjunction and the negation as usual. Define $\bot$ as a contradictory $\phi \wedge \neg\phi$ and $\top$ as a tautology $\phi \vee \neg\phi$ respectively.

Using these basic operators, we can define some additional operators useful for reasoning about multiple versions. We define the **SomePriorVersion** operator in terms of the **AllPriorVersions** operator as

$$\mathbf{SomePriorVersion}\phi =_{df} \neg\mathbf{AllPriorVersions}\,\neg\phi$$

The always-in-the-past **AllPriorVersions** operator is one which does not consider the current state. We can define a strong always-in-the-past **AllVersions** operator as

$$\mathbf{AllVersions}\phi =_{df} \phi \wedge \mathbf{AllPriorVersions}\,\phi,$$

which states that 'ϕ always holds in the past including the current state'.

Let S be a version space on an ontology set Os, and o be an ontology in the set Os, we extend the entailment relation for the extended language $\mathcal{L}+$ as follows:

$$
\begin{array}{lll}
S, o \models q & \text{iff} & o \models q, for\ q \in \mathcal{L}. \\
S, o \models \neg\phi & \text{iff} & S, o \not\models \phi. \\
S, o \models \phi \wedge \psi & \text{iff} & S, o \models \phi, \psi. \\
S, o \models \textbf{PreviousVersion}\,\phi & \text{iff} & \langle o', o \rangle \in S \text{ such that } S, o' \models \phi. \\
S, o \models \textbf{AllPriorVersions}\,\phi & \text{iff} & \text{ for any } o' \text{ such that } o' \prec_S o, S, o' \models \phi. \\
S, o \models \phi\textbf{Since}\psi & \text{iff} & \exists(o_1 \ldots o_i)(\langle o_1, o_2 \rangle, \ldots, \langle o_{i-1}, o_i \rangle \in S \text{ and } o_i = o) \\
& & \text{such that } S, o_j \models \phi \text{ for } 1 \leq j \leq i \text{ and } S, o_1 \models \psi.
\end{array}
$$

For a linear version space S, we are in particular interested in the entailment relation with respect to its latest version of the ontology $S(n)$ in the version space S. We use $S \models \phi$ to denote that $S, S(n) \models \phi$. Model checking has been proved to be an efficient approach for the evaluation of temporal logic formulas[Clarke *et al.*1999]. In the implementation of MORE, we are going to use the standard model checking algorithm for evaluation a query in the temporal logic **LTLm**. Therefore, we do not need a complete axiomatization for the logic **LTLm** in this document.

4.3 Formal Properties

The validity of a temporal formula in the logic **LTLm** is defined as a property which is independent of any particular **LTLm** model and any state in the model. Namely, the property is true in every state of any **LTLm** model. We have the following definition:

Definition 4 (Validity) $\models \phi$ *iff* $S, o \models \phi$ *for any* S, o.

Here is a list of formal properties in the logic **LTLm**:

Proposition 4.3.1 (Formal Properties of Temporal Operators)
(a) $\models$ **AllPriorVersions**$\phi \rightarrow$ **SomePriorVersion**ϕ.
(the always-in-the-past implies the sometimes-in-the-past.)

(b) $\models$ **PreviousVersion**$\phi \rightarrow$ **SomePriorVersion**ϕ.
(the previous implies the sometimes-in-the-past.)

(c) $\models$ **PreviousVersionSomePriorVersion**$\phi \rightarrow$ **SomePriorVersion**ϕ.
(the previous of the sometimes-in-the-past implies the sometimes-in-the-past.)

(d) $\models$ **SomePriorVersionSomePriorVersion**$\phi \rightarrow$ **SomePriorVersion**ϕ.
(idempotent of the sometimes-in-the-past.)

(e) $\models$ **AllPriorVersionsAllPriorVersions**$\phi \wedge$ **PreviousVersion**$\phi \rightarrow$

AllPriorVersionsϕ.
(quasi-idempotent of the always-in-the-past.)

(f) $\models$ **PreviousVersionPreviousVersion**$\phi \rightarrow$ **SomePriorVersion**ϕ.
(the previous of the previous implies the sometimes-in-the-past.)

(g) $\models \phi$**Since**$\psi \rightarrow$ **SomePriorVersion**$\psi \lor \psi$.
(relation bewteen the since operator and the sometimes-in-the-past.)

(h) $\models \phi$**Since**$\psi \rightarrow \phi$.
(ϕ since ψ implies that ϕ holds in the current version.)

(i) $\models \phi \land \psi \rightarrow \phi$**Since**$\psi$.
(trivial case for the since operator.)

4.4 LTLm as a Query Language

There are two types of queries: reasoning queries and retrieval queries. The former concerns with an answer either 'yes' or 'no', and the latter concerns an answer with a particular value, like a set of individuals which satisfy the query formula. Namely, the evaluation of a a reasoning query is a decision problem, whereas the evaluation of a retrieval query is a search problem. In this section, we are going to discuss how we can use the proposed temporal logic to support both reasoning queries and retrieval queries.

4.4.1 Reasoning queries

Using the **LTLm** logic we can formulate reasoning queries over a sequence of ontologies that correspond to the typical questions mentioned in Section3.

Are all facts still derivable? This question can be answered for individual facts using reasoning queries. In particular, we can use the query $\phi \land$ **PreviousVersion** ϕ to determine for facts ϕ derivable from the previous version whether they still hold in the current version. The same can be done for older versions by chaining the **PreviousVersion** operator or by using the operator **AllVersions** to ask whether formulas were always true in past versions and are still true in the current one (**AllVersions** ϕ).

What facts are not derivable any more? In a similar way, we can ask whether certain facts are not true in the new version any more. This is of particular use for making sure that unwanted consequences have been excluded in the new version. The corresponding

query is $\neg\phi \wedge \mathbf{PreviousVersion}\,\phi$. Using the $\mathbf{AllPriorVersions}$ operator, we can also ask whether a fact that was always true in previous versions is not true anymore.

Are the facts are newly derivable from the new version?   Reasoning queries can also be used to determine whether a fact is new in the current version. As this is true if it is not true in the previous version, we ca use the following query for checking this $\phi \wedge \neg\mathbf{PreviousVersion}\,\phi$. We can also check whether a new fact never held in previous versions using the following query $\phi \wedge \neg\mathbf{SomePriorVersion}\,\phi$.

What is the last version that can be used to derive certain facts?   Using reasoning queries we can check whether a fact holds in a particular version. As versions are arranged in a linear order, we can move to a particular version using the $\mathbf{PreviousVersion}$ operator.   The query $\mathbf{PreviousVersion}\,\mathbf{PreviousVersion}\,\phi$ for instance checks whether ϕ was true in the version before the previous one. The query $\phi\mathbf{Since}\psi$ states that ϕ always holds since ψ holds in a prior version.

A drawback of reasoning queries lies in the fact, that they can only check a property for a certain specific fact. When managing a different versions of a large ontology, the user will often not be interested in a particular fact, but ask about changes in general. This specific functionality is provided by retrieval queries.

4.4.2   Retrieval Queries

Many Description Logic Reasoners support so-called retrieval queries that return a set of concept names that satisfy a certain condition. For example, a children concept c' of a concept c, written $child(c,c')$, is defined as one which is subsumed by the concept c, and there exists no other concepts between them. Namely,

$$child(c,c') =_{df} c' \sqsubseteq c \wedge\ \not\exists c''(c' \sqsubseteq c'' \wedge c'' \sqsubseteq c \wedge c'' \neq c \wedge c'' \neq c').$$

Thus, the set of new/obsolete/invariant children concepts of a concept on an ontology o in the version space S is defined as follows:

$$new_{children}(S,o,c) =_{df} \{c'|S,o \models child(c,c') \wedge \neg\mathbf{PreviousVersion}\,child(c,c')\}.$$
$$obsolete_{children}(S,o,c) =_{df} \{c'|S,o \models \neg child(c,c') \wedge \mathbf{PreviousVersion}\,child(c,c')\}.$$
$$invariant_{children}(S,o,c) =_{df} \{c'|S,o \models child(c,c') \wedge \mathbf{PreviousVersion}\,child(c,c')\}.$$

The same definitions can be extended into the cases like parent concepts, ancestor concepts, descendant concept and equivalent concepts. Those query supports are sufficient to evaluate the consequences of the ontology changes and the differences among multi-version ontologies.

4.5   Making version-numbers explicit

Temporal logics allow us to talk about the temporal aspects without reference to a particular time point. For reasoning with multi-version ontologies, we can also talk about the temporal aspects without mentioning a particular version name. We know that each state in the temporal logic actually corresponds with a version of the ontology. It is not difficult to translate the temporal statements into a statement which refers to an explicit version number. Here are two approaches for it: relative version numbering and absolute version numbering.

4.5.1   Relative version numbering

The proposed temporal logic is designed to be one for past-oriented. Therefore, it is quite natural to design a version numbering which is relative to the current ontology in the version space. We use the formula $\textbf{Version}_0\phi$ to denote that the property holds in the current version. Namely, we refer to the current version as the version 0 in the version space, and other states are used to refer to a version relative to the current version, written as $\textbf{Version}_{-i}$ as follows:

$$\textbf{Version}_0\phi =_{df} \phi.$$

$$\textbf{Version}_{(-i)}\phi =_{df} \textbf{PreviousVersion}(\textbf{Version}_{(1-i)}\phi).$$

The formula $\textbf{Version}_{-i}\phi$ can be read as "the property ϕ holds in the previous i-th version".

4.5.2   Absolute version numbering

Given a version space S with n ontologies on it, i.e., $|S| = n - 1$. For the latest version $o = S(n)$, it is well reasonable to call the i-th ontology $S(i)$ in the version space the version i of S, denoted as $\textbf{Version}_{i,S}$. Namely, we can use the formula $\textbf{Version}_{i,S}\phi$ to denote that the property ϕ holds in the version i in the version space S. Thus, we can define the absolute version statement in terms of a relative version statement as follows:

$$\textbf{Version}_{(i,S)}\phi =_{df} \textbf{Version}_{(i-n)}\phi.$$

Explicit version numbering provides the basis for more concrete retrieval queries. In particular, we now have the opportunity to compare the children of a concept c in two specific ontologies i and j in the version space S. The corresponding definitions are the following:

$$newChildren(S,c)_{i,j} =_{df} \{c'|S \models \textbf{Version}_{(i,S)}\,child(c,c') \wedge \neg\textbf{Version}_{(j,S)}\,child(c,c')\}.$$

$obsoleteChildren(S, c)_{i,j} =_{df} \{c' | S \models \neg\mathbf{Version}_{(i,S)} \, child(c, c') \wedge \mathbf{Version}_{(j,S)} \, child(c, c')\}.$

$invariantChildren(S, c)_{i,j} =_{df} \{c' | S \models \mathbf{Version}_{(i,S)} \, child(c, c') \wedge \mathbf{Version}_{(j,S)} \, child(c, c')\}.$

Again, the same can be done for other predicates like parent-, ancestor or descendant concepts.

Chapter 5

Interfaces for Multi-version Ontology Reasoners

In this chapter we propose a simple API for Multi-version Ontology Reasoners (MORE). The MORE interface is designed to be independent of any particular platform and any application. Thus, we take the similar ideas from SOAP[1] which has been built message protocols using XML on top of HTTP, and the DIG Description Logic Interface[Bechhofer *et al.*2003] in which TELL request and ASK requests are designed for asserting and querying on ontologies. The DIG Description tion Logic Interface is supported by many description logic reasoning systems, like Racer[Haarslev and Möller2001] and Fact[Horrocks1999]. Thus, it makes the implementation of the prototype of MORE convenient. We will discuss the details in Chapter 6.

5.1 Query Language on a Single Ontology

As discussed in Chapter 4, multi-version ontology reasoning language $\mathcal{L}+$ is designed to be an extension to a language which is based on single ontology reasoning language $\mathcal{L}$. We would not limit the single ontology language to any particular one. It can be the DIG Description Logic Language and any other one which has been built message protocols using XML, like RDF and OWL[McGuinness and van Harmelen2004].

5.2 TELL Language

In the MORE interface, the TELL language is designed to be a natural extension to the TELL requests in the DIG interface for multi-version ontology reasoning. However, note

---

[1]http://www.w3.org/TR/soap/

that the extension to the DIG interface would not hinder its independence of any particular reasoning language.

In MORE, a TELL request contains a tellm (i.e., the tell request in MORE) element in its body. The following shows an example of TELL request which contains an ontology specification in the DIG format.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<tellm xmlns="http://wasp.cs.vu.nl/sekt/more/lang"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://wasp.cs.vu.nl/sekt/more/more.xsd">
     <ontology name="bird1" language="dig">
        <defconcept name="bird"/>
        <defconcept name="animal"/>
        <defconcept name="fly"/>
        <defconcept name="eagle"/>
        <impliesc>
          <catom name="bird"/>
          <catom name="animal"/>
        </impliesc>
        <impliesc>
          <catom name="bird"/>
          <catom name="fly"/>
        </impliesc>
        <impliesc>
          <catom name="eagle"/>
          <catom name="bird"/>
        </impliesc>
     </ontology>
</tellm>
```

A version space statement can be used to specify the ontology pairs in the version space inside a tellm statement as shown below:

```
<versionSpace name="bird">
  <pair ontology1="bird1" ontology2="bird2"/>
  <pair ontology1="bird2" ontology2="bird3"/>
</versionSpace>
```

An ontology pair can be removed from a space by a pair removal statement inside a tellm statement like this:

```
<versionSpace name="bird1">
<nopair ontology1="bird1-2" ontology2="bird1-3"/>
</versionSpace>
```

It is up to the users to maintain a version space to make it linear or networked. Thus, this interface is general enough to cover different scenarios of ontology versioning.

The tellm body can contain the data with the OWL format. The following is an example of the tellm statement for OWL with a cluster statement:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<tellm xmlns="http://wasp.cs.vu.nl/sekt/more/lang"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://wasp.cs.vu.nl/sekt/more/more.xsd">
  <ontology name="bird1-2"  language="owl">
   <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   xmlns:owl="http://www.w3.org/2002/07/owl#">
       <owl:Class rdf:ID="eagle">
         <rdfs:subClassOf>
           <owl:Class rdf:about="bird"/>
         </rdfs:subClassOf>
       </owl:Class>
       <owl:Class rdf:ID="bird">
        <rdfs:subClassOf rdf:resource="animal"/>
       </owl:Class>
       <owl:Class rdf:ID="bird">
        <rdfs:subClassOf rdf:resource="fly"/>
       </owl:Class>
       <owl:Class rdf:ID="penguin">
         <rdfs:subClassOf rdf:resource="bird"/>
           <owl:disjointWith rdf:resource="fly"/>
       </owl:Class>
    </rdf:RDF>
 </ontology>
 <versionSpace name="bird1">
   <pair ontology1="bird1-1" ontology2="bird1-2"/>
 </versionSpace>
</tellm>
```

An ontology can be told by a statement which refers to its URL like this:

```
<ontology name="bird3"  language="owl"
   url="file:ontology/bird/bird3.owl"/>
```

We can also specify multi-version ontologies in one tellm request like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<tellm xmlns="http://wasp.cs.vu.nl/sekt/more/lang"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://wasp.cs.vu.nl/sekt/more/more.xsd">
   <ontology name="bird1" language="dig">
       <defconcept name="bird"/>
       <defconcept name="animal"/>
       <defconcept name="fly"/>
       <defconcept name="eagle"/>
```

```
        <impliesc>
          <catom name="bird"/>
          <catom name="animal"/>
        </impliesc>
        <impliesc>
          <catom name="bird"/>
          <catom name="fly"/>
        </impliesc>
        <impliesc>
          <catom name="eagle"/>
          <catom name="bird"/>
        </impliesc>
    </ontology>
    <ontology name="bird2"  language="owl">
     <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
      xmlns:owl="http://www.w3.org/2002/07/owl#">
        <owl:Class rdf:ID="eagle">
          <rdfs:subClassOf>
            <owl:Class rdf:about="bird"/>
          </rdfs:subClassOf>
        </owl:Class>
        <owl:Class rdf:ID="bird">
         <rdfs:subClassOf rdf:resource="animal"/>
        </owl:Class>
        <owl:Class rdf:ID="bird">
         <rdfs:subClassOf rdf:resource="fly"/>
        </owl:Class>
        <owl:Class rdf:ID="penguin">
          <rdfs:subClassOf rdf:resource="bird"/>
            <owl:disjointWith rdf:resource="fly"/>
        </owl:Class>
      </rdf:RDF>
     </ontology>
     <ontology name="bird3"  language="owl"
       url="file:ontology/bird/bird3.owl"/>
  <versionSpace name="bird">
   <pair ontology1="bird1" ontology2="bird2"/>
   <pair ontology1="bird2" ontology2="bird3"/>
  </versionSpace>
</tellm>
```

The summary of the tellm statements and its semantics of the operation is shown in Figure 5.1.

Ontology Specification	<ontology name=o language=L> Ontology </ontology>	$o := Ontology$
Ontology Specification by URL	<ontology name=o language=L url=URL/>	$o := URL$
Version Space Specfication	<versionSpace name=S > <pair ontology1=o1 ontology2=o2/> </versionSpace>	$S := S \cup \{\langle o1, o2 \rangle\}$
Version Space Modification	<versionSpace name=S > <nopair ontology1=o1 ontology2=o2/> </versionSpace>	$S := S - \{\langle o1, o2 \rangle\}$

Figure 5.1: TELL language

5.3 Ask Language

Similarly, the ASK language in the MORE interface is designed to be a natural extension to the ASK requests in the DIG interface for multi-version ontology reasoning. An ASK request in the MORE is specified as an askm element which contains several querym statements which specify queries with respect to an ontology in a version space, like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<askm xmlns="http://wasp.cs.vu.nl/sekt/more/lang"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://wasp.cs.vu.nl/sekt/more/more.xsd">
    <querym id="sat bird on bird1-1"
        versionSpace="bird1" ontology="bird1-1">
      <query>
        <satisfiable>
            <catom name="bird"/>
        </satisfiable>
      </query>
    </querym>
</askm>
```

The querym statement can contain a query which corresponds with a temporal logic formula or other query statements which are encoded in XML format in MORE. The variants of the querym statement and its semantics is shown in Figure 5.2.

5.3.1 Queries on Temporal Aspects

The temporal logic language **LTLm** can be encoded in XML, which is shown in Figure 5.3.

Query on Ontology with versionSpace	<querym id=ID versionSpace=S ontology=o> Query </querym>	$S, o \models Query$
Query on versionSpace	<querym id=ID versionSpace=S> Query </querym>	$S \models Query$

Figure 5.2: Queries on Version Space

Query on Single Ontology	<query> Atomic Query </query>	q
Negation	<not> Q </not>	$\neg\phi$
Conjunction	<and> Q1 $\cdots$ Qn </and>	$\phi_1 \wedge \cdots \wedge \phi_n$
Disjunction	<or> Q1 $\cdots$ Qn </or>	$\phi_1 \vee \cdots \vee \phi_n$
Implication	<implies> Q1 Q2 </implies>	$\phi_1 \rightarrow \phi_2$
If and only if	<iff> Q1 Q2 </iff>	$\phi_1 \leftrightarrow \phi_2$
Previous	<previousVersion> Q </previousVersion>	$\mathbf{PreviousVersion}\phi$
Always in the Past	<allPriorVersions> Q </allPriorVersions>	$\mathbf{AllPriorVersions}\phi$
Sometimes in the Past	<somePriorVersion> Q </somePriorVersion>	$\mathbf{SomePriorVersion}\phi$
Always	<allVersions> Q </allVersions>	$\mathbf{AllVersions}\phi$
Since	<since> $Q_1 Q_2$ </since>	$\phi_1\mathbf{Since}\phi_2$

Figure 5.3: Query Language on Multi-version Ontologies

satisfiability	$<$satisfiable$>$ C $<$/satisfiable$>$	C is satisfiable
subsumption	$<$subsumes$>$ C_1 C_2 $<$/subsumes$>$	$C_2 \sqsubseteq C_1$
disjoint	$<$disjoint$>$ C_1 C_2 $<$/disjoint$>$	$C_2 \sqcap C_1 = \bot$
instance	$<$instance$>$ I C $<$/instance$>$	$C(I)$

Figure 5.4: Atomic Query in DIG1.0

An atomic query in MORE is any query in DIG which can return a Boolean value, i.e., $true$, or $false$. Figure 5.4 shows a list of query patterns in DIG1.0 which can be used as an atomic query in MORE.

Below there is an example of the query which corresponds with the following formula in the temporal logic:

$$S \models \text{satisfiable}(penguin) \wedge \textbf{PreviousVersion}\text{satisfiable}(penguin).$$

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<askm xmlns="http://wasp.cs.vu.nl/sekt/more/lang"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://wasp.cs.vu.nl/sekt/more/more.xsd">
<querym id="querym example2" versionSpace="bird1">
  <and>
    <query>
      <satisfiable>
        <catom name="penguin"/>
      </satisfiable>
    </query>
    <previous>
      <satisfiable>
          <catom name="penguin"/>
      </satisfiable>
    </previous>
  </and>
</querym>
</askm>
```

5.3.2 Ontology Comparison

Queries on ontology comparison like those on new/obsolete/invariant concept relations, can be expressed as the corresponding XML-encoded statements. Here is an example of queries on what are new children concept relations with respect to the concept 'bird' on the ontology 'bird2', compared with the ontology 'bird1':

```xml
<?xml version="1.0" encoding= "ISO-8859-1" ?>
<askm xmlns="http://wasp.cs.vu.nl/sekt/more/lang"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xsi:schemaLocation="http://wasp.cs.vu.nl/sekt/more/more.xsd">
<querym id="new child concepts of the concept bird on bird2 compared bird1"
     versionSpace="bird"  ontology="bird2">
  <newOnConcept  concept="bird" type="children"
     comparedOntology="bird1"/>
</querym>
</askm>
```

If the query does not contain a specific ontology to make comparisons with, the ontology will be compared to with the previous one (in the linear sequence of the version space).

```
<querym id="new child concepts of the concept bird on bird2 on versionSpace"
     versionSpace="bird"  ontology="bird2">
   <newOnConcept  concept="bird" type="children"/>
</querym>
```

The compared concepts can be specified as a list in the query as shown below:

```
<querym id="new child concepts for a concept list on bird2"
    versionSpace="bird"  ontology="bird2">
  <newOnConcept type="children">
    <catom name="bird"/>
    <catom name="fly"/>
  </newOnConcept>
</querym>
```

If there is no particular concept stated, that means the query should be carried on all concepts in the ontology:

```
<querym id="new child concepts on bird2"
    versionSpace="bird"  ontology="bird2">
  <newOnConcept type="children"/>
</querym>
```

If there is no stated type in the query, that means the query should be carried on all concept relations, namely, on children/parents/ancestors/descendants concept relations[2].

```
<querym id="new child concepts on bird2"
   versionSpace="bird"  ontology="bird2">
<newOnConcept/>
</querym>
```

Obsolete/Invariant concept relations can be stated in queries similarly:

```
<querym id="obsolete child concepts on bird2"
    versionSpace="bird"  ontology="bird2">
```

_____

[2]However, note that it costs much time for a large scale ontology.

New concept	\<querym id=ID versionSpace=S ontology=o\> \<newOnConcept concept=c type=Type/\> \</querym\>	$new_{Type}(S, o, c)$
Obsolete concept	\<querym id=ID versionSpace=S ontology=o\> \<obsoleteOnConcept concept=c type=Type/\> \</querym\>	$obsolete_{Type}(S, o, c)$
Invariant concept	\<querym id=ID versionSpace=S ontology=o\> \<invariantOnConcept concept=c type=Type/\> \</querym\>	$obsolete_{Type}(S, o, c)$
New concept on all concepts	\<querym id=ID versionSpace=S ontology=o\> \<newOnConcept type=Type/\> \</querym\>	$new_{Type}(S, o, c)$ for all c
Obsolete concept on all concepts	\<querym id=ID versionSpace=S ontology=o\> \<obsoleteOnConcept type=Type/\> \</querym\>	$obsolete_{Type}(S, o, c)$ for all c
Invariant concept on all concepts	\<querym id=ID versionSpace=S ontology=o\> \<invariantOnConcept type=Type/\> \</querym\>	$invariant_{Type}(S, o, c)$ for all c
New concept without a type	\<querym id=ID versionSpace=S ontology=o\> \<invariantOnConcept/\> \</querym\>	$new_{type}(S, o, c)$ for all c and for all $type$
Obsolete concept without a type	\<querym id=ID versionSpace=S ontology=o\> \<obsoleteOnConcept/\> \</querym\>	$obsolete_{type}(S, o, c)$ for all c and for all $type$
Invariant concept without a type	\<querym id=ID versionSpace=S ontology=o\> \<invariantOnConcept/\> \</querym\>	$invariant_{type}(S, o, c)$ for all c and for all $type$
New concept compared with arbitrary ontology	\<querym id=ID versionSpace=S ontology=o\> \<newOnConcept concept=c type=Type comparedOntology=o'/\> \</querym\>	New $type$ concept of c in o, compared with o'
New concept relation for a concept list	\<querym id=ID versionSpace=S ontology=o\> \<newOnConcept type=Type \> \<catom name = c_1\> $\cdots$ \<catom name = c_n/\> \< /newOnConcept\> \</querym\>	$new_{Type}(S, o, c)$ for all $c \in$ $\{c_1, \cdots, c_n\}$

where $type$ =children/parents/ancestors/descendants

Figure 5.5: Query patterns on concept comparison

```
<obsoleteOnConcept/>
</querym>
<querym id="invariant child concepts on bird2"
   versionSpace="bird"  ontology="bird2">
<invariantOnConcept/>
</querym>
```

Some typical query patterns on ontology concept comparison are shown in Figure 5.5

5.3.3   Version Retrieval

A version retrieval query is an element 'ontologies' which contains a temporal formula in its body. The following is an example of a version retrieval query which asks for a set of ontologies o in the versionspace $bird$ such that $bird, o \models$ satisfiable(fly):

```
<askm xmlns="http://wasp.cs.vu.nl/sekt/more/lang"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://wasp.cs.vu.nl/sekt/more/more.xsd">
<querym id="version retrieval for satisfiable fly"
   versionSpace="bird">
  <ontologies>
    <query>
     <satisfiable>
      <catom name="fly"/>
     </satisfiable>
    </query>
  </ontologies>
</querym>
</askm>
```

If an ontology o' is stated in a querym, that means that the query asks for a set of the ontologies o which are prior to o' or $o = o'$ such that the temporal formula ϕ holds in the ontology o.

```
<askm xmlns="http://wasp.cs.vu.nl/sekt/more/lang"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://wasp.cs.vu.nl/sekt/more/more.xsd">
<querym id="version retrieval for satisfiable
     fly before ontology bird3"
 versionSpace="bird" ontology="bird3">
  <ontologies>
    <query>
     <satisfiable>
      <catom name="fly"/>
     </satisfiable>
    </query>
  </ontologies>
</querym>
</askm>
```

A response to a version retrieval query can be like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<response>
  <ontologySet id="version retrieval for
      satisfiable fly before ontology bird3">
    <ontology name="bird3"/>
    <ontology name="bird2"/>
  </ontologySet>
</response>
```

5.3.4   Relative and Absolute Version Numbering

A temporal reasoning query can be stated with a relative version number or an absolute version number N. If $N > 0$, that means that the query is one with an absolute verison number. If $N < 0$, the query is one with a relative version number (with respect to the current ontology). If $N = 0$, the temporal reasoning query will be evaluated with respect to the current ontology o. The following is an example with two queries, a query with a relative version number and a query with an absolute version number:

```
<askm xmlns="http://wasp.cs.vu.nl/sekt/more/lang"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://wasp.cs.vu.nl/sekt/more/more.xsd">
<querym id="sat penguin on bird2 relative version -1"
  versionSpace="bird"  ontology="bird2">
    <version number="-1">
      <query>
        <satisfiable>
          <catom name="penguin"/>
        </satisfiable>
      </query>
    </version>
</querym>
<querym id="sat penguin absolute version 3"
  versionSpace="bird">
    <version number="3">
      <query>
        <satisfiable>
          <catom name="penguin"/>
        </satisfiable>
      </query>
    </version>
</querym>
</askm>
```

The query patterns on version retrieval and version numbering are shown in Figure 5.6.

Version Retrieval	<querym id=ID versionSpace=S> <ontologies> Query </ontologies> </querym>	$\{o\|S, o \models Query\}$
Relative Version Numbering	<querym id=ID versionSpace=S ontology=o> <version number=N> Query </version> </querym>	$S, o \models \mathbf{Version}_N Query$ where $N < 0$
Absolute Version Numbering	<querym id=ID versionSpace=S> <version number=N> Query </version> </querym>	$S \models \mathbf{Version}_{(N,S)} Query$ where $N > 0$

Figure 5.6: Query patterns on version retrieval and version numbering

5.4   Response Language

The response to an $askm$ request contains in its body a $response$ element. Each response has an attribute id which corresponds to the identifier of a submitted query, like those in the DIG response format.

This is an exmaple of the response which returns a boolean value to a query:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<response>
<true id="sat bird on bird1 (true)"/>
</response>
```

The answer in a response to an ontology comparison query $querym$ contains in its body an $answer$ element which states the comparison results, like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<response>
<answer id="new on concept for a concept list on bird2">
  <conceptRelated concept="bird">
    <descendants>
      <catom name="penguin"/>
    </descendants>
  </conceptRelated>
<conceptRelated concept="fly">
   <descendants>
    <catom name="penguin"/>
```

Error	$<$error id=ID $/>$	error
Boolean	$<$true id=ID $/>$	true
Boolean	$<$false id=ID $/>$	false
Concept Comparison Non-empty Result	$<$answer id=ID$>$ $<$conceptRelated concept=C1 $>$ $<type>\ C_{1,1}, \cdots, C_{1,n_1}\ </type>$ $<$/conceptRelated$>$ $\cdots$ $<$conceptRelated concept=Cm $>$ $<type>\ C_{m,1}, \cdots, C_{m,n_m}\ </type>$ $<$/conceptRelated$>$ $<$/answer$>$	$type(C1, C_{1,1}) \wedge$ $\cdots \wedge type(C1, C_{1,n_1})$ $\wedge$ $\cdots$ $\wedge type(C1, C_{m,1})$ $\wedge \cdots type(C1, C_{m,n_m})$
Concept Comparison Empty Result	$<$answer id=ID$>$ NIL $<$/answer$>$	$\emptyset$
Version Retrieval	$<$ontologySet id=ID$>$ $<$ontology name=O1$>$ $\cdots$ $<$ontology name=On$>$ $<$/ontologySet$>$	$\{O1, \cdots, On\}$

Figure 5.7: Response Language in MORE

```
    </descendants>
 </conceptRelated>
</answer>
</response>
```

Figure 5.7 shows a list of the response pattern in MORE.

Chapter 6

Prototype of MORE

6.1 Implementation

We implemented a prototypical reasoner for multi-version ontologies called MORE based on the approach described above. The system is implemented as an intelligent interface between an application and state-of-the art description logic reasoners and provides server-side functionality in terms of an XML-based interface for uploading different versions of an ontology and posing queries to these versions. Requests to the server are analyzed by the main control component that also transforms queries into the underlying temporal logic queries if necessary. The main control element also interacts with the ontology repository and ensures that the reasoning components are provided with the necessary information and coordinates the information flow between the reasoning components. The actual reasoning is done by model checking components for testing temporal logic formulas that uses the results of an external description logic reasoner for answering queries about derivable facts in a certain version.

An overview of the MORE architecture is shown in Figure 6.1. It has the following components:

- **MORE Server**: The MORE server acts as a server which deals with requests from other ontology applications.

- **Main Control Component**: The main control component performs the main processing, like query analysis, query pre-processing, and interacting with the ontology repositories.

- **Model Checking Component**: the model checking component provides the facilities to evaluate the queries and the system specification.

- **DIG Client**: MORE's DIG client is the standard XDIG client, which calls external
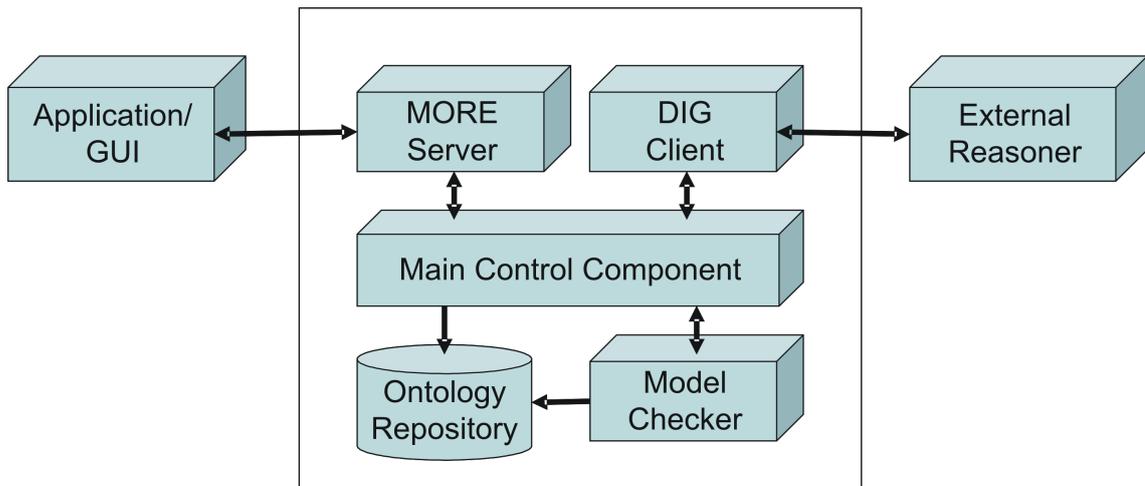
Figure 6.1: Architecture of MORE.

DL reasoners which support the DIG interface to obtain the standard DL reasoning capabilities.

- **Ontology Repositories**: The ontology repositories are used to store the multi-version ontologies and other system specifications.

The MORE prototype is implemented in Prolog and uses the XDIG interface [Huang and Visser2004], an extended DIG description logic interface for Prolog[1]. MORE is designed to be a simple API for a general reasoner with multi-version ontologies. It supports extended DIG requests from other ontology applications or other ontology and metadata management systems and supports multiple ontology languages, including OWL and DIG[Bechhofer *et al.*2003][2]. This means that MORE can be used as an interface to any description logic reasoner as it supports the functionality of the underlying reasoner by just passing requests on and provides reasoning functionalities across versions if needed. Therefore, the implementation of MORE will be independent of those particular applications or systems.

6.2   Functionalities

The current version of the MORE prototype (version 1.0.0) has the following characteristics:

- **Temporal Reasoning Queries**: supports the logic **LTLm**.

---

[1]http://wasp.cs.vu.nl/sekt/dig
[2]http://dl.kr.org/dig/

- **Ontology Comparision Queries**: supports new/obsolete/invariant concept queries with respect to children/parent/ancestor/descendant concept relations.

- **Versioning Queries** : supports version retrieval, relative version numbering, and absolute version numbering.

- **Ontology Languages**: supports the DIG format as well as the OWL language. Ontological data in the OWL format is translated automatically by the XDIG component 'owl2dig'[3].

6.3   Installation and Test Guide

1. **Download**: The MORE package is available from the MORE website:

   `http://wasp.cs.vu.nl/sekt/more/`

   Unzip the MORE package into a directory. We will call the directory $MORE\_ROOT$.

2. **Installation of SWI-Prolog**: MORE requires that SWI-Prolog (version 5.4.7 or higher) has been installed on your computers. It can be downloaded from the SWI-Prolog website:

   `http://www.swi-prolog.org`

   Install SWI-Prolog into a directory. We will call that directory $SWIPROLOG\_ROOT$.

3. **Installation of XDIG**: You can find the zip file 'diglibrary.zip', the XDIG libraries in the directory $MORE\_ROOT$. Unzip the file 'diglibrary.zip' into the SWI-Prolog library directory, i.e., $SWIPROLOG\_ROOT/library$.

4. **Installation of Racer**: MORE requires Racer (version 1.7.14 or higher) as the external DL reasoner. Other DL reasoners may work for MORE if they support the DIG DL interface, however, they have not yet been tested. Racer can be downloaded from the Racer website:

   `http://www.sts.tu-harburg.de/~r.f.moeller/racer/download.html`

The MORE testbed 'more_test.htm' is a MORE client with a graphical interface, which is designed as a webpage. Therefore it can be launched from a web browser which supports Javascript. A screenshot of the MORE testbed, is shown in Figure 6.2.

---

[3]However, note that the component 'owl2dig' is still under development. The complete specification of OWL DL is not yet supported .

Figure 6.2: MORE testbed.

The current version of the MORE testbed supports tests for which both the MORE server (with default port: 8003) and the external DL reasoner (with the default port: 8000) are running on the localhost. The default hostname of the MORE server and the external DL reasoner is 'localhost'. For a MORE server which runs on a remote host, change the host and port data in the 'moremain.htm' file. Namely, replace the URL 'http://127.0.0.1:8003' with another valid URL.

Before starting the MORE test, make sure that the MORE server and the external DL reasoner (i.e. Racer) are running at the host with the correct ports.

1. **Launch Racer**: Racer can be launched by the following command: racer -http 8000 Alternatively, click on the file 'racer8000.bat' if the MORE download package includes the Racer reasoner.

2. **Launch MORE server**: click on the file '$more\_server.pl$' in the MORE directory. If you encounter the global stack limit problem because of a big amount of test data, you should increase the size of the global stack. The windows users can edit the path setting of 'plwin.exe' in the file 'moreserver_bigGlobalStack.bat', then launch it.

The TELLm and ASKm request data can be copied into the TELL text area and the ASK text area respectively. After that, you can click on the buttons 'Tell' and 'Ask' to make the corresponding requests. The request data can also be posted from any application without using the MORE testbed. That is useful if the test data exceeds the text area limit.

6.4 Experiments with MORE

We have tested the current implementation of the MORE system on different versions of real life ontologies from different domains. In the following, we briefly report experiments we performed on detecting changes in the concept hierarchy of the following two ontologies.

The OPJK Ontology The OPJK Ontology (Ontology of Professional Judicial Knowledge) is developed within the SEKT Project [4] by the Autonomous University of Barcelona (UAB) and iSOCO (with the collaboration of the Spanish General Council of the Judiciary) to support an intelligent FAQ (Iuriservice). This system will provide the relevant answers through semantic matching to queries posed in natural language by Spanish judges in their first appointment [Benjamins *et al.*2004, Casanovas *et al.*2005]. We used five different versions of the ontology from different stages of the development process. Each of these version contains about 80 concepts and 60 relations.

---

[4]http://www.sekt-project.com/

Results for the BioSAIL Ontology									
Version(from)	Version(to)	NC	OC	NP	OP	NA	OA	ND	OD	Total
BioSAILv16	BioSAILv20	136	10	123	49	228	104	227	32	909
BioSAILv20	BioSAILv21	54	1	42	21	193	32	192	1	536

Results for the OPJK Ontology									
Version(from)	Version(to)	NC	OC	NP	OP	NA	OA	ND	OD	Total
ontoRDF	ontoRDF2	82	25	53	10	141	16	141	74	542
ontoRDF2	ontoRDF3	82	17	49	13	144	17	144	21	487
ontoRDF3	oplk	49	43	36	20	70	20	54	85	377
oplk	opjk	4	7	2	1	8	6	8	18	54

NC = New Children concept relation, OC = Obsolete Children concept relation, NP = New Parent concept relation, OP = Obsolete Parent concept relation, NA = New Ancestor concept relation, OA = Obsolete Ancestor concept relation, ND = New Descendant concept relation, and OD = Obsolete Descendant concept relation.

Figure 6.3: MORE Tests on Concept Relations

The BiosSAIL Ontology The BioSAIL Ontology which was developed within the BioSTORM project[5]. It has been used in earlier experiments on change management reported in [Klein2004]. the complete data set consists of almost 20 different versions of the ontology. We take three versions of the BioSAIL ontology for the tests reported below. Each version of BioSAIL ontology has about 180 classes and around 70 properties.

Those two ontologies have been tested with different temporal reasoning queries. We concentrated on retrieval queries about the structure of the concept hierarchy. In particular, we used retrieval queries with explicit version numbering as introduced in section 4.5. Below we show the results for the queries about the new and obsolete child, parent, ancestor, and descendant relations in the concept hierarchy.

It has to be noted that the result are not the result of a syntactic analysis of the concept hierarchy, but rely on description logic reasoning. This means that we also detect cases where changes in the definition of a concept lead to new concept relations that are only implicit in the Ontology. The results of these queries can be found at `http://wasp.cs.vu.nl/sekt/more/test/`. Further, arbitrary temporal queries using the operators introduced in this document can be formulated and executed. The only limitation is the interface to the underlying DL reasoner, that currently is only implemented for queries about the concept hierarchy. This can easily be extended to any functionality provided by the RACER system [Haarslev and Möller2001]. A list of the template queries for temporal reasoning queries are available at the MORE testbed, which can be downloaded from the MORE website.

---

[5]http://smi-web.stanford.edu/projects/biostorm/

In this document, we just briefly report experiments we performed on detecting changes in the concept hierarchy. A full test and evaluation of MORE will be reported in the sequel SEKT deliverable D3.5.2 entitled "Evaluation of MORE".

Chapter 7

Discussion and Conclusions

In this document, we discussed the integrated management of multiple versions of the same ontology as an open problem with respect to ontology change management. We proposed an approach for multi-version management that is based on the idea of using temporal logic for reasoning about commonalities and differences between different versions. For this purpose, we define the logic **LTLm** that consists of operators for reasoning about derivable statements in different versions. We show that the logic can be used to formulate typical reasoning and retrieval queries that occur in the context of managing multiple versions. We have implemented a prototypical implementation of the logic in terms of a reasoning infrastructure for ontology-based systems and successfully tested it on real ontologies.

Different from most previous work on ontology evolution and change management our approach is completely based on the formal semantics of the ontologies under consideration. This means that our approach is able to detect all implications of a syntactic change. In previous work, this could only be done partially in terms of ontologies if changes and heuristics that were able to predict some, but not all consequences of a change. Other than previous work on changes at the semantic level which were purely theoretical, we have shown that out approach can be implemented on top of existing reasoners and is able to provide answers in a reasonable amount of time. In order to be able to handle large ontologies with thousands of concepts, we have to think about optimization strategies. Existing work on model checking has shown that these methods scale up to very large problem sets if optimized in the right way. This makes us optimistic about the issue of scalability.

One of the reasons for the efficiency of the approach is the restriction to the retrospective approach, that only considers past versions. This restriction makes linear time logics sufficient for our purposes. A major challenge is the extension of our approach with the prospective approach that would allow us to reason about future versions of

ontologies. This direction of work is challenging, because it requires a careful analysis of a minimal set of change operators and their consequences. There are proposals for sets of change operators, but these operators have never been analyzed form the perspective of dynamic temporal logic. The other problem is that taking the prospective approach means moving from linear to branching time logic which has a serious impact on complexity and scalability of the approach.

The approach in this document is based on the linear temporal logic **LTLm** in which version names are not explicitly expressed in temporal formulas, but implicitly stated by relative/absolute version numbering. Sometimes it is more convenient and useful if version names can be explicitly stated in temporal formulas. Part of the future work is to extend the current temporal logic with nominals (i.e., version names) to support that functionality. The temporal logics with nominals are usually called hybrid logics[Blackburn and Tzakova1999, Blackburn2000].

The implementation of MORE will be integrated with SEKT work in the ontology evolution and change[Haase *et al.*2004, Haase *et al.*2005], so that it can be used to support multi-version reasoning under the ontology evolution and changes. We will report the detailed evaluation on multi-version reasoning for ontology evolution and changes in the sequel SEKT deliverable D3.5.2 entitled "Evaluation of MORE".

Bibliography

[Banerjee *et al.*1987] J. Banerjee, W. Kim, H.-J. Kim, and H. F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *SIGMOD Record (Proc. Conf. on Management of Data)*, number 3 in 16, pages 311–322, 1987.

[Bechhofer *et al.*2003] Sean Bechhofer, Ralf Möller, and Peter Crowther. The dig description logic interface. In *International Workshop on Description Logics (DL2003)*. Rome, September 2003.

[Benjamins *et al.*2004] V.R. Benjamins, J. Contreras, M. Blázquez, L. Rodrigo, P. Casanovas, and M. Poblet. The sekt use legal case components: ontology and architecture. In T.B. Gordon, editor, *Legal Knowledge and Information Systems*, pages 69–77. IOS Press, Amsterdam, 2004.

[Blackburn and Tzakova1999] P. Blackburn and M. Tzakova. Hybrid languages and temporal logic. *Logic Journal of the IGPL*, 7(1):27–54, 1999.

[Blackburn2000] P. Blackburn. Representation, reasoning, and relational structures: a hybrid logic manifesto. *Logic Journal of the IGPL*, 8(3):339–365, 2000.

[Casanovas *et al.*2005] P. Casanovas, M. Poblet, N. Casellas, J-J. Vallbé, F. Ramos, V.R. Benjamins, M. Blázquez, J. Contreras, and J. Gorronogoitia. Legal scenario. case study intelligent integrated decision support for legal professionals. Project Report Report D10.2.1, SEKT, 2005.

[Clarke *et al.*1999] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[Compatangelo *et al.*2004] E. Compatangelo, W. Vasconcelos, and B. Scharlau. Managing ontology versions with a distributed blackboard architecture. In *Proceedings of the 24th Int Conf. of the British Computer Societys Specialist Group on Artificial Intelligence (AI2004)*. Springer-Verlag, 2004.

[Haarslev and Möller2001] Volker Haarslev and Ralf Möller. Description of the racer system and its applications. In *Proceedings of the International Workshop on Description Logics (DL-2001)*, pages 132–141. Stanford, USA, August 2001.

[Haase *et al.*2004] Peter Haase, York Sure, and Denny Vrandecic. Ontology management and evolution - survey, methods and prototypes. Project Report D3.1.1, SEKT, 2004.

[Haase *et al.*2005] Peter Haase, Frank van Harmelen, Zhisheng Huang, Heiner Stuckenschmidt, and York Sure. A framework for handling inconsistency in changing ontologies. Project report, SEKT, 2005.

[Heflin and Pan2004] J. Heflin and Z. Pan. A model theoretic semantics for ontology versioning. In *Third International Semantic Web Conference*, pages 62–76, Hiroshima, Japan, 2004. Springer.

[Horrocks1999] I. Horrocks. Fact and ifact. In *Proceedings of the International Workshop on Description Logics (DL'99)*, pages 133–135, 1999.

[Huang and Visser2004] Zhisheng Huang and Cees Visser. Extended dig description logic interface support for prolog. Deliverable D3.4.1.2, SEKT, 2004.

[Klein2004] M. Klein. *Change Management for Distributed Ontologies*. Phd thesis, Vrije Universiteit Amsterdam, 2004.

[McGuinness and van Harmelen2004] D. McGuinness and F. van Harmelen. Owl web ontology language. Recommendation, W3C, 2004. http://www.w3.org/TR/owl-features/.

[Noy and Musen2003] N.F. Noy and M.A. Musen. The prompt suite: Interactive tools for ontology merging and mapping. *International Journal of Human-Computer Studies*, 59(6):983–1024, 2003.

[Rescher and Urquhart1971] N. Rescher and A. Urquhart. *Temporal Logic*. Springer-Verlag, 1971.

[Roddick1995] J. F. Roddick. A survey of schema versioning issues for database systems. *Information and Software Technology*, 37(7):383–393, 1995.

[Schlobach and Cornet2003] S. Schlobach and R. Cornet. Non-standard reasoning services for the debugging of description logic terminologies. In *Proceedings of the International Joint Conference on Artificial Intelligence - IJCAI'03*, Acapulco, Mexico, 2003. Morgan Kaufmann.

[Stojanovic2003] L. Stojanovic. *Methods and Tools for Ontology Evolution*. Phd thesis, University of Karlsruhe, 2003.

[Stuckenschmidt and Klein2003] H. Stuckenschmidt and M. Klein. Integrity and change in modular ontologies. In *Proceedings of the International Joint Conference on Artificial Intelligence - IJCAI'03*, Acapulco, Mexico, 2003. Morgan Kaufmann.

[van Benthem1995] Johan van Benthem. Temporal logic. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 4, pages 241–350. Oxford, Clarendon Press, 1995.