Understanding ontology evolution: A change detection approach

Peter Plessers *, Olga De Troyer, Sven Casteleyn

Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium

Abstract

In this article, we propose a change detection approach in the context of an ontology evolution framework for OWL DL ontologies. The framework allows ontology engineers to request and apply changes to the ontology they manage. Furthermore, the framework assures that the ontology and its depending artifacts remain consistent after changes have been applied. Innovative is that the framework includes a change detection mechanism that allows generating automatically a detailed overview of changes that have occurred based on a set of change definitions. In addition, different users (such as maintainers of depending artifacts) may have their own set of change definitions, which results into different overviews of the changes, each providing a different view on how the ontology has been changed. Using these change definitions, also different levels of abstraction are supported. Both features will enhance the understanding of the evolution of an ontology for different users.

Keywords: Ontology evolution; Change management; Semantic Web

1. Introduction

The Semantic Web is defined as an extension of the current Web in which information is given well-defined meaning, better enabling computers and people to work in cooperation [2]. To realize this vision, the Semantic Web relies on the use of ontologies. Ontologies are defined as formal, explicit specifications of a shared conceptualization of a domain of interest [4]. Ontologies on the Semantic Web are intended to be used and extended by other ontologies. Furthermore, several other artifacts (e.g., Websites, applications) may rely on ontologies as well. As a consequence, the Semantic Web extends the Web with what can be considered to be a true web of ontologies, where ontologies itself are interlinked and linked to depending artifacts. Furthermore, the distributed and decentralized characteristics of the Web also hold for the Semantic Web. Consequently, ontologies are managed independently from each other and ontologies can be used and extended without the explicit permission of the owner. Moreover, the owner of an ontology is often unaware of who uses or extends his ontology.

The context of this paper is ontology evolution. Evolution is an intrinsic part of the Semantic Web. Alterations in a particular domain, changes to user requirements or corrections of design flaws, they all may induce changes to the corresponding ontologies. Moreover, changes to one ontology may have implications on many depending artifacts [7]. The manual handling of the evolution process of ontologies in a distributed, decentralized environment such as the Semantic Web is not feasible as it is a too laborious, time intensive and complex process. It is therefore vital to have an approach that will guide and support the ontology engineer as well as the maintainers of depending artifacts in this complex process of ontology evolution.

At this moment, there exists no generally accepted definition of ontology evolution in the research community. The term 'ontology evolution' is often used with (slightly) different meanings. Therefore, we first define what we mean by ontology evolution. We define ontology evolution as the process of adaptation of an ontology to arisen changes in the corresponding domain while maintaining both the consistency of the ontology itself as well as the consistency of depending artifacts. Examples of depending artifacts include other ontologies, Websites, Web applications, etc. which depend on the ontology. This definition reflects a similar viewpoint as taken by [3,15].

In general, several problems are associated with ontology evolution. These problems include aspects such as consistency maintenance, backward compatibility, ontology manipulation,

* Corresponding author.
E-mail addresses: Peter.Plessers@vub.ac.be (P. Plessers), Olga.DeTroyer@vub.ac.be (O. De Troyer), Sven.Casteleyn@vub.ac.be (S. Casteleyn).

understanding of ontology evolution, change propagation, etc. In this article, we focus on the problem of understanding of ontology evolution, i.e., understanding which changes have been made to an ontology. The problem of understanding of evolution is a fundamental problem of ontology evolution as, based on this understanding, maintainers of depending artifacts need to take a decision about possible changes to their depending artifact whenever an ontology they depend on changes. Moreover, as ontologies are often developed by several ontology engineers, it is also important for them to understand what changes have been made by each other. In this paper, we argue that only listing the changes as applied is in general insufficient for the purpose of understanding evolution.

In this article, we introduce the notion of a *version log* and present a *Change Definition Language* (CDL) for the OWL DL ontology language as a solution to deal with the problem of understanding ontology evolution. The version log keeps track of all the different versions of all concepts ever defined in an ontology, while the CDL allows users to define the *meaning* of changes in a formal way. The version log and the CDL make it possible to pose queries on the evolution of an ontology. The version log is explained and a formal model is presented. Furthermore, the Change Definition Language is explained and its syntax and semantics are defined.

The paper is structured as follows. Section 2 presents the motivation behind our approach, while Section 3 provides a general overview the overall ontology evolution framework and briefly describes its different phases. Section 4 discusses the version log for keeping track of the different versions of concepts. Section 5 presents the Change Definition Language that allows defining changes in a formal and declarative way. Section 6 discusses our change detection approach, while Section 7 discusses the implementation of this approach. Section 8 presents related work. Finally, Section 9 provides conclusions.

2. Motivation

The general idea, to assist users in understanding the evolution of an ontology, is to provide them an overview of the changes applied to a particular ontology. In this way, they can check what has been changed to the ontology and take appropriate actions with regard to their own depending artifacts. The simplest way to achieve this is by providing a list of all the change operations that were explicitly used by an ontology engineer to change the ontology. However, this approach has a number of serious drawbacks:

- *Level of granularity*: The list of change operations applied only provides one level of granularity for the different changes. Consider for example the case where an ontology engineer changes an ontology by applying the following sequence of change operations: changing the subclass relation for a class A from a class B to a class C by deleting the subclass relation between A and B, and then adding a subclass relation between A and C. In such a case, only providing a list of change operations is rather low level and fails to give the change at a higher level of abstraction, i.e., as a more complex change representing the change of a subclass property.
- *Different viewpoints*: Even if the ontology engineer is using complex changes (such as the one given above) to change the ontology, the overview of change operations will be restricted to the set of (complex) change operations used by the ontology engineer. The change operations used by the ontology engineer represent the personal view (in terms of complex changes) of the ontology engineer on the changes. As different users may have different views (e.g., in terms of granularity) and different needs, this might be too restrictive to satisfy other users' desires. It would therefore be desirable that users can define explicitly those (complex) changes they are interested in, and this at the level of abstraction appropriate for them. For e.g., a user must be able to define that he is interested in changes that weak the domain of some property (even in the case that this was indirectly achieved by the ontology engineer by using a combination of change operations).

  In addition, while several users may agree on a particular ontology, they do not necessarily need (or agree on) the same set of (complex) changes. Consider the example illustrated in Fig. 1.[1] In this example, the range of the object property p changes from class A to class B (from step 1 to step 2). Going from step 2 to step 3, a *subClassOf* property is added between class A and B. Probably some users would agree that this sequence of changes corresponds to "a weakening of the property p" (weakening in the sense that its range has been replaced by a super class of the original range). However, others might disagree with this and might require that in order to speak of "a weakening of a property p", a subclass property between the new and the old range is required already before the range of the property is changed. In that case, A must have been a subclass of B in step 1. Therefore, it is important that different users can use their own definitions for complex changes.
- *Implications of changes*: The list of change operations applied only focuses on one aspect of the evolution, i.e., *what* has changed. They do not offer much information on the implications of a particular change (side effects) for other concepts in the ontology. This kind of information could greatly improve the understanding of evolution. Consider the following example illustrated in Fig. 2. The example expresses that the class *Student* is a subclass of *Person*, and for *Person* there exists a cardinality restriction on the property *name* stating that a *Person* has exactly one *name*. Because *Student* is a subclass of Person, each *Student* has also exactly one *name*. When we would delete the subclass property between *Student* and *Person*, it might be useful for users to know not only that the subclass property has been deleted, but also that for *Students* there is no longer a cardinality restriction on the property *name*.

[1] We use the VisioOWL representation by John Flynn to visualize OWL ontologies. See http://semanticsimulations.com/VisioOWL/VisioOWL.htm for more information.

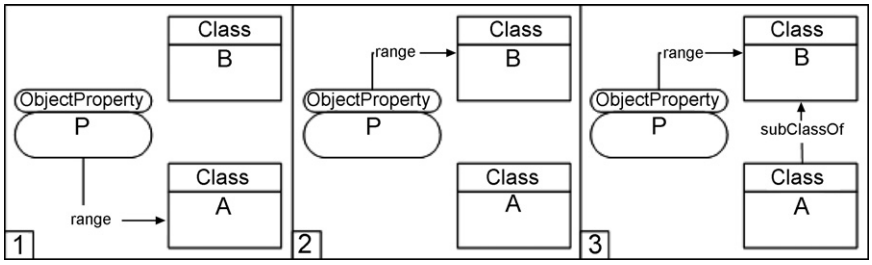

Fig. 1. Example sequence of changes.

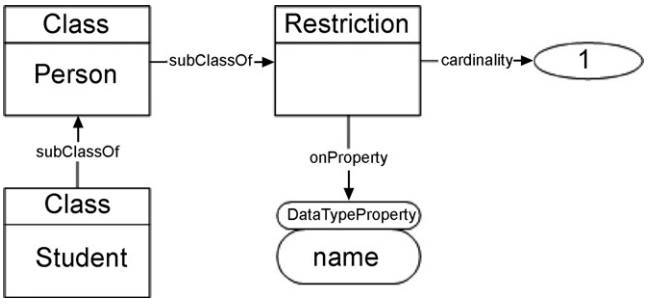

Fig. 2. Example ontology modification.

It is clear now that only providing the list of change operations used by the ontology engineer to change an ontology as a solution to the ontology evolution problem is not sufficient. In our approach, we make a clear distinction between the ontology evolution itself by keeping track of the different versions ontology concepts pass through during their lifetime (expressed by the version log) and the (different) *interpretation*(*s*) of the ontology evolution, which will be expressed by (an) evolution log(s). Doing so, users can easily define their own needs (in terms of changes they want to see) and subsequently an evolution log based on these personal preferences and needs will be generated. This view will correspond to their own set of definitions of changes, which will make the evolution log easier to understand for them.

3. Ontology evolution framework

In this section, we give a brief overview of our overall ontology evolution framework, before we go into deeper detail on the version log and Change Definition Language in the next sections. For further details about (other aspects of) the framework, we refer the interested reader to [11] for an initial version, and [13], for the latest version of the framework.

The structure of this section is as follows. Section 3.1 describes the main characteristics of our ontology evolution framework. Section 3.2 gives a brief overview of the framework itself and its different phases.

3.1. Characteristics

Our ontology evolution framework targets two types of users (or at least two user roles): on the one hand, the ontology engineer that is modifying an ontology for which he is (partly) responsible, on the other hand, the ontology engineer that needs to be informed about the changes made to an ontology he depends on. For this reason, the framework consists of two parts, each focusing on one role: respectively, the *evolution-on-request part* and the *evolution-in-response part*. Each part has its own particular phases.

The framework represents the evolution of an ontology by means of a *version log*. A version log stores for each concept ever defined in the ontology (including Classes, Properties and Individuals), the different versions it passes through during its life cycle: starting from its creation, over its modifications, until the eventual retirement. Whenever an ontology concept is modified in the ontology, the version log is updated by creating a new version in the version log for the affected concept that represents its current state. So, a version log describes all the states (at the different moments in time) of the different concepts defined in an ontology. Note that, with the right tool support, a version log can be created automatically (as is shown in Section 6).

In our approach, possible changes are formally defined by means of a temporal logic based language, called the *Change Definition Language* (CDL). Using this language, changes are specified in terms of conditions that must hold before and after the appliance of the change (respectively, pre- and post-conditions). Together with the version log, the CDL provides the foundation of the change detection approach.

The purpose of the change detection phase is to detect if changes as defined by means of the CDL have occurred. For this purpose, change definitions expressed in the CDL are evaluated as temporal queries on a version log. The outcome of the change detection phase is a collection of occurrences of change definitions. We call this collection an *evolution log* because it describes the evolution of an ontology in terms of the change definitions used. As is shown in the following section, the change detection approach is used in both the evolution-on-request and the evolution-in-response part.

3.2. Framework overview

Fig. 3 shows an overview of the different phases for both parts of the ontology evolution process. The evolution-on-request part consists of the following phases: (1) change request, (2) consistency maintenance, (3) change detection, (4) change recovery and (5) change implementation. The evolution-in-response consists of the following phases: (1) change detection, (2) cost of evolution, and (3) version consistency.

We give an overview of the different phases for both parts of the framework in the subsequent subsections.

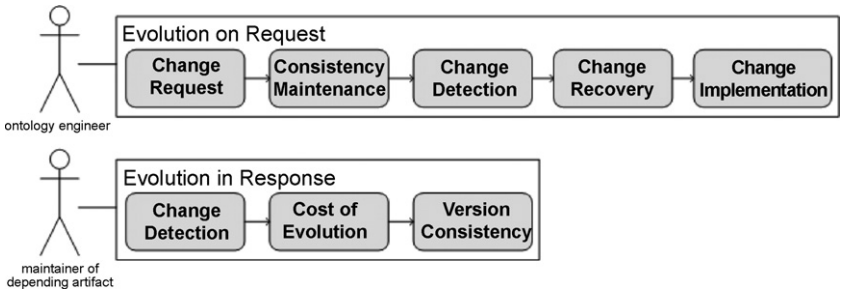

Fig. 3. Overview of the ontology evolution framework.

3.2.1. Evolution-on-request

In this section, we discuss the various phases of the ontology evolution framework to fulfill the evolution-on-request task: (1) change request, (2) consistency maintenance, (3) change detection, (4) change recovery, and (5) change implementation.

Change request: This phase lets ontology engineers express their request for change in terms of defined changes. The framework supports both *primitive* and *complex changes*. We use the term 'primitive change' to refer to changes that only affect a single OWL primitive. The set of primitive changes is exhaustive as it is derived from the underlying ontology language. We use the term 'complex change' to refer to changes that affect more then one ontology construct. Note that the set of complex changes is infinite as new complex changes can always be defined [9].

Consistency maintenance: As modifications to an ontology may turn the ontology into in an inconsistent state, this phase deals with the (automatic) verification whether the ontology remains consistent after changes are applied. It will highlight the axioms that are responsible for a particular inconsistency and offer a number of possible solutions to resolve the detected inconsistency. As the problem of consistency maintenance is not the focus of this article, we refer the interested reader to [12] which provides a complete overview of our approach to consistency maintenance.

Change detection: The purpose of the change detection phase for the evolution-on-request part is to detect which (complex) change definitions have occurred as a consequence of the modifications made to the ontology. These are the complex changes defined by the ontology engineer but not explicitly used when formulating the change request. Instead of exclusively basing the evolution log on the changes explicitly requested by an ontology engineer, we also include in the evolution log the changes automatically detected during this phase. This means that in our approach, the evolution log is constructed from requested changes as well as detected changes. We discuss the approach to change detection in more detail in Section 6.

Change recovery: It is possible that, whenever a sequence of change operations is used instead of an appropriate more complex change, concepts are also changed as a consequence of the consistency maintenance phase to ensure the consistency of the ontology after each individual change operation. However, when considering the complete sequence of change operations, some of the changes to (some of) the concepts made after the individual change operations to maintain consistency, may turn out to be unnecessary (as they were not necessary to maintain the consistency of the ontology when looking at the sequence of changes as a whole). To overcome this problem, this phase will take care of recovering from these unnecessary changes, by *undoing* these changes. A change is undone by applying the inverse change to the ontology. The exact details of the change recovery process can be found in [13].

Change implementation: Until this phase, the changes have only been applied to a local copy of the ontology. The purpose of this phase is to implement these changes to the public version of the ontology. This is straightforward as it only consists of replacing the original, public ontology by the evolved, local copy.

3.2.2. Evolution-in-response

In this section, we discuss the various phases of the ontology evolution framework to fulfill the evolution-in-response task: (1) change detection, (2) cost of evolution, and (3) version consistency.

Change detection: The objective of this phase is the same as of the change detection phase in the evolution-on-request part. The reason why it is also included it in this part of the framework, is because users may have their own set of change definitions and therefore they will obtain another evolution log for the ontology they depend on. Based on the information in their evolution log, maintainers of depending artifacts can decide whether they still agree with the modified ontology and whether they find the changes to the ontology worthwhile to update their depending artifact to the latest version of the ontology. Note that adopting the change detection phase in our ontology evolution framework allows ontology engineers and maintainers of depending artifacts to use a different set of change definitions and to have different views on the same evolution.

Cost of evolution: Besides understanding the changes that have occurred, another key element in the decision whether or not to update a depending ontology when the ontology it depends on is changed is the *cost* of updating. With the cost of updating we refer to the number of updates needed in the depending ontology in order to keep it consistent with the new version of the ontology. In other words, which subset of the depending ontology becomes inconsistent, and which axioms are responsible for an inconsistency and possibly need to be changed?

Version consistency: The objective of this phase is to maintain consistency of a depending ontology with the ontology it depends on, regardless of the fact whether the ontology engineer decides to update or not. If the depending ontology is not updated, it keeps depending on the old version of the ontology, and therefore remains consistent with the old version. If the depending ontology is updated, possible inconsistencies need to be resolved to assure that the depending ontology is consistent with the new version of the ontology it depends on.

4. Version log

In this section, we describe in more detail the version log that is used in our approach to represent the evolution of an ontology. Section 4.1 discusses the general principles used for the version log. Section 4.2 presents a formal model of a version log.

4.1. General principles

Preserving the history of an ontology can be achieved in different ways. First of all, an approach based on *timestamps* or *snapshots* can be taken. Although both approaches are equally expressive, their manner of representation differs. The former consists of labeling triples in an OWL ontology with a timestamp. The latter is based on taking snapshots to capture the different states of an ontology over time. Snapshots can be taken either of the ontology as a whole, where each snapshot contains all facts valid at a given moment in time. In that case, the history of an ontology is described as a sequence of snapshots. Snapshots can also be taken of single concept definitions, in which case a sequence of snapshots describes the evolution of a single ontology concept. The first form is considered highly inefficient as each snapshot stores the complete ontology. Moreover, capturing the evolution of the individual concepts is required if one wants to pose sensible queries on the evolution of concepts. This is not always possible when snapshots of the whole ontology are taken, as they do not capture relations between successive versions of ontology concepts (e.g., in case of renaming concepts). In our approach, we have adopted a snapshot approach that keeps track of the evolution of each individual ontology concept.

In temporal databases, in general two dimensions of time are considered: *transaction time* and *valid time* [14]. Transaction time represents the time when data is actually stored in the database, while valid time represents the time when data is valid in the modeled world. The same two dimensions can also be applied to ontologies. With regard to the version log, transaction time would reflect the time when a new version of a concept is created, while valid time would represent the moment in time when the concept is valid in the described domain. We only keep track of the transaction time in the version log as we are only interested in the moment in time when changes are applied to the ontology.

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

4.2. Formal model

In this section, we formally define the version log. Instead of using the native OWL syntax (i.e., the RDF/XML syntax), we rely on the syntax of the SHOIN(D) description logic[2] variant to represent concept definitions. This means that the definition of an ontology concept consists of a set of DL axioms.

We first introduce the set of all concepts (and more specifically all classes, properties and individuals) that have been defined during the complete lifetime of the associated ontology. Assume the following sets: the set of all classes, **CL**, the set of all properties **PR**, and the set of all individuals **IN**. The set of all concepts $\boldsymbol{C}$ is then defined as $\boldsymbol{C} = \mathbf{CL} \cup \mathbf{PR} \cup \mathbf{IN}$. Furthermore, we also introduce the set of all concept names $\boldsymbol{N}$ that have been used during the complete lifetime of the associated ontology.[3] Note that both the set $\boldsymbol{C}$ and $\boldsymbol{N}$ are dynamically extended whenever, respectively, a new concept or concept name is introduced.

We define the timeline of a version log as follows:

Definition 4.1 *(Timeline).* The discrete, linearly ordered timeline $\boldsymbol{T}$ of a version log is defined as the finite set $\boldsymbol{T} \subseteq \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers. The time point $now \in \boldsymbol{T}$, also called the current time, is defined so that $\forall t \in \boldsymbol{T}\ (t \leq now)$.

Note that the timeline $\boldsymbol{T}$ is dynamically extended with additional time points whenever the version log is updated. The version log keeps track of the history of each class, property and individual that is defined in the associated ontology by storing their successive versions. A version of a concept contains the definition of that concept at a given moment in time. In DL, the definition of a concept c is formed by a set of axioms $\boldsymbol{A} = \{\varphi_1, \ldots, \varphi_n\}$. We say that the set $\boldsymbol{A}$ forms the concept definition of a concept c, notation $\mathit{ConceptDefinition}(\boldsymbol{A}, c)$.

Each version has a start and end time point. When a particular version is still the current version, the end time is set equal to the *now* time point. Each concept version contains the concept name, i.e., the name the concept has during that version (might be the empty string), and the set of axioms that form its definition. A concept version is formally defined as follows:

Definition 4.2 *(Concept version).* A concept version v for a given concept $c \in \boldsymbol{C}$ is a tuple $(\sigma, \boldsymbol{A}, t_s, t_e)$ where $\sigma \in \boldsymbol{N}$ is the concept name, $\boldsymbol{A}$ the set of axioms forming the definition of c, i.e., ConceptDefinition($\boldsymbol{A}$, c), and $t_s, t_e \in \boldsymbol{T}$ are, respectively, the start- and end-time of the version. The start- and end-time denote a closed interval $[t_s, t_e]$.

The start times of concept versions introduce a total ordering between concept versions of the same concept. We therefore introduce the precedence relation $\prec_v$ for concept versions. The precedence relation expresses that one concept version ended before a second concept started.

[2] SHOIN(D) is the DL variant on which OWL is based.

[3] Note that the same concept name can be reused over time to identify different concepts.

Definition 4.3 *(Ordering of concept versions).* For a concept $c \in \boldsymbol{C}$, $v_1 \prec_v v_2$ iff $t_e < t'_s$ where $v_1 = (\sigma, \boldsymbol{A}, t_s, t_e)$ and $v_2 = (\sigma', \boldsymbol{A}', t'_s, t'_e)$, and *ConceptDefinition*($\boldsymbol{A}$, c) and *ConceptDefinition*($\boldsymbol{A}'$, c). The order relation $\prec_v$ is an anti-symmetric, transitive and total relation.

We now define the complete history of an ontology concept as a sequence of concept versions for this concept. We call this a concept evolution:

Definition 4.4 *(Concept evolution).* A concept evolution $\boldsymbol{E}_c$ for a concept $c \in \boldsymbol{C}$ is a finite set of concept versions so that $\forall v \in \boldsymbol{E}_c\ (v = (\sigma, \boldsymbol{A}, t_s, t_e))$ where *ConceptDefinition*($\boldsymbol{A}$, c).

Before concluding this section with the definition of a version log, we define a number of auxiliary definitions that are based on Definitions 4.3 and 4.4. The first auxiliary definition introduces the notion of a concept version at a given point in time.

Definition 4.5. Given a concept version v, a concept $c \in \boldsymbol{C}$, and a time point $t \in \boldsymbol{T}$, we note version (v, c, t) iff $v = (\sigma, \boldsymbol{A}, t_s, t_e)$ and $t_s \leq t \leq t_e$ where *ConceptDefinition*($\boldsymbol{A}$, c).

Based on the previous definition, we are able to define when we a concept version is the previous version for a given concept w.r.t. a given time point. Furthermore, we also define when a concept version is the first version for a concept. Both definitions will be used when the semantics of our Change Definition Language is defined (see Section 5.1.2).

Definition 4.6 *(Previous concept version).* A concept version v is defined as the previous concept version for a given concept $c \in \boldsymbol{C}$ w.r.t. a given time point $t \in \boldsymbol{T}$, notation *PreviousConceptVersion*(v, c, t), iff $\exists v_i \in \boldsymbol{E}_c\ (version(v_i, c, t) \wedge (v \prec_v v_i) \wedge \neg \exists v_j \in \boldsymbol{E}_c\ (v \prec_v v_i \wedge v \prec_v v_i))$.

Definition 4.7 *(First concept version).* A concept version v is defined as the first concept version for a given concept $c \in \boldsymbol{C}$, notation *FirstConceptVersion*(v, c), iff $\neg \exists v_i \in \boldsymbol{E}_{\boldsymbol{\sigma}}\ (v_i \prec_v v)$.

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

Definition 4.8 *(Version log).* A version log $\boldsymbol{V}$ is defined as the set of concept evolutions so that $\forall c \in \boldsymbol{C}\ (\boldsymbol{E}_c \in \boldsymbol{V}) \wedge \forall \boldsymbol{E}_c \in \boldsymbol{V}\ (c \in \boldsymbol{C})$.

5. Change Definition Language

In this section, we discuss the Change Definition Language (CDL) that allows specifying change definitions in a formal and declarative way. The language has its foundation in temporal logic, which makes it possible to express changes in a declarative way in terms of differences between past and current versions. Section 5.1 presents the temporal logic underlying the CDL to clarify the semantics of the language. Section 5.2 discusses the syntax and use of the CDL.

5.1. Temporal logic

5.1.1. General approach

The term temporal logic has been broadly used to cover approaches to the representation of temporal information within a logical framework. Most approaches take a modal-logic based approach to the problem of temporal representation. Modal-logic approaches introduce modal-operators to gain the ability to model properties such as belief, time-dependence, obligation, and so on. When applied to the temporal domain, modal-logic approaches introduce so-called *tense operators* as modal-operators.

Although most temporal languages consider both past and future tense operators, for the purpose our change definitions we restrict ourselves to past tense operators. Generally, two basic past tense operators P ('some time in the past') and H ('always in the past') exist. A number of common extensions to these basic tense operators were introduced including the binary S ('since') operator. Another common extension for discrete timelines is the previous time operator called R to refer to the immediately preceding moment in time. In addition to the tense operators previously mentioned, we also introduce the tense operator A ('after') as a weak version of the S tense operator.

The temporal logic supports two different views on the timeline $\boldsymbol{T}$ of a version log. The first view considers the complete timeline as it takes the history of the whole ontology into account (i.e., all concept versions of a version log). The second view only considers the part of the timeline that belongs to the history of a particular concept (i.e., all concept versions v that belong to a given concept evolution $\boldsymbol{E}_c$). To reflect both views in our temporal logic, we have further extended the set of tense operators by introducing parameterized versions of the tense operators H, P and R. The original, non-parameterized version of the tense operators correspond to the view that takes all concept versions of the ontology into consideration, while the parameterized version, where the parameter refers to a concept, only considers the concept versions associated with the evolution of that particular concept.

5.1.2. Syntax and semantics

In this section, we define both the syntax and semantics of the temporal logic that underlies the Change Definition Language.

The syntax is defined as follows:

Definition 5.1 *(Syntax).* Assume $\sigma \in \boldsymbol{N}$ to be a concept name. If ϕ and ψ are axioms of SHOIN(D), then so are $\langle H \rangle \phi$ and $\langle H(\sigma) \rangle \phi$, $\langle P \rangle (\phi)$ and $\langle P(\sigma) \rangle \phi$, $\langle R \rangle \phi$ and $\langle R(\sigma) \rangle) \phi$, $\phi \langle S \rangle \psi$, and $\phi \langle A \rangle \psi$ (tense operators); $\neg \phi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, and $\varphi \rightarrow \psi$ (common logic operators). Nothing else is a valid axiom.

To shorten the definition of the semantics of the temporal logic, we define the disjunction and implication in terms of negation and conjunction:

$$\phi \vee \psi = \neg(\neg \phi \wedge \neg \psi)$$
$$\phi \rightarrow \psi = \psi \wedge \neg \phi$$

Before defining the semantics of the temporal logic, we introduce a number of auxiliary definitions. We first define how we

can derive a snapshot from the version log given a point in time. Intuitively, a snapshot $\boldsymbol{S}(t)$ is the set of all axioms, kept in a version log, that hold at a given moment in time $t \in \boldsymbol{T}$.

Definition 5.2 *(Snapshot).* For a given time point $t \in \boldsymbol{T}$, a snapshot of a version log at moment t, notation $\boldsymbol{S}(t)$, is defined as $\boldsymbol{S}(t) = U_{\forall v}\{\phi \in \boldsymbol{A} \mid v = (\sigma, \boldsymbol{A}, t_s, t_e) \wedge t_s \leq t \leq t_e\}$.

To be able to define the semantics of the parameterized tense operators, we define two views on the timeline $\boldsymbol{T}$. The first view, called $\boldsymbol{T}^{\boldsymbol{p}}_{c,t}$, contains the time points of the directly preceding concept version for a given concept c w.r.t. a given time point t. The second view, called $\boldsymbol{T}^{\boldsymbol{\alpha}}_{c,t}$, contains the time points of all preceding concept versions for a given concept c w.r.t. a given time point t.

Definition 5.3 *(Timeline $\boldsymbol{T}^{\boldsymbol{p}}_{c,t}$).* The set $\boldsymbol{T}^{\boldsymbol{p}}_{c,t} \subseteq \boldsymbol{T}$ is defined as the timeline spanning the previous concept version for a given concept $c \in \boldsymbol{C}$ w.r.t. a given time point $t \in \boldsymbol{T}$, so that $\boldsymbol{T}^{\boldsymbol{p}}_{c,t} = \{t' \in \boldsymbol{T} \mid PreviousConceptVersion(v, c, t) \wedge t_s \leq t' \leq t_e\}$ where $v = (\sigma, \boldsymbol{A}, t_s, t_e)$.

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

$$\boldsymbol{T}^{\boldsymbol{\alpha}}_{c,t} = \left\{ t' \in \boldsymbol{T} \;\middle|\; \begin{array}{r} \exists v = (\sigma, \boldsymbol{A}, t_s, t_e), \exists v' = (\sigma', \boldsymbol{A}', t'_s, t'_e). \\ (FirstConceptVersion(v, c) \wedge PreviousConceptVersion(v', c, t) \wedge t_s \leq t' \leq t'_e) \end{array} \right\}.$$

Finally, we define the semantics of the temporal logic by the following definition:

Definition 5.4 *(Semantics).* Given an axiom ϕ, an interpretation I and a time point $t \in \boldsymbol{T}$, we extend the entailment relation $I, t \models \phi$ (ϕ holds in I at moment t) of SHOIN(D) as follows:

$$\begin{aligned}
I, t \models \phi &\leftrightarrow \boldsymbol{S}(t) \models \phi, \text{where } \phi \in \text{SHOIN}(\mathbf{D}) \\
I, t \models \phi \wedge \psi &\leftrightarrow I, t \models \phi \text{ and } I, t \models \psi \\
I, t \models \neg\phi &\leftrightarrow I, t \not\models \phi \\
I, t \models \langle R \rangle \phi &\leftrightarrow \exists t' \in \boldsymbol{T}.(t' = t - 1 \wedge I, t' \models \phi) \\
I, t \models \langle H \rangle \phi &\leftrightarrow \forall t' \in \boldsymbol{T}.(t' < t \wedge I, t' \models \phi) \\
I, t \models \langle P \rangle \phi &\leftrightarrow \exists t' \in \boldsymbol{T}.(t' < t \wedge I, t' \models \phi) \\
I, t \models \langle R(\sigma) \rangle \phi &\leftrightarrow \exists t' \in \boldsymbol{T}^{\boldsymbol{p}}_{c,t}.(I, t' \models \phi) \\
I, t \models \langle H(\sigma) \rangle \phi &\leftrightarrow \forall t' \in \boldsymbol{T}^{\boldsymbol{a}}_{c,t}.(I, t' \models \phi) \\
I, t \models \langle P(\sigma) \rangle \phi &\leftrightarrow \exists t' \in \boldsymbol{T}^{\boldsymbol{a}}_{c,t}.(I, t' \models \phi) \\
I, t \models \phi \langle S \rangle \psi &\leftrightarrow \exists t' \in \boldsymbol{T}.\big(t' < t \wedge I, t' \models \psi \wedge \forall t'' \in \boldsymbol{T}.(t' < t'' < t \wedge I, t'' \models \phi)\big) \\
I, t \models \phi \langle A \rangle \psi &\leftrightarrow \exists t' \in \boldsymbol{T}.\big(t' < t \wedge I, t' \models \psi \wedge \exists t'' \in \boldsymbol{T}.(t' < t'' < t \wedge I, t'' \models \phi)\big)
\end{aligned}$$

5.2. CDL syntax and use

In this section, we give a complete overview of the syntax of CDL. We specify the syntax by means of its EBNF specification. Furthermore, we provide a number of example change definitions to illustrate the usage of CDL.

A change definition expressed in the CDL is always composed of a *header* and a *body*. The header of a change definition contains the name (or *identifier*) of the change and a *list of parameters*. The body consists of an expression.

```
changeDefinition  ::=  identifier '(' (parameterList)? ')' ':=' body ';'
parameterList     ::=  parameter (',' parameter)*
parameter         ::=  '?' identifier
body              ::=  expression
```

Expressions can be formed using the well-known logical operators $\neg$, $\wedge$ and $\vee$, respectively, NOT, AND and OR in the syntax of the CDL. Parentheses can be used to change the standard preceding rules. The building blocks of expressions are statements, temporal expressions and native functions.

```
expression        ::=  factor ('OR' factor)*
factor            ::=  secondary ('AND' secondary)*
secondary         ::=  (primary | 'NOT' primary)
primary           ::=  (statement | parenExp | tempExp | natFunc)
parenExp          ::=  '(' expression ')'
```

The identifier of a statement is either a class or a property that we defined in our OWL DL meta-schema.[4] An optional asterisk

[4] The meta-schema can be downloaded from http://wise.vub.ac.be/members/peterp/versionontology.owl.

symbol (‘*’) following the identifier of the statement indicates whether the transitive characteristic of a property (when present) should be taken into account (we discuss this feature in more detail in Section 7). Omitting the asterisk symbol restricts the statement to direct properties only.

```
statement        ::=  identifier (‘*’)? ‘(‘ subject (‘,’ object)? ‘)’
subject          ::=  (parameter | identifier )
object           ::=  (parameter | identifier | value )
value            ::=  ‘"’ STRING ‘"’
identifier       ::=  STRING
```

The tense operators defined in the previous section are represented by means of the more meaningful terms ALWAYS, SOMETIME and PREVIOUS for, respectively, the tense operators *H*, *P* and *R*. SINCE and AFTER are, respectively, used for the tense operators *S* and *A*.

```
tempExp          ::=  (unaryTempExp | binaryTempExp)
unaryTempExp     ::=  ‘<’ (‘ALWAYS’ | ‘SOMETIME’ | ‘PREVIOUS’) ‘>’ parenExp
binaryTempExp    ::=  ‘<’ (‘SINCE’ | ‘AFTER’) ‘>’
                      ‘(‘ expression ‘,’ expression ‘)’
```

The final part of our language consists of the native functions *equal*, *It* and *gt*. Intuitively, the function *equal* expresses that two concepts or two values are equal. The functions *It* and *gt* (which only operate on values) express, respectively, that a first value is lesser than or greater than a second value.

```
tempExp          ::=  (unaryTempExp | binaryTempExp)
natFunc          ::=  (‘equal’ | ‘lt’ | ‘gt’) ‘(‘ natArg ‘,’ natArg ‘)’
natArg           ::=  (parameter | identifier | value)
```

We illustrate the use of the CDL by means of a number of example change definitions. The first example defines the change that corresponds to the addition of a new *subClassOf* property between two classes:

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

The change definition states that a *subClassOf* property is added between a Class ?s and a Class ?o (line 1) whenever in the previous version of the ontology ?s *was not* a subclass of ?o (line 2), and in the current version ?s *is* a subclass of ?o (line 3).

Consider for a second example an ontology engineer who considers a property to be *stable* if the property has *always* been either a datatype property or an object property. As he is interested in knowing when a *stable* property becomes *unstable* (i.e., when a datatype property for the first time is changed to an object property and the other way round), he introduces the *changePropertyFromStableToUnstable* change definition as shown below. Note that his opinion about a *stable* property is his personal opinion which may not be shared by others.

```
(1)  changePropertyFromStableToUnstable(?p) :=
(2)    (<ALWAYS(?p)>(DatatypeProperty(?p)) AND
(3)     ObjectProperty(?p))
(4)    OR
(5)    (<ALWAYS(?p)>(ObjectProperty(?p)) AND
(6)     DatatypeProperty(?p));
```

The change definition states that a property ?p either has always been a datatype property in the past (line 2) but has changed to an object property now (line 3), or has always been an object property in the past (line 5) and has changed to a datatype property now (line 6). Note that we use the parameterized version of the ALWAYS tense operator to indicate that we are only interested in the past versions of the ?p property, and not in the complete evolution of the ontology as a whole. Line 2 for example expresses that ?p, *during the entire lifetime of the property ?p*, has always been a datatype property. If we, however, would omit the parameter for the tense operator, line 2 it would express that ?p, *during the entire lifetime of the ontology*, has always been a datatype property which expresses a completely different meaning. Removing the parameterized tense operators from the change definition would strengthen the meaning of a *stable property* by additionally requiring that such a property should have existed since the beginning of the lifetime of the ontology (as the non-parameterized version of the ALWAYS tense operator takes the complete timeline of the version log into account), which does not correspond to the meaning of a *stable property* given above.

6. Change detection

To detect occurrences of change definitions, a set of change definitions is evaluated as temporal queries on a version log. An evaluation of a change definition as a temporal query intends to find all possible bindings for the parameters used in the body expression of the change definition that satisfy this expression (according to the semantics defined in Section 5.1.2). For e.g., an evaluation of the second example in Section 5.2 will try to find all possible bindings for the ?p parameter, i.e., all properties that have always been a datatype property in the past but then became an object property, or the other way around. The parameter bindings that are returned from the evaluation are the bindings for the parameters listed in the header of the change definition. If no valid bindings for the parameters used in the body expression can be found, no bindings are returned.

For each returned set of parameter bindings, a new entry is added to the corresponding evolution log. An entry in the evolution log stores the set of parameter bindings, a reference to the identity of the change definition used, and a time point indicating the moment of detection. This time point is an element of the version log timeline $\boldsymbol{T}$. Note that the occurrence of a change definition is only added to the evolution log if the same occurrence does not already exist in the evolution log (it is possible that the change was specified in a change request). So, an evolution log lists occurrences of change definitions where such an occurrence refers to the concepts and/or values from the version log for which the referred change definition was satisfied at the given moment in time.

7. Implementation

To validate our approach, we have developed two prototype extensions for the Protégé ontology editor.[5] A first plug-in automatically creates a version log by tracking all the changes that are applied to an ontology with the Protégé ontology editor (called the Version Log Generator). A second plug-in takes as input a set of change definitions and a version log, and outputs an evolution log by evaluating the given change definitions on the given version log (called the change detection plug-in).

Whenever an ontology is loaded in Protégé, the Version Log Generator either creates a new version log or restores an existing one. Whenever changes are applied to the loaded ontology, the Version Log Generator catches the events thrown by Protégé and updates the version log by setting the end time of the latest concept versions of the concepts involved in the change and by creating the appropriate new concept versions representing the new state of the changed concepts.

Changes to an ontology may cause previously inferred knowledge to be no longer valid and, the other way round, previously not inferred knowledge may become inferable after the change. As we also want to be able to detect changes to implicit knowledge, we explicitly store this inferred knowledge in the version log. To retrieve implicit knowledge from an ontology, we rely on an external description logic reasoner. We use the Jena framework to retrieve a model of inferable statements from the ontology by means of the plugged-in reasoner. Inferred statements that are not present in current concept versions are added to the version log, while statements in current concept versions that are not present in the model of inferred statements are removed.

The change detection plug-in implements an evaluator for the CDL. The implementation of the evaluator of the CDL relies on the RDQL implementation of the Jena framework to query the version log. The statements used in change definitions are converted to RDQL queries. The tense operators, logical operators and native functions (as specified in Section 5.1.2) are implemented by the plug-in on top of the RDQL query results.

[5] See http://protege.stanford.edu/.

8. Related work

In this section, we give an overview of current practices in the domain of ontology evolution, focusing in particular on the problem of change detection. Stojanovic [16] has defined ontology evolution as the timely adaptation of an ontology to the arisen changes and the consistent propagation of these changes to depending artifacts. In [15], the authors identified a possible evolution process. To represent changes, they introduced in [10] three levels of abstractions for ontology changes for the KAON language. They distinguish: elementary changes (modifications to one single ontology entity), composite changes (modifications to the direct neighborhood of an ontology entity) and complex changes (modifications to an arbitrary set of ontology entities).

To support the understanding of evolution, their framework provides an overview of all changes applied. Although our framework has a number of phases in common with their framework, the main difference is that their framework does not include a change detection approach. As a consequence, the changes that get listed in an evolution log are only the result of changes explicitly requested by an ontology engineer. As discussed in Section 2, this most likely results in a less rich overview of changes (e.g., implicitly applied complex changes are not listed) and prevents different users from having different views on the same evolution.

This framework has been further extended by Haase including support for the OWL language [5]. However, no support for change detection has been included.

An ontology evolution framework for OWL has also been developed by Klein et al. [8]. Similar to Stojanovic, he gives a similar taxonomy for the OWL language for which he defines both basic and complex change operations. Basic change operations are changes to one single ontology entity whereas complex change operations are a mechanism for grouping basic change operations together to form a logical unit. Furthermore, in [7], he indicated the usefulness of a composite change detection approach. The authors introduced a detection mechanism based on rules and heuristics to detect composite changes between two ontology versions (V_{old} and V_{new}). Compared to our work, their change detection approach has a number of disadvantages:

- Multiple changes to V_{old} may interfere, thereby possibly invalidating defined change detection rules. This would mean that a rule, as formulated, no longer applies. They have tried to overcome this problem by introducing heuristics to change the precise criteria of the rules to approximations. While heuristics may provide the ontology engineer with some flexibility in the rule definitions, it is clear that it does not offer a bullet-proof solution as it makes the detection process imprecise and unpredictable.
- No direct support for different views on the same ontology evolution is provided.

Finally, the authors of [6] have developed a framework for reasoning with multi-version ontologies. Similar to our approach, a temporal logic has been used as the foundation of the multi-

version reasoning. Although both approaches take a modal-logic approach, there are some differences. The most notable difference is that our approach includes parameterized tense operators that limit the scope of the operator to the evolution of a single concept. The use of parameterized tense operators makes it more convenient to define changes as was clearly illustrated by the second example of Section 5.2. Furthermore, in contrast with their work, our framework includes a formal language (the Change Definition Language) based on this temporal logic that allows users to express change definitions in a general way.

9. Conclusions

In this article, we have presented a change detection approach for the OWL language in the context of ontology evolution.

A change detection phase during ontology evolution is important because it allows detecting changes that were not explicitly requested by an ontology engineer. These are usually higher-level types of changes, which may provide a user of the ontology a better insight on what has changed in the ontology. The change detection approach proposed in this paper is novel because it allows different users to define different sets of change definitions representing the changes they are interesting in, and allows specifying them at an abstraction level that is most appropriate for them. These change definitions are next used to automatically detect which of these changes have occurred. In this way, each user obtains a kind of personalized view on the changes that were applied to the ontology. This will provide a better support for understanding the ontology evolution. To realize this, the change detection approach is based on two key elements: the version log and the Change Definition Language. The purpose of the version log is to keep track of the different versions of all concepts ever created in the ontology. Whenever a concept is added, modified, or removed, the version log is updated to reflect the new state. It is important to note that we keep versions of the individual concepts and not versions of the ontology itself. Of course, an ontology version can be reconstructed from the concept versions. The Change Definition Language is based on a temporal logic and allows users to define the changes in which they are interested. In addition, the change definitions specified by means of this language are used to detect which of these changes have been occurred. This is done by considering the change definitions as temporal queries on the version log. The results of these queries are the individual changes that have been occurred. All occurred changes together, explicit ones as well as detected ones, are called the evolution log. As each user can have its own set of change definitions, each user can also have an own evolution log. This is exactly the advantage of keeping a separate representation of the evolution of an ontology in terms of a version log and an evolution log.

References

[1] J. Allen, Time and time again: the many ways to represent time, Int. J. Intell. Syst. 6 (4) (1991) 341–356.

[2] T. Berners-Lee, J. Hendler, O. Lassila, The semantic web: a new form of web content that is meaningful to computers will unleash a new revolution of possibilities, Sci. Am. 5 (1) (2001).

[3] G. Flouris, D. Plexousakis, G. Antoniou, On applying the AGM theory to DLs and OWL, in: Y. Gil, E. Motta, V. Benjamins, M. Musen (Eds.), Proceedings of the Fourth International Semantic Web Conference (ISWC 2005), Galway, Ireland, 2005, pp. 216–231.

[4] T. Gruber, A translation approach to portable ontology specifications, Knowl. Acquisition, Int. J. Knowl. Acquisition Knowl.-Based Syst. 5 (2) (1993) 199–220.

[5] P. Haase, L. Stojanovic, Consistent evolution of OWL ontologies, in: Proceedings of the Second European Semantic Web Conference (ESWC '05), vol. 3532 of Lecture Notes in Computer Science, 2005, pp. 182–197.

[6] Z. Huang, H. Stuckenschmidt, Reasoning with multi-version ontologies: a temporal logic approach, in: Proceedings of the Fourth International Semantic Web Conference (ISWC 2005), Galway, Ireland, 2005, pp. 398–412.

[7] M. Klein, D. Fensel, Ontology versioning for the Semantic Web, in: Proceedings of the First International Semantic Web Working Symposium (SWWS 2001), Stanford University, California, USA, 2001, pp. 75–91.

[8] M. Klein, D. Fensel, A. Kiryakov, D. Ognyanov, Ontology versioning and change detection on the Web, in: Proceedings of 13th International Conference on Knowledge Engineering and Knowledge Management (EKAW02), Sigüenza, Spain, 2002, pp. 197–212.

[9] M. Klein, Change management for distributed systems, Ph.D. Thesis, 2004.

[10] A. Maedche, L. Stojanovic, R. Studer, R. Volz, Managing multiple ontologies and ontology evolution in OntoLogging, in: Proceedings of the Conference on Intelligent Information Processing (IIP-2002), Montreal, Canada, 2002, pp. 51–63.

[11] P. Plessers, O. De Troyer, Ontology change detection using a version log, in: Proceedings of the Fourth International Semantic Web Conference, Galway, Ireland, Springer-Verlag, 2005, ISBN 978-3-540-29754-3, pp. 578–592.

[12] P. Plessers, O. De Troyer, Resolving inconsistencies in evolving ontologies, in: Y. Sure, J. Domingue (Eds.), Proceedings of the Third European Semantic Web Conference (ESWC 2006), Budva, Montenegro, Springer-Verlag, 2006, pp. 200–214.

[13] P. Plessers, An approach to web-based ontology evolution, Ph.D. Thesis, Vrije Universiteit Brussel, Belgium, 2006.

[14] G. Ozsoyoglu, R. Snodgrass, Temporal and real-time databases: a survey, Knowl. Data Eng. 7 (4) (1995) 513–532.

[15] L. Stojanovic, A. Maedche, B. Motik, N. Stojanovic, User-driven ontology evolution management, in: Proceeding of the Thirteenth European Conference on Knowledge Engineering and Knowledge Management EKAW, Madrid, Spain, 2002, pp. 285–300.

[16] L. Stojanovic, Methods and tools for ontology evolution, Ph.D. Thesis, University of Karlsruhe, Germany, 2004.

Peter Plessers received his Master degree in computer science in 2002 from the Vrije Universiteit Brussel, and his PhD degree from the same university in 2006. He is currently working at the Web and Information System Engineering research lab of the Vrije Universiteit Brussel. His research activities are situated in the field of the Semantic Web and Web engineering. More specifically, he is dealing with research questions surrounding ontology evolution, semantic annotations, Web design methods, accessibility, conceptual modeling, etc.

Olga De Troyer is professor in computer science at the Vrije Universiteit Brussel (Belgium) and head of the WISE research group. She has many years of experience in conceptual modeling and design methods applied to, e.g., databases, Web information systems, Semantic Web applications, and Virtual Reality (VR). Ontologies are used in many aspects of the research: for modeling domain knowledge, for adding semantics to Web applications as well as to VR applications, as internal knowledge representation mechanism, in the context of accessibility and localization, etc. This experience with ontologies has also given rise to more basic research on ontologies.

Sven Casteleyn received his Master degree in computer science from the Vrije Universiteit Brussel in 1999, and his PhD degree from the same university in 2005. His research interests lie primarily within the field of Web engineering, more specifically Website design methods, adaptation and personalization, accessibility, semantic annotations, Semantic Web, ontology evolution, conceptual modeling, etc. Currently, Sven works as a post-doctoral researcher at the Web and Information System Engineering lab of the Vrije Universiteit Brussel (Belgium).