Proactive Prevention of False-positive Conflicts in Distributed Ontology Development

Lavdim Halilaj, Irlán Grangel-González, Maria-Esther Vidal, Steffen Lohmann and Sören Auer
Enterprise Information Systems, Fraunhofer IAIS and University of Bonn, Bonn, Germany

Keywords: Ontology Development, Unique Serialization, Version Control System, Editor Agnostic, RDF, OWL, Turtle.

Abstract: A Version Control System (VCS) is usually required for successful ontology development in distributed settings. VCSs enable the tracking and propagation of ontology changes, as well as collecting metadata to describe changes, e.g., who made a change at which point in time. Modern VCSs implement an *optimistic* approach that allows for *simultaneous* changes of the same artifact and provides mechanisms for automatic as well as manual conflict resolution. However, different ontology development tools serialize the ontology artifacts in different ways. As a consequence, existing VCSs may identify a huge number of *false-positive* conflicts during the merging process, i.e., conflicts that do not result from ontology changes but the fact that two ontology versions are differently serialized. Following the principle of *prevention is better than cure*, we designed *SerVCS*, an approach that enhances VCSs to cope with different serializations of the same ontology. *SerVCS* is based on a unique serialization of ontologies to reduce the number of false-positive conflicts produced whenever different serializations of the same ontology are compared. We implemented *SerVCS* on top of Git, utilizing tools such as Rapper and Rdf-toolkit for syntax validation and unique serialization, respectively. We have conducted an empirical evaluation to determine the conflict detection accuracy of *SerVCS* whenever simultaneous changes to an ontology are performed using different ontology editors. The evaluation results suggest that *SerVCS* empowers VCSs by preventing them from wrongly identifying serialization related conflicts.

1 INTRODUCTION

During the ontology development process, the number, structure, and terminology of the modeled concepts and relations is subject to continuous change. This process, which requires significant efforts and knowledge, is often a collaborative one, involving many people, or even different teams, who are geographically distributed (Palma et al., 2011). The main challenge for the involved ontology engineers is to work collaboratively on a shared objective in a harmonic and efficient way, while avoiding misunderstandings, uncertainty, and ambiguity (Halilaj et al., 2016a). Tracking and propagating the changes made to the ontology to all contributors and thus allowing them to be synchronized with the work of each other is crucial in this process. Therefore, supporting change management is indispensable for successful ontology development in distributed settings.

A *Version Control System* (VCS) assists users in working collaboratively on shared artifacts, and helps to prevent them from overwriting changes made by others. Two prominent mechanisms used to avoid change overwriting are called the *pessimistic* and *optimistic* approach (Mens, 2002). The first is based on the *lock-modify-unlock* paradigm, which implies that modifications to an artifact are permitted only for one user at a time. The second mechanism is based on the *copy-modify-merge* paradigm, where users work on personal working copies, each reflecting the remote repository at a certain time. After the work is completed, the local changes are merged into the remote repository by an *update* command, comprising the phases *comparison*, *conflict detection*, *conflict resolution*, and *merge*.

Different techniques, such as line-, tree-, and graph-based ones, can be employed to compare two versions of the same artifact (Altmanninger et al., 2009). The line-based technique, which achieved wide applicability, compares artifacts line by line, with each line being treated as a single unit. This technique is also known under the terms textual or line-based *comparison* (Mens, 2002). Examples of VCSs that are based on the line-based approach are Subversion, CVS, Mercurial, and Git. Line-based comparisons are applicable on any kind of text artifact,

Halilaj, L., Grangel-González, I., Vidal, M-E., Lohmann, S. and Auer, S.
Proactive Prevention of False-Positive Conflicts in Distributed Ontology Development.
DOI: 10.5220/0006054600430051
In *Proceedings of the 8th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K 2016) - Volume 2: KEOD*, pages 43-51
ISBN: 978-989-758-203-5

as they do not consider syntactical information (Altmanninger et al., 2009). Accordingly, the line-based approach does also neglect syntactical information of ontologies, which are commonly represented in some text-based OWL serialization nowadays.

Challenges emerge when two ontology developers modify the same artifact on their personal working copies in parallel. The modifications might contradict each other, for instance, the developers may both change the name of an ontology concept simultaneously. Such parallel and controversial modifications can result in conflicts during the merging of two ontology versions. In general, a conflict is defined as "a set of contradicting changes where at least one operation applied by the first developer does not commute with at least one operation applied by the second developer" (Altmanninger et al., 2009). Conflicts can be detected by the identification of units that were changed (i.e., added, updated, deleted) in parallel. They can either be automatically resolved or it requires the user to manually fix them by resolving the conflictual changes.

From the ontology development point of view, the situation is exacerbated when different ontology editors are used during the development process. This is due to the fact that these editors often produce different serializations of the same ontology, i.e., the ontology concepts are grouped and sorted differently in the files generated by the editors.[1] As a result, the ability of VCSs to detect the actual changes in ontologies is lowered, since a number of conflicts are detected that are actually not given but are a result of different serializations of the ontology file. In order to increase the accuracy of conflict detection in VCSs, the problem of different groupings and orderings must be tackled.

In this paper, we present *SerVCS*, a generic approach for the realization of optimistic and tool-independent ontology development on the basis of Version Control Systems. As a result, VCSs become *editor agnostic*, i.e., capable to detect actual changes and automatically resolve conflicts using the *built-in* merging algorithms. We implemented and applied the *SerVCS* approach on the basis of the widely used Git VCS. In addition, we developed a middleware service to generate a unique serialization of ontologies before they are pushed to the remote repository. The unique serialization ensures that ontologies have always the same serialization in the remote repository, regardless of the used ontology editor. Therefore, we avoid the incompatibility problem with regard to wrongly detected conflicts resulting from the use of different ontology editors, and assist ontology developers to collaborate more efficiently in distributed environments.

The remainder of this paper is organized as follows: In Section 2, a motivating scenario is presented. In Section 3, our *SerVCS* approach is described. In Section 4, we outline the implementation of *SerVCS*. We evaluate our approach against concrete cases in Section 5 and compare our work with the current state-of-the-art in Section 6. In Section 7, we conclude the paper and provide an outlook to meaningful extensions of this work.

2 MOTIVATING EXAMPLE

As a motivating example, we consider two users working together in developing an ontology for a specific domain. In order to ease collaboration and maintain different versions of the developed ontology that result from changes, they decide to use Git. They proceed by setting up the working environment and creating an initial ontology repository which contains several files. Together, the users define the ontology structure with the most fundamental concepts and upload the ontology file *F* to the *remote repository*. After that, they decide to proceed with their tasks by separately working on their local machines.

The users start synchronizing their local working copies with the remote repository, as illustrated in *Scene 1* of Figure 1. *Scene 2* depicts simultaneous changes performed on different copies of the same ontology file, such as adding new concepts, modifying existing ones, or deleting concepts. For realizing this task, different ontology editors are used. In our case, *User 1* works with *Desktop Protégé*[2], whereas *User 2* prefers to edit the ontology with *TopBraid Composer*[3]. After finishing the task, *User 1* uploads her personal working copy (*F**) to the *remote repository*, as shown in *Scene 3*. Next, *User 2* completes his task and starts uploading the changes he made on his local copy to the *remote repository*. While trying to trigger this action, he receives a rejection message from the VCS, listing all changes which result in conflicts, as depicted in *Scene 4*. These conflicts need to be resolved in order for the VCS to allow the user to successfully upload his version (*F***) to the *remote repository*. *User 2* starts resolving the conflicts manually by comparing his version of the ontology with the one of *User 1* that has already been uploaded to the remote repository. Since the users are working with

[1] With "different serializations", we refer to two different ontology files that represent the same ontology using the same syntax (RDF/XML, Turtle, Manchester, etc.) but use a different structure to list and group the ontology concepts.

[2] http://protege.stanford.edu

[3] http://www.topquadrant.com/composer/

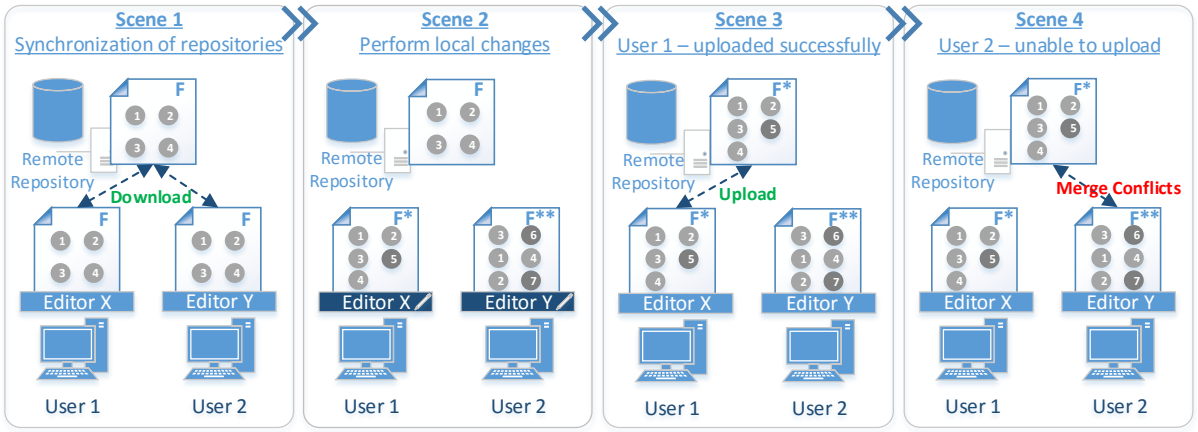

Figure 1: Illustration of the problem that results from the use of different ontology editors.

different ontology editors that use each its own serialization when saving the ontology file, the files are differently organized. For instance, while the concepts in one of the files are grouped into categories, such as *Classes* and *Properties*, they are ordered alphabetically in the other case, without any grouping. Consequently, the information about actual changes, i.e., concrete changes on the ontology performed by each user, can no longer be detected by the line-based comparison of the VCS, but a huge number of conflicts result that are due to the different organization of the ontology files. This prevents *User 2* from merging his changes, and his version of the ontology cannot be uploaded to the remote repository.

This scenario illustrates that, despite the various benefits provided by a VCS for collaborative ontology development, it has not been possible so far to effectively use a VCS in cases where different editors and ontology serializations are used. This is changed with the *SerVCS* approach we present in this paper.

3 APPROACH

In this section, we define the basic terminology and provide a formal description of the *SerVCS* approach. Formally, an RDF document A is defined as $A \subset (\mathbf{I} \cup \mathbf{B}) \times \mathbf{I} \times (\mathbf{I} \cup \mathbf{B} \cup \mathbf{L})$, where $\mathbf{I}$, $\mathbf{B}$, and $\mathbf{L}$ correspond to sets of *IRI*s, *blank nodes*, and *literals* (typed and untyped), respectively (Gutierrez et al., 2011).

Definition 1 (Changeset)**.** *Given two RDF documents A and A^*, a changeset of A^* with respect to A is defined as follows:*

$ChangeSet(A^*/A) = (\delta^+(A^*/A), \delta^-(A^*/A), <)$, *where*

- $\delta^+(A^*/A) = \{t \mid t \in A^* \wedge t \notin A\}$,
- $\delta^-(A^*/A) = \{t \mid t \in A \wedge t \notin A^*\}$, *and*
- $<$ *is a partial order between the RDF triples in* $\delta^+(A^*/A) \cup \delta^-(A^*/A)$.

Example 1. *Consider two RDF documents* $A = \{t_1, t_2, t_3\}$ *and* $A^* = \{t_1, t_2, t_4\}$ *such that* A^* *is a new version of* A *where the RDF triple* t_4 *was added and the triple* t_3 *was deleted. Then, the changeset of* A *with respect to* A^*, $ChangeSet(A^*/A)$, *is as follows:*

- $\delta^+(A^*/A) = \{t_4\}$,
- $\delta^-(A^*/A) = \{t_3\}$, *and*
- $<= \{(t_4, t_3)\}$.

Definition 2 (Syntactic Conflicts)**.** *Given two RDF documents A and A^*, and the changeset of A^* with respect to A, $ChangeSet(A^*/A) = (\delta^+(A^*/A), \delta^-(A^*/A), <)$, there is a syntactical conflict between A and A^* iff there are RDF triples t_i and t_j such that:*

- $t_i \in \delta^-(A^*/A)$,
- $t_j \in \delta^+(A^*/A)$,
- $(t_i, t_j) \in <$, *and*
- $t_i = (s, p, o_i)$, $t_j = (s, p, o_j)$, *and* $o_i \neq o_j$.

Example 2. *Consider two RDF documents A and A^* with triples* t_3=(`:Bus, rdfs:label, "Bus"@en`) *and* t_4=(`:Bus, rdfs:label, "Buss"@en`). *Since the object value of the property* `rdfs:label` *of the subject* `:Bus` *has been changed, there is a syntactic conflict between the RDF documents A and A^*.*

Definition 3 (RDF Document Serialization)**.** *Given an RDF document A and an ordering criteria η, a* serialization *of A according to η, $\Gamma(A, \eta)$ corresponds to an ordering of the triples in A according to η:*

$$\Gamma(A, \eta) = < t_1, t_2, ..., t_n >$$

Example 3. *Suppose three RDF triples t_1, t_2, and t_3 are defined as follows in an RDF document A:* t_1=(`:Car, rdfs:label, "Car"@en`),

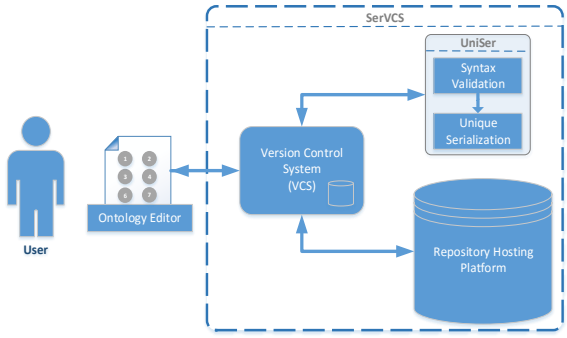

Figure 2: SerVCS architecture: (1) the VCS handles different RDF document versions via changesets; (2) the UniSer component generates unique serializations; (3) the repository hosting platform stores the RDF documents and propagates the changes.

t_2=(`:Truck, rdfs:label, "Truck"@en`), *and* t_3=(`:Bus, rdfs:label, "Bus"@en`), *respectively. A serialization* $\Gamma(A,\eta)$ *of A listing the triples by their labels in alphabetical order* η *would be:*

$$\Gamma(A,\eta) =< t_3,t_1,t_2 >$$

Definition 4 (False-Positive Conflicts)**.** *Given two RDF documents A and* A^* *such that* F_1 *and* F_2 *are serializations of A and* A^* *according to some ordering criteria* η_1 *and* η_2*, respectively. There is a* false-positive conflict *between* F_1 *and* F_2*, iff there exist* η *ordering criteria such that:*

$$\Gamma(A,\eta) = \Gamma(A^*,\eta) \text{ and } F_1 \neq F_2$$

Example 4. *Consider serializations* $F_1 =< t_1,t_3,t_2 >$ *and* $F_2 =< t_2,t_1,t_3 >$ *both representing two identical RDF documents* $A = A^*$*, respectively, such that* $A = \{t_1,t_2,t_3\}$*. Then, there are three false-positive conflicts between* F_1 *and* F_2*, because there exist ordering criteria* η, $\Gamma(A,\eta) = \Gamma(A^*,\eta)$.

3.1 SerVCS

With the objective of enabling ontology development in distributed environments, where sets of changes are performed (cf. Definition 1) using different editors, the detection of *False-Positive Conflicts* (cf. Definition 4) by the VCS must be avoided. For this reason, ontologies should have a *unique serialization* (see Definition 3). In order to realize that, we developed *SerVCS*, which generates a unique serialization of ontologies regardless of the used editing tool. The modeled concepts (triples) are ordered alphabetically in this unique serialization, first according to the *subject* name, then by *property* name. That way, ontologies (represented as text-based RDF documents) have always a consistent serialization in the remote repository. As a result, a high accuracy of conflict detection can be achieved and the identified conflicts are reduced to those caused by overlapping changes, *Syntactical Conflicts* (cf. Definition 2). This enables a VCS to automatically resolve most conflicts using its built-in algorithms. In the worst case, a user is confronted with conflicting changes and has to manually resolve them by providing a valid and consistent ontology. Since all ontologies have a unified serialization in the remote repository, the user is able to see the differences between any two versions of the ontology. Figure 2 illustrates the *SerVCS* architecture, which consists of three main components: (1) a VCS, which handles different RDF document versions via changesets; (2) a UniSer component, which generates unique serializations for the RDF documents; and (3) a repository hosting platform, which stores the RDF documents and propagates the changes.

Figure 3 depicts the ontology development workflow using the *SerVCS* approach. After personal working copies are synchronized with the remote repository (cf. *Scene 1* of Figure 1), users start performing their tasks using different ontology editors. When making any changes, such as adding, removing, or modifying existing concepts, the updated ontology is saved locally on the machine of the user, as illustrated in Figure 3, *Scene 2* (which is still identical to *Scene 2* of Figure 1). Next, these changes are uploaded to the remote repository. *Scene 3* shows that a unique serialization of the ontology is created as intermediate step. As a result, the concepts are organized using a common ordering criteria. In *Scene 4*, *User 1* uploads her changes successfully to the remote repository. Lastly, as illustrated in *Scene 5*, *User 2* starts uploading his changes to the remote repository. Since the ontology has a unified serialization, the VCS can merge both versions. In case of overlapping changes, the VCS shows exactly the lines which resulted in conflicts. Formally, a list of conflicts *LC* identified by *SerVCS* is defined as follows:

Definition 5 (List of Conflicts)**.** *Given two RDF documents A and* A^* *such that* F_1 *and* F_2 *are serializations of A and* A^* *according to ordering criteria* η_1 *and* η_2*, a list LC* $=< c_1,\ldots,c_n >$ *of conflicts between* F_1 *and* F_2*, identified by* SerVCS*, comprises triples* $c_i = (i, entry_{i1}, entry_{i2})$*:*

- $i \in [1, MIN(size(F_1), size(F_2))]$,
- $entry_{i1} = (s_{i1}, p_{i1}, o_{i1})$ *and* $entry_{i2} = (s_{i2}, p_{i2}, o_{i2})$ *are RDF triples at the position i in* F_1 *and* F_2*, respectively,*
- $entry_{i1}$ *and* $entry_{i2}$ *are different, i.e.,* $s_{i1} \neq s_{i2}$ *or* $p_{i1} \neq p_{i2}$ *or* $o_{i1} \neq o_{i2}$.

Theorem 1. *Given serializations* F_1 *and* F_2 *according to ordering criteria* η *of RDF documents A and* A^**, respectively. Consider LC* $=< c_1,\ldots,c_n >$ *the list of*

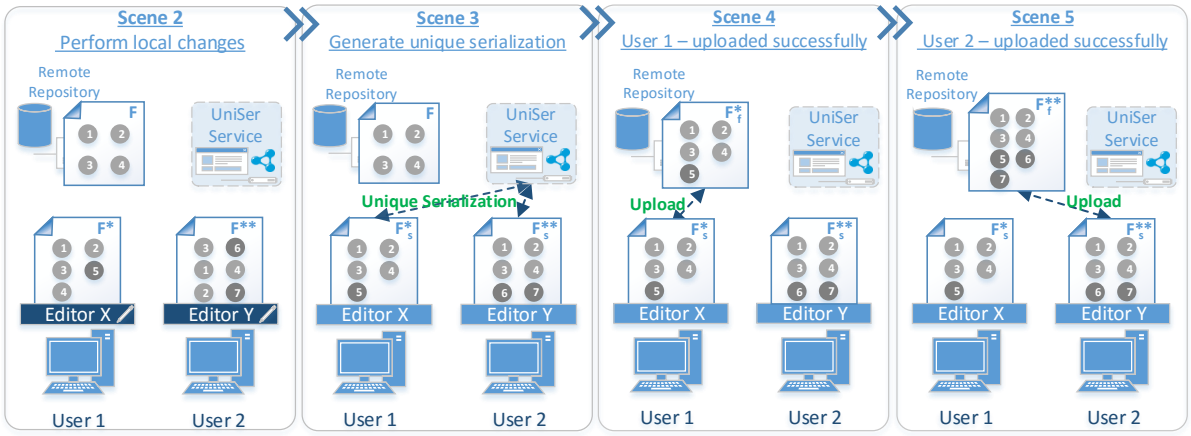

Figure 3: The ontology development workflow using SerVCS.

conflicts between F_1 and F_2 identified by SerVCS. *If there are only syntactical conflicts between A and A^**[4], *then for all $c_i = (i, entry_{i1}, entry_{i2}) \in LC$*

- $entry_{i1} = (s, p, o_{i1})$ *and* $entry_{i2} = (s, p, o_{i2})$, *and*
- $o_{i1} \neq o_{i2}$.

Proof. We proceed with a proof by contradiction. Assume that there are only syntactical conflicts between A and A^*, and there is a conflict c_i in LC, such that $c_i = (i, (s_{i1}, p_{i1}, o_{i1}), (s_{i2}, p_{i2}, o_{i2}))$, and $s_{i1} \neq s_{i2}$ or $p_{i1} \neq p_{i2}$. Since F_1 and F_2 are serializations according to the same ordering criteria η, $entry_{i1} \in \delta^-(A^*/A)$ and $entry_{i2} \in \delta^+(A^*/A)$. However, the statement $s_{i1} \neq s_{i2}$ or $p_{i1} \neq p_{i2}$ contradicts the fact that only syntactical conflicts exist between A and A^*. □

4 IMPLEMENTATION

We implemented the architecture depicted in Figure 2 to empower VCSs for preventing wrongly indicated conflicts.

4.1 Version Control System

Git[5] is used as the Version Control System, i.e., Git is responsible for managing different versions of the RDF documents. Furthermore, the Git *hook* mechanism is used to automatize the process of generating the unique serialization of the ontologies before they are pushed to the remote repository. Once the modification of the ontology is finished, it is added to the Git *stage* phase. The next step proceeds with committing the current state to the personal working copy. The initialization of the commit event triggers a hook named *pre-commit*. This hook is adapted with a new workflow to handle the process of automatically generate a unique serialization, apart from the default one provided by Git. SerVCS uses *Curl*[6] as command-line HTTP client to send the modified files to the *UniSer* service. In case that ontologies fail to pass the integrated validation process, the commit is aborted and a corresponding error message is shown to the user. Otherwise, the files are organized according to the unique serialization. Subsequently, newly generated content overwrites the current content of the files by replacing the old serialization created by the ontology editor with the new unique serialization created by *UniSer*. When no error occurs during the entire process, the pre-commit hook event is completed and the commit is applied successfully. As a result, a new revision of the modified ontologies is created and the user is able to further proceed with successfully pushing her version to the remote repository.

In addition, *Github*[7] is used as hosting platform for the repository to ease the collaborative development among several contributors.

4.2 UniSer

Furthermore, we implemented a stand-alone service, *UniSer*, using the cross-platform JavaScript runtime environment *NodeJS*[8]. Other tools are integrated to

[4]Given two RDF-documents A and A^*, and $ChangeSet(A^*/A) = (\delta^+(A^*/A), \delta^-(A^*/A), <)$, there are only syntactical conflicts between A and A^*, iff $size(A^*) = size(A)$, and for each RDF triples t_i and t_j:

- $t_i \in \delta^-(A^*/A)$ and $t_j \in \delta^+(A^*/A)$,

then, there is a pair $(t_i, t_j) \in <$, and $t_i = (s, p, o_i)$, $t_j = (s, p, o_j)$, and $o_i \neq o_j$.

[5]https://git-scm.com

[6]https://curl.haxx.se

[7]https://github.com

[8]https://nodejs.org

realize the tasks required for this service, e.g., syntax validation and unique serialization. The service accepts the ontology files as input through an HTTP interface and returns to the client either the error message from the validation process or the unique serialization of the file. Once the input is received, *UniSer* validates the ontology, since a prerequisite for the unique serialization process is that ontology files are free of syntactic errors. The syntax validation is performed by *Rapper*[9]. In case of errors, a detailed report comprising the file name, error type, and error line is returned to the client. Otherwise, the process continues with creating the unique serialization using Rdf-toolkit[10]. During this task, a unified serialization of the ontology file is created by (1) grouping the elements into categories, such as classes, properties, and instances, and (2) ordering the elements within the categories alphabetically. The unique serialization of the ontology is send back to the client as final outcome.

In the following, we give some serializations of a simple ontology in *Turtle*[11] format comprising two concepts: a *Bus* class and a *MiniBus* instance.

```
@prefix :     <http://example.com/> .
@prefix rdf:  <http://.../22-rdf-syntax-ns#> .
@prefix rdfs: <http://.../rdf-schema#> .
@prefix owl:  <http://www.w3.org/2002/07/owl#> .

:Bus     rdf:type     owl:Class ;
         rdfs:comment "Bus"@en;
         rdfs:label   "Bus"@en .

:MiniBus rdf:type     :Bus ;
         rdfs:label   "MiniBus"@en .
```

The above excerpt serialized with Protégé is shown as follows:

```
@prefix :     <http://example.com/> .
@prefix rdf:  <http://.../22-rdf-syntax-ns#> .
@prefix rdfs: <http://.../rdf-schema#> .
@prefix owl:  <http://www.w3.org/2002/07/owl#> .

#################################################
#        Classes
#################################################

###      http://example.com/Bus

:Bus     rdf:type     owl:Class ;
         rdfs:label   "Bus"@en ;
         rdfs:comment "Bus"@en .

#################################################
#        Individuals
#################################################

###      http://example.com/Bus1

:MiniBus rdf:type   :Bus ;
         rdfs:label "MiniBus"@en .
```

The following listing depicts the same ontology serialized with TopBraid Composer:

```
@prefix : <http://example.com/> .
@prefix rdf:  <http://.../22-rdf-syntax-ns#> .
@prefix rdfs: <http://.../rdf-schema#> .
@prefix owl:  <http://www.w3.org/2002/07/owl#> .

:Bus
    a            owl:Class ;
    rdfs:label   "Bus"@en ;
    rdfs:comment "Bus"@en ;
    .

:MiniBus
    a          :Bus ;
    rdfs:label "MiniBus"@en ;
    .
```

Using the *UniSer* service, the excerpt of the ontology is generated according to the unique serialization. The following listing depicts the result after the serialization by UniSer (which is nearly identical to the serialization of TopBraid Composer in this case).

```
@prefix : <http://example.com/> .
@prefix rdf:  <http://.../22-rdf-syntax-ns#> .
@prefix rdfs: <http://.../rdf-schema#> .
@prefix owl:  <http://www.w3.org/2002/07/owl#> .

:Bus
    a              owl:Class ;
    rdfs:label     "Bus"@en ;
    rdfs:comment   "Bus"@en ;
    .

:MiniBus
    a          :Bus ;
    rdfs:label "MiniBus"@en ;
    .
```

5 EVALUATION

We conducted an empirical evaluation to assess the usefulness of the *SerVCS* approach. During this evaluation, the following two hypotheses were tested:

H1. Is the number of false-positive conflicts between two RDF documents that are modified by differ-

[9] http://librdf.org/raptor/rapper.html
[10] https://github.com/edmcouncil/rdf-toolkit
[11] https://www.w3.org/TR/turtle/

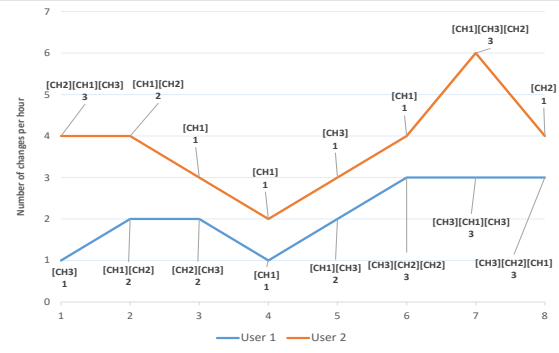

Figure 4: Number and types of changes per user in an interval of 8 hours. A Poisson distribution with $\lambda = 2$ models an average of two changes per hour. A uniform distribution with replacement is followed to sample the type of changes.

ent ontology editors reduced when using *SerVCS* compared to when it is not used?

H2. Are users able to effectively resolve the indicated conflicts and collaborate with different ontology editors when *SerVCS* is applied?

5.1 Experimental Setup

Ontology Generation. An ontology with a given number of RDF triples is considered as initial input. Then, changes are randomly generated following a *Poisson distribution*, i.e., we simulate changes performed by users assuming that these changes obey a *Poisson distribution*. The parameter λ indicates the average number of changes per time interval. It is set to two ($\lambda = 2$) to simulate the number of expected changes that users in the experiment may perform per hour during a period of eight hours. Figure 4 illustrates the number and types of changes per hour for two users. To ensure that our evaluation represents as much as possible a real usage scenario, we used a list of basic changes typically performed in ontology development, as given in Table 1. These changes are randomly chosen following a *uniform distribution* with replacement. We consider the change type *modification* to be a combination of *deletion* and *addition*.

Metrics. We report on the number of conflicting lines (**NCL**). It is computed as the number of conflicts indicated by Git during the merge process of two versions of the ontology after each hour, and corresponds to the cardinality of the list of conflicts LC (cf. Definition 5).

Gold Standard. We compute the gold standard by summing up the number of conflicting lines, which corresponds to the cardinality of overlapping changes made by users in a specific hour (cf. Definition 2).

Implementation. Experiments were run on a Linux Ubuntu 14.04 machine with a 4th Gen Intel Core i5-4300U CPU, 3MB Cache, 2.90GHz with 8GB RAM 1333MHz DDR3. *SerVCS* was implemented using NodeJS version 4.4.5. The syntax validation was performed using Rapper version 2.0.15. The *unique serialization* was created using Rdf-toolkit version 1.3.0. The used Git version was 1.9.1. The change generator was implemented using RStudio version 0.99.902[12].

5.2 Effectiveness of SerVCS

In order to test hypotheses **H1** and **H2**, two users were chosen and asked to use different ontology editors. *User 1* worked with TopBraid, whereas *User 2* worked with Protégé. The evaluation was conducted in two scenarios in a controlled environment. In the first scenario, the two users worked purely with the functionalities of Git. In the second scenario, they used *SerVCS* along with Git as VCS. We asked the users to keep a log of the changes made to the ontology during the experiment. In total, they made 30 changes: 11 additions, 9 modifications, and 10 deletions. The distributions of changes per user are aligned with the Poisson distribution shown in Figure 4. The complete history of changes, including the conflicts that occurred in both scenarios, is available on GitHub[13]. Figure 5 shows the number of conflicting lines (NCL) detected in each scenario (using plain Git and Git with *SerVCS*) compared to the *gold standard*. It can be observed that the number of conflicts is significantly reduced when the *SerVCS* approach is used. These results support hypotheses **H1** and **H2**.

5.3 Discussion

Figure 5 shows that the number of conflicting lines (NCL) wrongly indicated by Git is much higher compared to *SerVCS*, i.e., the number of false-positive conflicts of Git is higher. This negative performance of Git is caused by both: different serializations of the ontologies generated by the ontology editors, and the line-based comparison implemented in Git. Contrary, *SerVCS* exhibits much better performance and is able to considerably reduce the number of false-positive conflicts, which validates hypothesis **H1**. *SerVCS* performs better because the modified ontologies are serialized using a *unique serialization* where concepts are alphabetically sorted before they are pushed to Git.

[12] https://www.rstudio.com/products/RStudio/
[13] https://github.com/lavdim/unistruct

Table 1: Basic changes in ontology development.

ID	Change Type	Description	Example
CH1	Addition	Adding new elements like classes and properties	Add a new class, e.g., the class *Car* with properties *rdfs:label* and *rdfs:comment*
CH2	Modification	Modifying existing elements	Modify a property value, e.g., *rdfs:label* of *Buss* class
CH3	Deletion	Deleting existing elements	Delete an instance, e.g., the *MiniBus* instance if it exists

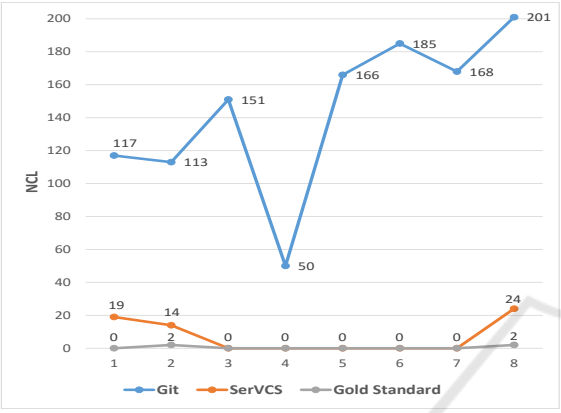

Figure 5: Number of conflicting lines (**NCL**) indicated by Git and SerVCS compared to the gold standard.

However, as shown in Figure 5, *SerVCS* may also wrongly identify conflicts, i.e., the number of false-positive conflicts is not zero. This happens whenever users concurrently modify the subject or predicate of an RDF triple, i.e., a non-syntactical conflict is generated and Definition 1 is not satisfied.

Since the number of conflicting lines (NCL) indicated by *SerVCS* is smaller, users are able to quickly and easily resolve the reported conflicts. Thus, *SerVCS* enables users to effectively resolve the indicated conflicts and collaborate with different ontology editors, which validates hypothesis **H2**.

6 RELATED WORK

Recently, there has been some research investigating the use of different ontology editors in distributed ontology development using version control systems.

We first discuss approaches focusing on providing version management for ontologies in the collaborative development process. An ontology for unique identification of changes between two RDF graphs is presented by (Lee and Connolly, 2001). To recognize these changes, a pretty-printed version of the RDF graphs is also utilized. The authors distinguish two types of deltas, which can be applied in form of patches to the RDF graphs. Firstly, *weak* deltas, which are directly applied to the graph from where they are computed. Secondly, *strong* deltas, which specify the changes independently from the context. In contrast to SerVCS, the proposed approach focuses on the semantic representation of changes and its application to RDF graphs.

(Völkel and Groza, 2006) present *SemVersion*, an RDF-based system for ontology versioning. It is based on the two core components *data management* and *versioning functionality*. The first is responsible for the storage and retrieval of data chunks. The second deals with specific features of the ontology language, such as structural and semantic differences. To find semantic differences between two versions, e.g., whether a statement has been added or removed, SemVersion uses a simplified heuristic method for conflict detection.

A holistic approach for collaborative ontology development based on the ontology change management is described by (Palma et al., 2011). It comprises different strategies and techniques to realize collaborative processes in inter-organizational settings, such as centralized, decentralized, and hybrid ones.

(Edwards, 1997) proposes techniques for managing *high-level* application-defined conflicts. Consequently, the introduced mechanisms should be able to handle conflict resolutions. Further, certain types of conflicts can be tolerated and others forbidden according to the specified application requirements.

All these works rely on their own version control mechanisms tailored for ontology development. Therefore, they lack rich features provided by generic VCSs, such as branching and merging. Our solution targets generic VCS by enriching them with features for avoiding wrongly indicated conflicts.

Next, we look at approaches whose main focus is on overcoming the problem of wrongly indicated conflicts. Several efforts have been made in the field of *Model-Driven Development*, where the *model* itself is the main artifact. (Altmanninger, 2007) describes an approach for semantically enhancing VCS's allowing semantic conflict detection for models. Using the semantic views concept to explain aspects of a modeling

language, a better conflict detection is achieved and the reason of the conflict can be easily determined. (Brosch, 2009) suggests using a model checker for detecting semantic merge conflicts of an evolving UML sequence diagram. When an automatic merge is not possible due to conflicting changes, additional redundant information essential for the models is used to determine invalid solutions. By using this technique, it is possible to assert the concrete modifications realized in a sequence diagram.

In contrast to these works, SerVCS focuses on ontologies as main artifacts. It utilizes the functionality of available VCSs to merge versions of the same ontology created by different editors.

7 CONCLUSION

We presented *SerVCS*, an approach for empowering VCSs to deal with various serializations of the same ontology. As a result, the number of false-positive conflicts is reduced allowing users to collaboratively develop ontologies in a distributed and multi-editor environment. We conducted an empirical evaluation to study the effectiveness of *SerVCS* in comparison to existing VCSs, in this case Git. The results suggest that *SerVCS* is able to reduce the number of false-positive conflicts whenever users work concurrently using different ontology editors on the same ontology.

As one of the next steps, SerVCS will be added as a service to VoCol (Halilaj et al., 2016b). VoCol is an integrated environment for developing ontologies and vocabularies in distributed settings. Along with other services, such as documentation generation, visualization, and evolution tracking, VoCol facilitates ontology and vocabulary development using version control systems. It enables people with different knowledge backgrounds to develop ontologies and vocabularies collaboratively.

Future work also concerns the development of new mechanisms for improving conflict detection and resolution. Further, we plan to add a semantic layer to *SerVCS* in order to prevent semantic inconsistencies generated after merging two ontology versions. Finally, we plan to conduct a more comprehensive evaluation of the effectiveness of conflict prevention in terms of accuracy and usefulness.

ACKNOWLEDGEMENTS

This work has been supported by the German Federal Ministry of Education and Research (BMBF) in the context of the projects LUCID (grant no. 01IS14019C) and SDI-X (grant no. 01IS15035C).

REFERENCES

Altmanninger, K. (2007). Models in conflict – A semantically enhanced version control system for models. In *Doctoral Symposium at the ACM/IEEE 10th International Conference on Model-Driven Engineering Languages and Systems*, CEUR-WS 262. CEUR-WS.org.

Altmanninger, K., Seidl, M., and Wimmer, M. (2009). A survey on model versioning approaches. *International Journal of Web Information Systems*, 5(3):271–304.

Brosch, P. (2009). Improving conflict resolution in model versioning systems. In *Companion Volume of the 31st International Conference on Software Engineering (ICSE '09)*, pages 355–358. IEEE.

Edwards, W. K. (1997). Flexible conflict detection and management in collaborative applications. In *10th Annual ACM Symposium on User Interface Software and Technology (UIST '97)*, pages 139–148. ACM.

Gutierrez, C., Hurtado, C. A., Mendelzon, A. O., and Pérez, J. (2011). Foundations of semantic web databases. *Journal of Computer and System Sciences*, 77(3):520–541.

Halilaj, L., Grangel-González, I., Coskun, G., Lohmann, S., and Auer, S. (2016a). Git4voc: Collaborative vocabulary development based on git. *International Journal on Semantic Computing*, 10(2):167–192.

Halilaj, L., Petersen, N., Grangel-González, I., Lange, C., Auer, S., Coskun, G., and Lohmann, S. (2016b). Integrated environment to support version-controlled vocabulary development. In *20th International Conference on Knowledge Engineering and Knowledge Management (EKAW 16)*. Springer, to appear.

Lee, T. B. and Connolly, D. (2001). Delta: an ontology for the distribution of differences between rdf graphs. Technical report, W3C.

Mens, T. (2002). A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462.

Palma, R., Corcho, Ó., Gómez-Pérez, A., and Haase, P. (2011). A holistic approach to collaborative ontology development based on change management. *Journal of Web Semantics*, 9(3):299–314.

Völkel, M. and Groza, T. (2006). SemVersion: An RDF-based ontology versioning system. In *IADIS International Conference on WWW/Internet (IADIS '06)*, pages 195–202. IADIS.