Reasoning with Multi-version Ontologies: A Temporal Logic Approach

Zhisheng Huang and Heiner Stuckenschmidt

AI Department, Vrije Universiteit Amsterdam, The Netherlands
{huang, heiner}@cs.vu.nl

Abstract. In this paper 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. We have tested MORE with several realistic ontologies. In this paper, we also discuss the implementation issues and report the experiments with MORE.

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 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 section. Those benefits can justify to some extent that multi-version ontology management and reasoning systems are really useful. The change recovery requires that the system provides 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 that 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 paper. In this paper 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 MORE prototype and report the preliminary experiences with applying MORE to realistic ontologies.

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

2 Solved and Open Problems in Ontology Evolution

Database schema evolution is an important area related to the problem of ontology evolution. In the following, we summarize some of the basic requirements for schema evolution and versioning 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 terms of *change operations* and a *structure for representing changes.*

Integrity. An important aspect of schema evolution is to preserve the integrity of the database during change. *Syntactic conflicts* may occur for example due to multiply defined attribute names in the same class . Further, *semantic conflicts* can appear if changes to the schema break up referential integrity or if the modification of an integrity constraints makes it in compatible with another one.

Compatibility. 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.

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.

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 [12] and for the KAON ontology language [15]. All approaches distinguish between atom and complex changes. 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 [13].

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 [16, 12]. While deciding whether an ontology is consistent or not can easily be done using existing technologies, repairing inconsistencies in ontologies is an open problem although there is some preliminary work on diagnosing the reasons for an inconsistency which is prerequisite for a successful repair [14].

Compatibility. The problem of compatibility with applications that use an ontology has received little attention so far. The problem 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 [10, 9]. This work provides the basis for analyzing compatibility which currently is an open problem.

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. We therefore need methods that work a the semantic level and are aware of logical implications caused by 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.

3 Multi-version Management: 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.

Our goal is to provide ontology managers and users with a tool that helps to detect effects of changes in ontologies and select versions based on their propoerties. Another more ambitious goal for the future is to also provide support for predicting such effects before the ontology has actually been changed [7]. In this section, we introduce the general idea of providing tool support for this purpose and identify relevant use cases for the technology.

3.1 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 [5] 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.

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.

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.

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. In particular this includes determining necessary changes that will make it possible to derive certain wanted statements and the analysis of different development choices using defeasible reasoning techniques.

3.2 The General Approach

The different use cases described above have quite different requirements with respect to inferences that have to be supported. The common feature of all use cases, however, is that they require to reason in the individual ontologies and about the whole set of versions and their relations to each other. While there are existing tools for reasoning with ontologies (i.e. Description Logic Reasoners), being able to reason about different versions is an open issue. In our approach, we mainly address this issue of reasoning about the set of all versions. We do this based on the notion of a version space. A version space is a graph in which different versions of the same ontology form the nodes. Edges represent change operations that led to a new version. We use modal logic to make statements about version spaces, interpreting each version of the ontology as a possible world and change operations the accessibility relation. Queries about a concrete set of versions can now be formulated as a formula in modal logics and model checking techniques can be used to determine whether the version space at hand has the properties specified in the query. In order to determine the facts that hold in a particular world, we use an existing reasoner to derive statements implied by a certain version of the ontology.

The choice of the concrete approach and in particular, the concrete logic to be used to reason about the version space strongly depends on the requirements of the use case. When we look at the three use cases mentioned above, we can see that they have quite different requirements with respect to the expressive power of the query language. The semantic change log only need a very simple logic enabling us to compare different worlds and the statements that hold in each of them. As we will see below, this can be done using a simple temporal logic. Version selection requires explicit references to possible worlds that represent certain versions. This kind of expressiveness is provided by hybrid modal logics [2]. In contrast to the other use cases, evaluation planning requires explicit representations of change operations in the logical language. This requirement is met by dynamic logics [8] that would be appropriate for this use case.

In the remainder of this paper, we discuss a concrete implementation of the general approach outlined above. This concrete implementation addresses the first of the use cases, namely the semantic change log and makes a number of simplifying assumptions in terms of the structure of the version space and the types of statements about an ontology that can be used in queries about the version space. These simplifying assumptions are not general limitations of the approach but address the practical needs of our work in the context of the SEKT Project. In future work, we will extend the MORE system to also meet the requirements of the other use cases.

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 a sequence of versions. Branching time logics 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 [3, 4].

Temporal logics are often future-oriented, because their operators are designed to be ones which involve the future states. Typical operators are: the operator **Future** ϕ which states that 'ϕ holds sometimes in the future with respect to the current state', and the operator **Alwaysf** ϕ which states that 'ϕ always holds in the future with respect to the current state', and the operator ϕ **Until** ψ which states that 'ϕ always holds in the future until ψ holds'. For a discrete time model, the operator **Next** 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 current state.
- the sometimes-in-the past operator states that a fact ϕ holds sometimes in the past with respect to the current state.
- the always-in-the-past operator states that ϕ holds always in the past with respect to the current state.

In this paper, 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 each other via change operations. Each of these ontologies has a unique name. This is different from the work in [10], in which an ontology is considered as one which contains the set of other ontologies which are backwards compatible with. 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*

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

such that $i \neq j \Rightarrow o_i \neq o_j$. 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., $S = (\cdots, o, \cdots, o', \cdots)$.*

It is easy to see that the prior version relation $\prec_S$ is a linear ordering.

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:

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

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 $\perp$ 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 addition operators useful for reasoning about multiple versions. We define the **SomePriorVersion** operator in terms of the **AllPriorVersions** operator as

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

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

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

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:

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

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 [4]. In the implementation of MORE, we are going to use the standard model checking algorithm for evaluating a query in the temporal logic **LTLm**. Therefore, we do not need a complete axiomatization for the logic **LTLm** in this paper.

5 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 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.

5.1 Reasoning Queries

Using the **LTLm** logic we can formulate reasoning queries over a sequence of ontologies that correspond to the typical questions mentioned in Section 3.

Are all facts still derivable? This question can be answered for individual facts using reasoning queries. In particular, we can use the query $\phi \wedge$ **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 was always true in past versions and is 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 $\neg\phi \wedge$ **PreviousVersion**ϕ. Using the **AllPriorVersions** operator, we can also ask whether a fact that was always true in previous versions is not true anymore.

What 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 can use the following query for checking this $\phi \wedge \neg$**PreviousVersion** ϕ. We can also check whether a new fact never holded in previous versions using the following query $\phi \wedge \neg$**SomePriorVersion** ϕ.

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 **PreviousVersion** operator. The query **PreviousVersion PreviousVersion** ϕ for instance checks whether ϕ was true in the version before the previous one. The query ϕ**Since**ψ 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.

5.2 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,

$$child(c, c') =_{df} c' \sqsubseteq c \wedge \nexists c''(c' \sqsubseteq c'' \wedge c'' \sqsubseteq c \wedge c'' \neq c \wedge c'' \neq c').$$

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

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

$$obsoleteChildren(S, o, c) =_{df} \{c' | S, o \models \neg child(c, c') \wedge \textbf{PreviousVersion}\, child(c, c')\}.$$

$$invariantChildren(S, o, c) =_{df} \{c' | S, o \models child(c, c') \wedge \textbf{PreviousVersion}\, child(c, c')\}.$$

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. We will discuss more details in the section about the tests on MORE.

5.3 Making Version-Numbers Explicit

Temporal logics allow us to talk about temporal aspects without reference to a particular time point. For reasoning with multi-version ontologies, we can also talk about 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 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.

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 $\mathbf{Version}_0\phi$ 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 $\mathbf{Version}_{-i}$ as follows:

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

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

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

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 $\mathbf{Version}_{i,S}$. Namely, we can use the formula $\mathbf{Version}_{i,S}\phi$ 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:

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

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:

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

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

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

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

6 Implementation of MORE

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 (compare Fig.1) 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 acts in a certain version.

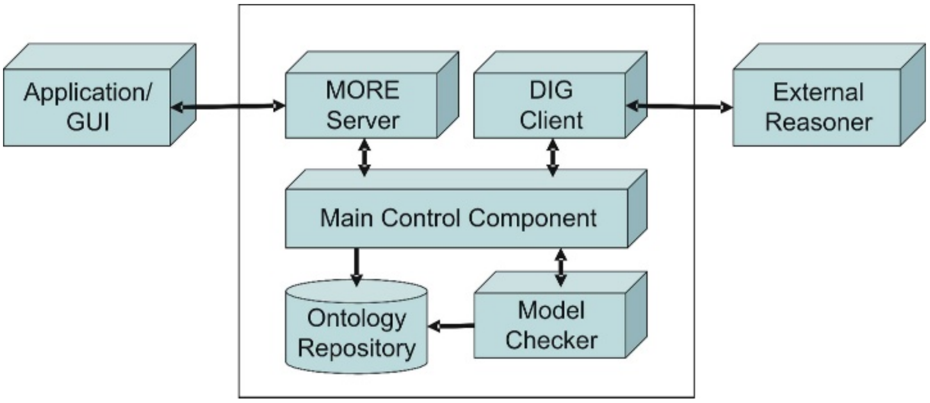

Fig. 1. Architecture of MORE

The MORE prototype is implemented in Prolog and uses the XDIG interface [11], 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 [1][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. A prototype of MORE is available for download at the website: `http://wasp.cs.vu.nl/sekt/more`.

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

6.1 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 a legal Ontology that has been developed in the SEKT project[3] to support the content-based retrieval of legal documents[3]. 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.

The BiosSAIL Ontology. The BioSAIL Ontology which was developed within the BioSTORM project[4]. It has been used in earlier experiments on change management reported in [12]. 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 5.3. In Fig.2 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/`. In a semantic change log, of course, the concrete changes between the versions will be represented. We aggregate the results due to space limitations. What we can immediately see from these numbers alone is that the versions become more stable over time. Especially in the case of the legal ontology, the number of changes from one version to the other becomes significantly lower over time. This can be seen as a sign of maturity.

Besides this change log functionality, arbitrary temporal queries using the operators introduced in this paper 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 [6]. A list of the template queries for temporal reasoning queries are available at the MORE testbed, which can be downloaded from the MORE website. The average time cost for each temporal reasoning query is about 7 seconds for the OPJK Ontology and 3 seconds for the BioSAIL ontology on a PC with 2Ghz CPU 512 MB memory under Windows 2000.

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

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

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	oblk	49	43	36	20	70	20	54	85	377
oblk	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.

Fig. 2. MORE Tests on Concept Relations

7 Discussion and Conclusions

In this paper, 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.

Acknowledgements. We want to thank Pompeu Casanovas and Nuria Casellas Caralt for providing the OPJK ontology, and thank Michel Klein for providing the BiosSAIL Ontology for the tests. The work reported in this paper was partially supported by the EU-funded SEKT project(IST-506826).

References

1. Sean Bechhofer, Ralf Möller, and Peter Crowther. The DIG description logic interface. In *International Workshop on Description Logics (DL2003)*. Rome, September 2003.
2. P. Blackburn and M. Tzakova. Hybrid languages and temporal logic. *Logic Journal of the IGPL*, 7(1):27–54, 1999.
3. V.R. Benjamins P. Casanovas, J. Contreras, J. M. López-Cobo, and L. Lemus. Iurisservice: An intelligent frequently asked questions system to assist newly appointed judges. In V.R. Benjamins et al, editor, *Law and the Semantic Web*, pages 205–522. Springer-Verlag, London, Berlin, 2005.
4. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
5. 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.
6. 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.
7. Peter Haase, Frank van Harmelen, Zhisheng Huang, Heiner Stuckenschmidt, and York Sure. A framework for handling inconsistency in changing ontologies. In *Proceedings of ISWC2005*, 2005.
8. D. Harel. Dynamic logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic Volume II — Extensions of Classical Logic*, pages 497–604. D. Reidel Publishing Company: Dordrecht, The Netherlands, 1984.
9. J. Heflin and J. Hendler. Dynamic ontologies on the web. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000)*, pages 443–449. AAAI/MIT Press, Menlo Park, CA., 2000.
10. J. Heflin and Z. Pan. A model theoretic semantics for ontology versioning. In *Proceedings of ISWC2004*, pages 62–76, Hiroshima, Japan, 2004. Springer.
11. Zhisheng Huang and Cees Visser. Extended DIG description logic interface support for PROLOG. Deliverable D3.4.1.2, SEKT, 2004.
12. M. Klein. *Change Management for Distributed Ontologies*. Phd thesis, Vrije Universiteit Amsterdam, 2004.
13. 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.

14. S. Schlobach and R. Cornet. Non-standard reasoning services for the debugging of description logic terminologies. In *Proceedings of IJCAI2003*, Acapulco, Mexico, 2003. Morgan Kaufmann.
15. L. Stojanovic. *Methods and Tools for Ontology Evolution.* Phd thesis, University of Karlsruhe, 2003.
16. H. Stuckenschmidt and M. Klein. Integrity and change in modular ontologies. In *Proceedings of IJCAI2003*, Acapulco, Mexico, 2003. Morgan Kaufmann.