Git4Voc: Git-based Versioning for Collaborative Vocabulary Development

Lavdim Halilaj	Irlán Grangel-González	Gökhan Coskun	Sören Auer
EIS Department	EIS Department	EIS Department	EIS Department
University of Bonn	University of Bonn	University of Bonn	University of Bonn
Bonn, Germany	Bonn, Germany	Bonn, Germany	Bonn, Germany
halilaj@cs.uni-bonn.de	grangel@cs.uni-bonn.de	coskun@cs.uni-bonn.de	auer@cs.uni-bonn.de

Abstract—Collaborative vocabulary development in the context of data integration is the process of finding consensus between the experts of the different systems and domains. The complexity of this process is increased with the number of involved people, the variety of the systems to be integrated and the dynamics of their domain. In this paper we advocate that the realization of a powerful version control system is the heart of the problem. Driven by this idea and the success of Git in the context of software development, we investigate the applicability of Git for collaborative vocabulary development. Even though vocabulary development and software development have much more similarities than differences there are still important differences. These need to be considered within the development of a successful versioning and collaboration system for vocabulary development. Therefore, this paper starts by presenting the challenges we were faced with during the creation of vocabularies collaboratively and discusses its distinction to software development. Based on these insights we propose Git4Voc which comprises guidelines how Git can be adopted to vocabulary development. Finally, we demonstrate how Git hooks can be implemented to go beyond the plain functionality of Git by realizing vocabulary-specific features like syntactic validation and semantic diffs.

Keywords-version control system; collaborative vocabulary development; git;

I. Introduction

One of the key obstacles for the wider deployment of semantic technologies is the lack of comprehensive vocabularies. This is because vocabulary development requires a significant investment, which is difficult to make by a single person or organisation. If we look at current vocabularies (e.g. LOV[1]), we observe that they are rather simplistic. For a total of 457 vocabularies listed in LOV, a straightforward query against the LOV SPARQL endpoint tells us that the average number of classes for each vocabulary is 42 whereas the average number of properties is 59. Omitting the four vocabularies with the highest number of classes and properties, these figures decrease to 31 classes and 37 properties on average. We also observe that a large number of crucial domains is not or only superficially covered by existing vocabularies. One of the main reasons for the lack of vocabularies is also the lack of adequate methodological and tool support.

At the same time, the problem of integrating data from different systems receives ever-increasing attention. Identifying the main terms across heterogeneous data sources by finding a consensus between the developers and defining a shared vocabulary is an effective approach to tackle this problem. However, this process, which we refer to as *collaborative vocabulary development*, itself is a complex problem to be solved. In fact, the main challenge for the vocabulary engineers is to work collaboratively on a shared objective in a harmonic and efficient way while avoiding misunderstandings, uncertainty and ambiguity. The quality of the produced vocabularies is another challenge that should be tackled as well. In [1], we identified and elaborated important aspects for vocabulary development such as: reuse, vocabulary structure, naming conventions, multilinguality, documentation, validation and authoring. These aspects are relevant from collaborative point of view as well. Taking into consideration above aspects will impact the quality of vocabulary itself.

Therefore, finding a suitable collaboration methodology is exacerbated by the number and diversity of the involved stakeholders as well as the complexity of the domains. Due to the open, distributed and participatory nature of the Web, such a solution is of paramount interest for the Semantic Web community.

Our approach to tackle the mentioned problem is to focus on supporting the collaborative vocabulary development with a well-known method for distributed version control in a domain-agnostic way. In this regard, we have chosen *Git* for the following two reasons. On the one hand, Git is a mature version control system supported by sophisticated tools and broadly used in software development projects. More than 10 million repositories[2] are hosted on GitHub for open source and commercial projects [2]. On the other hand, existing popular vocabularies like *schema.org*[3], *Description of a Project* (DOAP)[4], the *music ontology*[5] publish their efforts in GitHub to leverage the contribution of the community. This indicates that the vocabulary development community is already familiar with Git.

The remainder of this paper is structured as follows: In section II we present a comprehensive list of requirements

---

[1]http://lov.okfn.org

[2]https://github.com/blog/1724-10-million-repositories

[3]https://github.com/schemaorg/schemaorg

[4]https://github.com/edumbill/doap

[5]https://github.com/motools/musicontology

aggregated from the current state of the art and our ongoing work on *MobiVoc*[6] and *SCORVoc*[7]. In section III we present *Git4Voc* which comprises guidelines how Git can be used for collaborative vocabulary development. With Git4Voc we propose to utilize Git's hooks mechanism to realize vocabular-specific features. In section IV we demonstrate concrete example of hook implementations. We provide an overview about related work in section V. The conclusion and an outlook to future work are presented in section VI.

II. REQUIREMENTS OF COLLABORATIVE VOCABULARY DEVELOPMENT

Collaborative vocabulary development is considered to be very related to the broad field of software development. In fact, most proposals for supporting the former are inspired by experiences in the latter. However, a vocabulary is not totally equal to software code. The development of vocabularies raises challenges which are new and not or at least not to that extend raised during software development. In this section we focus on requirements which are more critical for vocabularies. We gathered these requirements by aggregating insights from the current state of the art and our own experiences during the development of MobiVoc[8] and SCORVoc[9]. In the following these requirements are presented in detail.

Communication support (R1) Collaborative vocabulary development is about finding consensus between members of a team. In order to share ideas and finding agreements, communication among the contributors is essential [3]. During the whole life cycle, especially in agile development, supporting and recording discussions, changes and their reasons are crucial [4]. This is especially very important in the case of heterogeneous teams with experts from different domains. Some critical examples to be communicated within a team are introducing new elements, extending or modifying the subsumption hierarchy, integration of external resources and changing the underlying semantic expressivity [5]. An effective communication has a significant impact on the quality of the collaboration and its outcome.

Provenance of information (R2) In collaborative development the capability to track the changes made by contributors is an important feature [4]. This is due to the fact that each change in the vocabulary reflects the understanding of the authors regarding the domain. In case of disagreements, it is necessary to know which change was made by whom at which time and for what reason.

Different roles (R3) Creating vocabularies with the purpose of realizing data integration across heterogeneous independent systems, involves domain experts from various fields with different levels of expertise. For instance, in large projects like the *Gene Ontology*[10] (GO) many participants and curators take part in the development process. Most participants can only

add comments and discuss terms. A core team is allowed to edit the main components of the vocabulary by adding modules, classes, properties, removing terms and performing refactoring. For that reason, there is a need for the definition of roles along with the permissions [5], [4], [6], [7].

Workflow independence (R4) The overall field of methodologies and workflows for collaborative vocabulary development is changing continuously [4]. To the best of our knowledge, there are no established methodologies nor workflows which are broadly applied. Tools supporting collaboration should be generic and be able to adapt in highly dynamic context. Therefore, it is important that a system is flexible enough to be used within different methodologies and workflows.

Quality assurance (R5) Developing vocabularies includes many requirements of quality assurance. Syntax and semantic correctness as well as the application of best practices on designing vocabularies are some of the quality aspects. Therefore providing tool support is a significant feature to prevent contributors from making errors. Later correcting phases might lead to a wasting of resources in terms of time and money.

Documentation generation (R6) As mentioned before, a team for vocabulary development comprises domain experts with less technical expertise in knowledge representation and engineering tools. In order to enable them contributing to the development process, providing user friendly view to the current state of the vocabulary is vital. Therefore, an automatic documentation generation feature is necessary.

Deltas among versions (R7) Collaborative development of vocabularies should respond to the evolution of the knowledge domain [7]. It should also respect the evolution of connected vocabularies within the Linked Data Cloud, in order to avoid semantic inconsistencies. Therefore, support for detecting and documenting the semantic difference between versions is needed, to enable developers to understand the mentioned evolutions. This includes the modification, the addition of new elements (i.e. classes, properties) as well as the removal of existing terms. Authors of well-known vocabularies such as *SKOS*[11] and *schema.org*[12] publish release notes containing what has been changed among different versions.

Editor agnostic (R8) In contrast to software code, vocabularies are abstract artefacts which can be serialized with different techniques. Since contributors can use different editors which style the syntax in different ways, the support of the collaboration must be editor agnostic and syntax independent.

Modularity (R9) Modularization is recognized as an important step in collaborative vocabulary building [8]. Reusability, the decrease of complexity, ownership and customization are some of the benefits of vocabulary modularization. Some studies report that there is no universal way to perform this process and that the choice of a particular technique should be guided by application specific requirements [9]. In contrast, other reports show that a module in a mid-sized vocabulary should contain between 200 and 300 lines of code [10]. Especially

---

[6]https://github.com/vocol/mobivoc

[7]http://purl.org/eis/vocab/scor

[8]https://github.com/vocol/mobivoc

[9]http://purl.org/eis/vocab/scor

[10]http://www.geneontology.org/

[11]http://www.w3.org/2004/02/skos/history

[12]http://schema.org/docs/releases.html

in an agile development process with large vocabularies and many contributors, it is of paramount importance that the system provides means to support the modularization activity.

Multilinguality (R10) In order to have a wide range of applicability to different cultures and communities, vocabulary terms *must* be translated into various languages [11]. The localization (and internationalization) process of vocabularies should be supported by the system.

Labeling versions (R11) Release versions of vocabularies should be labeled appropriately. This ensures that users that can be humans or machines have always the possibility to use specific version, not only the latest one.

III. GIT4VOC

In this section we present Git4Voc. On the one hand, we propose guidelines how Git can be used for collaborative vocabulary development project. On the other hand, we present how the requirements from section II can be technically implemented by Git *hooks*. Additionally, in terms of guidelines we analyzed best practices from collaborative software development and identified the following aspects as critical for the quality of the vocabulary: (1) management of generated information; (2) rights management; (3) branching and merging; (4) automate development and deployment tasks by hooks; (5) tool independence; (6) vocabulary organization structure; and (7) labeling of release versions. In the next subsections we show in detail how our approach responds to the above mentioned requirements.

A. Management of Generated Information

During the development process a bunch of information is generated by the contributors. The capability to manage this information within the entire project life-cycle is essential. In fact, value added services like GitHub, GitLab or BitBucket enrich Git functionality with powerful information management features. For instance, issues are a great way of tracking communications, reporting problems as well as bug fixes and announcing version releases. Communities like *schema.org* manage their discussions using GitHub. The above mentioned means support requirement **(R1)**. Based on this fact, we propose that activities gathered in Table II should be documented. If possible, the name of issues should correspond to the name of the activities.

Another important requirement in collaborative vocabulary development is the ability to view the history of the changes (called traceability in software engineering). This addresses the requirement **(R2)**. Using commands *git log* and *git diff* a user can explore the history of the commits and the differences between them. Each commit should be realized based on *Best Commit Practices*[13]. In vocabulary development the *atomicity* of commits is of paramount importance.

[13] http://www.git-tower.com/learn/git/ebook/command-line/appendix/best-practices

TABLE I: Different roles and their primary activities.

Roles	Basic Activities	Semantic Issues	Structural Issues
Vocabulary Eng.	+	+	+
Domain Expert	+	-	-
Users	-	-	-
Translators	+	-	-

B. Rights Management

Standalone solutions such as *GitLab*[14] and *Gitolite*[15] as well as third-party services like *Bitbucket*[16] and *GitHub*[17] offer basic options for user rights managements, like reading, writing, posting, adding new team members and adding tags. However, even with these solutions a high level of user management i.e. restricting editing a specified number or type of classes, properties or instances cannot be achieved with Git. In order to address requirement **(R3)**, we explore a combination of branching and hooks.

With the combination of branching and hooks with role definition for users, fine grained access management can be achieved. Concretely, by using server-side hooks, realizing rights managements on top of user roles is possible. For instance, an implementation of a *pre-push* hook can check for the user's role and permissions and deny if the necessary rights according the activity and branch are not set.

Table I shows common roles and their permissions, with respect to the defined categories of activities. In a trusted environment right management can also be realized with client-side hooks. An example for this is depicted in Listing 3, where the user is denied to push to the master branch.

C. Branching and Merging

Git is a very flexible tool, which addresses requirement **(R4)**. Using Git, teams are able to organize their work in different types of workflows[18]. Branching strategies affect the quality in collaborative software development [12], [13]. Vocabulary development is mostly accepted to be a specific type of software development. Therefore, it is considered that the branching strategy affects the quality of the vocabularies. Well-known projects such as *schema.org* use branches to organize their work. In order to design a branching model, it is important to understand the possible activities that a team can perform. In this regard, we collected common activities of collaborative vocabulary development which are listed in Table II. Aiming at producing a vocabulary with good quality, the entire team should be aware of these activities and how to face them in the development process. Due to their impact

[14] https://gitlab.com/gitlab-org/gitlab-ce/blob/master/doc/permissions/permissions.md
[15] https://github.com/sitaramc/gitolite
[16] https://confluence.atlassian.com/display/BITBUCKET/Add+Users,+Set+Permissions,+and+Review+Account+Plans
[17] https://help.github.com/articles/permission-levels-for-an-organization-repository
[18] https://www.atlassian.com/git/tutorials/comparing-workflows/forking-workflow

TABLE II: Common Activities in Collaborative Vocabulary Engineering

| Activity	Name	Description	Example
ACT1	Simple Addition/Deletion	Adding new or deleting existing elements like classes and properties	Adding a class in the last level of the taxonomy
ACT2	Complex Addition/Deletion	Adding new elements to be interconnected within the existing class or properties taxonomy	Adding a object property as a super property of two existing properties
ACT3	Modification	Modifying existing elements	Modifying the domain and range of an existing object property
ACT4	Reusing	Reusing elements of the Linked Data Cloud	Defining new local concepts by using external resources
ACT5	Alignment	Alignment of existing elements with equivalents in the cloud	Alignment of classes and instances with *owl:equivalentClass* and *owl:sameAs*
ACT6	Refactoring	Changing the name and metadata of an specific element and its connections	Renaming a class which is connected in many domain and range relation of properties and need to be renamed everywhere
ACT7	Common Metadata	Adding/Removing/Modifying predefine *RDFS* metadata	Adding metadata to a class with *rdfs:label, rdfs:comment*
ACT8	External Metadata	Adding/Removing/Modifying external metadata	Adding metadata to a class with *skos:prefLabel, dc:title*
ACT9	Translating	Adding/Removing/Modifying translation for the terms	Using *rdfs:label* to translate elements into different languages
ACT10	Modularization	Adding new modules to the existing vocabulary	Creating and integrating new modules due to new requirements
ACT11	Partitioning	Partitioning into different modules with existing elements	The vocabulary has grown in size and semantic complexity. Partitioning the existing vocabulary into different modules
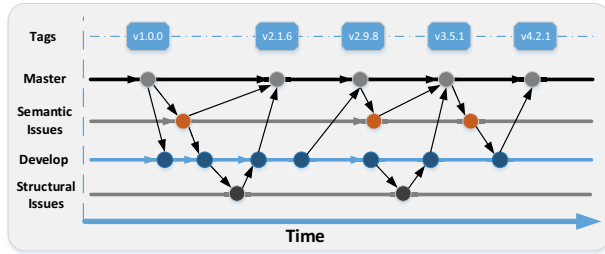
Fig. 1: Branching model for vocabulary development

on the overall vocabulary, we have classified these activities into three categories: (1) basic activities (*ACT1, ACT7, ACT9*), (2) semantic issues (*ACT2, ACT3, ACT4, ACT5, ACT6, ACT8*) and (3) structural issues (*ACT10, ACT11*).

This led us to the branching model that is depicted in Figure 1. We designed different branches to handle the mentioned categories. Basic activities have to be performed in the *Develop Branch*. For the second category we propose a dedicated branch called *Semantic Issues*. In case of the third category a branch named *Structural Issues* has to be applied. It is important to bear in mind that we are not restricting the flexibility of Git regarding branches. On the contrary, other branches can be used as a complement of this model. Nevertheless, our approach of branching model will help developers because those branches are connected to specific activities in collaborative vocabulary development.

Our solution is built on top of the best practices for branching in software development[19].

D. Automate Development and Deployment Tasks by Hooks

Despite the fact that Git has many implemented features, it allows extending its functionality by using so-called hooks. This is a mechanism that allows running scripts before or after specific Git events. Based on the execution place, two types

of hooks are distinguished, client-side and server-side hooks. Due to space limitations, the examples in this article showcase only client-side hooks. In order to address requirements (**R5**) and (**R6**), we implemented the following three important tasks for collaborative vocabulary development: (1) syntax checking, (2) assessing vocabularies against best design practices and (3) documentation generation. Figure 2 illustrates how these examples are integrated into the *commit* process.

After modifying the local vocabulary and adding changes to the stage phase, the next step is to *commit* the current state to the local repository. The initialization of commit triggers a hook named *pre-commit*. Listing 3 shows our implementation of this hook which realizes the tasks syntax checking and best practice assessment. First, it retrieves all modified files with extensions such as *rdf*, *owl*, *ttl* and checks for syntax errors by using *Rapper*[20]. In case that vocabularies fail to pass the validation process, the commit is canceled. The user is notified with a message which shows detailed description about the error which comprised of the file name, line number and the error type. If syntax validation is passed successfully, the modified files are posted to the *OOPS*[21] Web Service through *curl*, a command-line HTTP client. This service assesses vocabulary files for certain quality metrics. The result of this is a descriptive message that contains recommendations of best practices for vocabulary development. If no errors exist, the pre-commit hook is finished and the commit is accepted. Afterwards a *post-commit hook* is called. Listing 4 demonstrates our implementation of a post-commit hook for documentation generation in a human friendly format. This script uses *Parrot*[22] as an external tool.

For security reasons Git repository services do not allow to automatically distribute predefined *hooks* on cloning phase. In order to accomplish this task, the repository itself should have a dedicated folder that contains the implemented hooks. After the first clone, these hooks need to be copied to the *.git/hooks* directory. For that purpose, we implemented a script
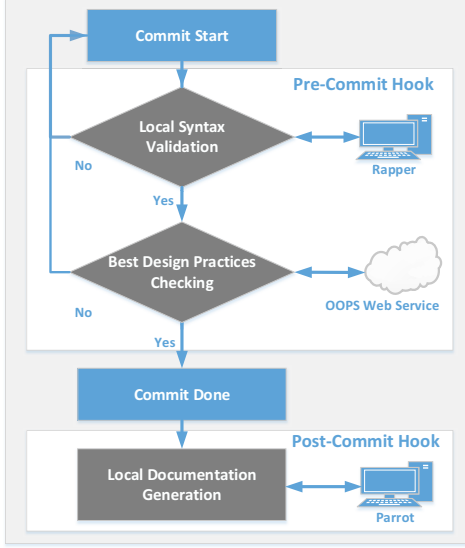
---

[19]http://nvie.com/posts/a-successful-git-branching-model

[20]http://librdf.org/raptor/

[21]http://oops-ws.oeg-upm.net/

[22]https://bitbucket.org/fundacionctic/parrot/

Fig. 2: Client-side *Hooks* Workflow

the merge tool is necessary, which is a time consuming and error prone task that could lead to information lose.

In order to avoid the above mentioned problems, we propose the use of Turtle format. This addresses the requirement (**R8**). A similar approach describes a pattern to express data on GitHub storing it in CSV files[26]. Listing 1 presents our proposal to write one triple per line.

Listing 1: One triple per line

```
@prefix rdf:  <http://www.w3.org/1999/02/22−rdf−syntax−ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf−schema#>.
@prefix owl:  <http://www.w3.org/2002/07/owl#>.
@prefix scor: <http://purl.org/eis/vocab/scor#>.
@prefix vs:   <http://www.w3.org/2003/06/sw−vocab−status/ns#>.

scor:Process
   rdf:type owl:Class ;
   rdfs:comment "A process is a unique activity..."@en ;
   rdfs:label "Process"@en ;
   rdfs:isDefinedBy scor: ;
   vs:term_status "testing".

scor:Enable
   rdfs:subClassOf    scor:Process ;
   rdfs:comment            "Enable describes the ...";
   rdfs:label              "Enable".
```

F. Vocabulary Organization Structure

Git's basic functionalities do not support modularizing code or vocabularies. Therefore, in order to address the requirement (**R9**), we propose some guidelines for organizing the vocabulary in files where each file represents a module. Considering the fact that each line should represent a triple and based on the insights on [10], we propose that files should not contain more than 300 triples. We highlight three possible forms of organizing the files. All of these cases use single Git repository to store the files.

1. The complete vocabulary is contained in one single file. When the vocabulary is small (e.g. contains less than 300 lines of code) and represents a domain which cannot be divided in sub domains, it should be saved within one single file. If the number of contributors is relatively small and the domain of the vocabulary is very focused, organizing it into one single file might be possible, even if it exceeds 300 lines of code. However, if the comprehensibility is exacerbated, splitting it into different files should be considered.

2. The vocabulary is split in multiple files. If the vocabulary contains more than 300 lines of code or covers a complex domain, it should be organized into different sub domains or modules. In this regard, we mapped sub domains with modules. When the sub domains themselves are small enough they should be represented by different files within the parent folder. There exists patterns for vocabulary modularization [15]. We developed the MobiVoc based on the pattern *n modules importing 1 module*. In this case, *1 module* was the vocabulary itself. The *n modules* like Aircraft, Fuel were saved in separate files. Each file represents a specific sub domain. By following this approach, domain experts can contribute independently to vocabulary development according to their

which needs to be executed after cloning the repository. Once this process is finished, predefined *hooks* will be automatically executed after each commit. However, when the hooks have been changed, e.g. to use different validation or documentation generation tools, this script has to be executed again. Apart from installing the hooks, this script can also be used to download and install tools like Rapper, which are necessary for the hooks. If these tools are to be placed within the local repository, the file *.gitignore* should be used to prevent them from being pushed to the remote repository.

Git does not show semantic diffs between versions of vocabulary. *Owl2VCS* [14] shows deltas among different versions. By using such a tool and hooks, generated deltas can be published is human friendly format as well. This corresponds to the requirement (**R7**).

E. Tool Independence

Collaborative working with Git can be facilitated by using vocabulary editors like *Protégé*[23], *TopBraid Composer*[24], *Neon Toolkit*[25]. As each of them has different algorithms for writing files, there might arise consistency problems in case that contributors are not using the same editor. For instance, one contributor use Protégé, whereas another one uses Neon Toolkit. They are editing the same file simultaneously. After saving it, different representations of that file will be created. As a consequence Git recognizes lot of changes and asks for conflict resolution. This is due to fact that Git is a version control based on text line changing. It detects when a line has been changed from the previous version. In such a case using

---

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

[24]http://www.topquadrant.com/tools/IDE-topbraid-composer-maestro-edition/

[25]http://neon-toolkit.org/wiki/Main_Page.html

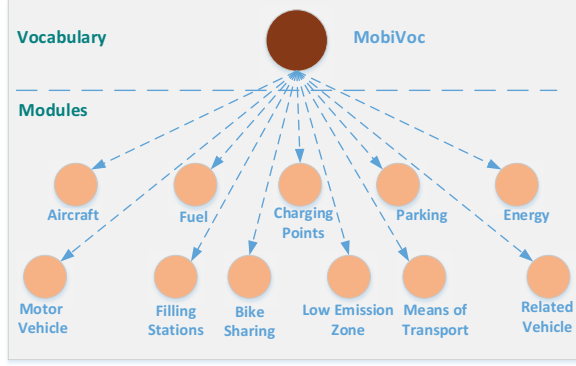[26]http://blog.okfn.org/2013/07/02/git-and-github-for-data/

Fig. 3: The structure of MobiVoc

field of expertise. Figure 3 depicts the structure of MobiVoc and its modules.

3. Vocabulary modules are stored in files and folders. For huge vocabularies that comprises complex domains, splitting it into files is not sufficient. This would lead to a large amount of files within a single folder. Therefore, if the sub domains are large enough to be split into files they should be represented by folders. Each folder contains files which represents modules. In this case, the folder and file structure should reflect the complex hierarchy of the overall domain.

Through splitting the vocabulary in files for specific purposes, the requirement **(R10)** is addressed as well. This can be achieved by creating dedicated files for translating. In these files users with the role *Translators* can contribute by translating the terms into the required language.

G. Labeling of Release Versions

Based on requirement **(R11)**, proper labeling of release versions is vital, as it facilitates re-usability. One of the common ways to realize that is to deploy each release version in different files. However, this could lead to following problems as identified in [16]: (1) the number of files could increase rapidly, (2) choosing versions creates confusion, (3) maintenance needs additional resources and (4) synchronizing with latest version from dependent applications requires additional effort. To avoid the above mentioned problems, we have kept versions of vocabularies in the same file. These versions are separated by Git implemented functionality of tagging and saved in the master branch which is part of the branching model and illustrated in Figure 1. It is possible to create and filter tags at any time. Moreover, users can obtain a specific version of the vocabulary just by giving the tag name. Therefore, each released version of a vocabulary *must* have a version number. Based on the scheme from [17] and the mentioned categories of activities in Table II, we propose tagging different versions according to the following pattern: *v[StI.SeI.BA]*, where *StI* stands for Structural Issues, *SeI* for Semantic Issues and *BaA* for Basic Activities. Each category is related with a number, in the respective position. Changes in the vocabulary regarding to the categories are commonly reflected by increasing the numbers. For instance, the difference between releases *v[1.0.0]* and *v[2.0.0]* shows structural issue changes (*StI*).

IV. IMPLEMENTATION

We have developed Git4Voc[27], which is an environment for collaborative vocabulary development. Table III provides an overview which of the previously described requirements are fulfilled by Git4Voc. This solution combines Git4Voc with a set of state-of-the-art tools like Rapper, OOPS Service and Parrot. Each tool is exchangeable and can be easily replaced by alternatives. They provide services which are called by the hooks mechanism. In the following these hooks are presented in detail.

The Listing 2 shows an example how predefined hooks are copied into the *.git/hooks* folder after cloning the repository. In addition, it shows installing of the tools: Raptor and Parrot and their necessary libraries in case they do not exist.

Listing 2: Install Hooks and Tools

```sh
#!/bin/sh
# Copy the modified hooks
for i in `ls -1 hooks` do
    cp hooks/$i .git/hooks/$i
done
# Create directory for necessary tools
if [ -d "$tools" ]; then
  mkdir -p "tools"
  ...
fi
cd tools

#Install Raptor
if [ ! -d "$Raptor" ]; then
  curl -O http://download.librdf.org/source/raptor2-2.0.15.
      tar.gz
  ...
  sudo apt-get install libxml2-dev libxslt1-dev python-dev
  sudo apt-get -y install raptor2-utils
  ...
fi

#Install Parrot
if [ ! -e "parrot-jar-with-dependencies.jar" ]; then
  curl -O https://github.com/vocol/vocol/raw/master/Hooks/
      tools/parrot-jar-with-dependencies.jar
fi

...
```

The *pre-commit hook* is adapted to prevent users from committing to the master branch as shown in the Listing 3. This example can be further customized to restrict committing to other branches as well. By doing so, a low level of rights management is achieved on the local repository, before the changes are pushed to the remote repository. Furthermore, to reduce the efforts needed for subsequent corrections, we integrated tools for (1) syntax validation; and (2) checking for bad modeling practices. For the first, the Rapper tool is used, which validates each turtle file for syntactic errors. For the second, we used OOPS Web Service to scan vocabulary files for bad modeling practices.

---

[27]https://github.com/vocol/vocol/tree/master/Git4Voc

TABLE III: Collaborative Requirements from Git4Voc perspective.

No.	Requirement	Supported by plain Git	Means
R1	Communication support	+	Issues tracking system offered by hosting platforms GitHub, GitLab, BitBucket
R2	Provenance of information	+	Git log and git diff functionality
R3	Different roles	+	Further extension by a combination of branching and hooks
R4	Workflow independence	+	Git, branching and merging strategies
R5	Quality assurance	-	Extended by a combination of hooks and tools like: Rapper, JenaRiot
R6	Documentation generation	-	Extended by a combination of hooks, documentation generation tools like: SchemaOrg, Widoco
R7	Deltas among releases	-	Extended by a combination of Owl2VCS as an external tool and hooks
R8	Editor agnostic	+	Git base functionality
R9	Modularity	+	Hierarchy organization of vocabularies where modules are represented by files
R10	Multilinguality	+	Hierarchy organization of vocabularies where dedicated files are used for multilinguality
R11	Labeling versions	+	Git tag functionality

Listing 3: Pre-Commit Hook: Syntax validation and checking for bad modeling practices

```bash
#!/bin/bash
currentBranch=$(git symbolic-ref HEAD)

if [ "$currentBranch" = "refs/heads/master" ]; then
    echo "Not allowed to commit to master branch!"
    exit 1
else
    # Get only modified files with ttl extension
    files=$(git diff --cached --name-only --diff-filter=ACM
        | grep ".ttl$")
    ...

    for file in ${files}; do
        # Validate each file using Rapper
        res=$(rapper -i turtle "${files}" -c 2>&1)
    ...

    if ! $error; then
        for file in ${files}; do
          fileContent=`cat ${files}`
          request="<?xml ...>"
          res=$(curl -X POST -d "$request" -H "Content-
              Type: application/xml" http://oops-ws.oeg-
              upm.net/rest)
          ...
      done
    fi

    if ${succeed}; then
        echo "COMMIT SUCCEEDED"
    else
        echo "COMMIT FAILED"
        exit 1
    fi
fi
```

The Listing 4 shows the *post-commit hook*, which uses Parrot to create the human friendly representation of the developed vocabulary. The generated content is saved as a single HTML file which consists of all vocabulary elements. User is able to navigate through entire vocabulary by merely selecting the element name. Moreover, in order to create different representation style of the vocabulary, alternative tools such as: *Widoco*[28], *Specgen*[29], *Dowl*[30], etc. (c.f. commented part of the code) can be used as well.

Listing 4: Post-Commit Hook: Documentation Generation

```sh
#!/bin/sh
files=$(git diff --cached --name-only --diff-filter=ACM |
    grep ".ttl$")
```

[28] https://github.com/dgarijo/Widoco
[29] https://github.com/specgen/specgen
[30] https://github.com/ldodds/dowl

```sh
if [ "$files" = "" ]; then
    exit 0
fi

for file in ${files}; do
    # Generate documentation using Parrot
    java -jar @path/parrot-jar-with-dependencies.jar -i "${
        file}" -o "${file}".html
    # Generate documentation using Widoco
    # java -jar  @path/widoco-0.0.1-jar-with-dependencies.
        jar -ontFile "${file}" -outFolder /home/
    ...
done
echo "\Documentation Generation is completed.\n"
exit 1
```

V. RELATED WORK

Collaborative vocabulary development is an active research area in the Semantic Web community [19]. Existing approaches like *WebProtégé* [20] provides a collaborative web frontend for a subset of the functionality of the Protégé OWL editor. The aim of WebProtégé, is to lower the threshold for collaborative ontology development. *Neologism* [21] is a vocabulary publishing platform, with a focus on ease of use and compatibility with Linked Data principles. Neologism focuses more on vocabulary publishing and less on collaboration. VocBench [22], is an open source web application for editing thesauri complying with the SKOS and SKOS-XL standards. VocBench has a focus on collaboration, supported by workflow management for content validation and publication.

The main limitation of the aforementioned tools is the lack of version control. Therefore, we only consider approaches focused on using version control systems for collaborative vocabulary development.

SVoNt [23] extends the functionality of Apache SubVersion (SVN) by providing a possibility for versioning OWL conform lightweight description logic. SVN manipulates only with deltas of files, therefore SVoNt use a separate server to create conceptual changes between versions of ontologies. These changes are generated as a result of *diff* operation between the modified ontology and the base ontology. *ContentCVS* [24] is a Protégé plugin. It adapts concepts from *concurrent versioning* to enable developers to work in parallel. Moreover, it has features for conflict detection and resolution by checking structure and semantic of the ontology versions. In [17] is described

how the developers of *RDA Vocabularies*[31] adopt rules from *SemVer*[32] to realize a meaningful versioning using Git. Additionally, it provides general notes for organizing the vocabulary development in branches. [25] describes Owl2VCS, a toolset designed to facilitate version control of OWL 2 ontologies using version control systems. It can be integrated as an external tool with Git, Mercurial and Subversion and provide algorithms for structural diff [14]. However, none of the above mentioned approaches cover all the identified requirements (c.f. section II) for collaborative vocabulary development. On the contrary, our work analyze and address each one of them by using Git and Git4Voc as an extension.

VI. CONCLUSION AND FUTURE WORK

In this paper, we investigated the applicability of Git for collaborative vocabulary development. We defined collaborative vocabulary development as the process of identifying the main terms across heterogeneous data sources by finding a consensus between the developers. The main challenge in this regard is the realization of a powerful collaborative environment. Distributed version control systems enable developers around the world to work collaboratively on complex software systems. Since software and vocabularies are not the same, we analyzed their differences in detail by identifying requirements for a version control system that supports collaborative vocabulary development. Our approach extends plain Git functionality by utilizing the hooks mechanism in combination with external tools to address these requirements. The presented approach is easily extensible and can accommodate additional external tools.

Regarding the future work, we are going to extend our approach with the full implementation of *server side hooks*. By doing so, tasks like: deploying specific versions of vocabularies to a dedicated server, generating deferencable URI's, ontology partitioning and modularization tasks can be performed in a fully automated way. We also plan to develop and integrate a tool that validates vocabularies against conventions [1] and provides recommendations for solving possible issues. This will lead to a convenience and less error prone collaborative vocabulary development environment. As a result, all generated artefacts will be publicly accessible from all interested parts.

REFERENCES

[1] I. Grangel-González, L. Halilaj, G. Coskun and S. Auer, "Towards vocabulary development by convention," in *7th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management, Vol 2*, 2015, pp. 334–343.

[2] E. Kalliamvakou, D. Damian, K. Blincoe, L. Singer, and D. German, "Open source-style collaborative development practices in commercial projects using github," in *37th International Conference on Software Engineering (ICSE15)*, 2015.

[3] N. F. Noy, A. Chugh, W. Liu, and M. A. Musen, "A framework for ontology evolution in collaborative environments," in *ISWC*. Springer, 2006, pp. 544–558.

[4] N. F. Noy and T. Tudorache, "Collaborative ontology development on the (semantic) web." in *AAAI Spring Symposium: Symbiotic Relationships between Semantic Web and Knowledge Engineering*, 2008, pp. 63–68.

[5] J. Seidenberg and A. L. Rector, "The state of multi-user ontology engineering." in *WoMO*. Citeseer, 2007.

[6] T. Tudorache, N. F. Noy, S. M. Falconer, and M. A. Musen, "A knowledge base driven user interface for collaborative ontology development," in *16th International Conference on Intelligent User Interfaces*. ACM, 2011, pp. 411–414.

[7] E. Simperl and M. Luczak-Rösch, "Collaborative ontology engineering: a survey," *The Knowledge Engineering Review*, vol. 29, no. 01, pp. 101–131, 2014.

[8] M. C. Suarez-Figueroa, A. Gómez-Pérez, and M. Fernandez-Lopez, "The neon methodology for ontology engineering," in *Ontology engineering in a networked world*. Springer, 2012, pp. 9–34.

[9] M. d'Aquin, A. Schlicht, H. Stuckenschmidt, and M. Sabou, "Ontology modularization for knowledge selection: Experiments and evaluations," in *Database and Expert Systems Applications*. Springer, 2007, pp. 874–883.

[10] A. Schlicht and H. Stuckenschmidt, "Towards structural criteria for ontology modularization," in *ISWC 2006 Workshop on Modular Ontologies*. Citeseer, 2006.

[11] J. Gracia, E. Montiel-Ponsoda, P. Cimiano, A. Gómez-Pérez, P. Buitelaar, and J. McCrae, "Challenges for the multilingual web of data," *Journal of Web Semantics*, vol. 11, pp. 63–71, 2012.

[12] S. Phillips, J. Sillito, and R. Walker, "Branching and merging: an investigation into current version control practices," in *4th International Workshop on Cooperative and Human Aspects of Software Engineering*. ACM, 2011, pp. 9–15.

[13] E. Shihab, C. Bird, and T. Zimmermann, "The effect of branching strategies on software quality," in *ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2012, pp. 301–310.

[14] R. S. Gonçalves, B. Parsia, and U. Sattler, "Ecco: A hybrid diff tool for owl 2 ontologies." in *OWLED*, 2012.

[15] S. B. Abbes, A. Scheuermann, T. Meilender, and M. d'Aquin, "Characterizing modular ontologies," in *7th International Conference on Formal Ontologies in Information Systems (FOIS)*, 2012, pp. 13–25.

[16] P. Bedi and S. Marwaha, "Versioning owl ontologies using temporal tags," *International Journal of Computer, Control, Quantum and Information Engineering*, 2007.

[17] J. P. Diane I. Hillmann, Gordon Dunsire, "Versioning vocabularies in a linked data world," *IFLA Lion*, 2014.

[18] *VoCol: An Agile Methodology and Environment for Collaborative Vocabulary Development*. Zenodo, Feb. 2015. [Online]. Available: http://dx.doi.org/10.5281/zenodo.15023

[19] R. Palma, O. Corcho, A. Gómez-Pérez, and P. Haase, "A holistic approach to collaborative ontology development based on change management," *Journal of Web Semantics*, vol. 9, no. 3, pp. 299–314, 2011.

[20] T. Tudorache, C. Nyulas, N. F. Noy, and M. A. Musen, "Webprotégé: A collaborative ontology editor and knowledge acquisition tool for the web," *Semantic Web*, vol. 4, no. 1, 2013.

[21] C. Basca, S. Corlosquet, R. Cyganiak, S. Fernández, and T. Schandl, "Neologism: Easy vocabulary publishing," 2008.

[22] A. Stellato, S. Rajbhandari, A. Turbati, M. Fiorelli, C. Caracciolo, T. Lorenzetti, J. Keizer, and M. T. Pazienza, "Vocbench: A web application for collaborative development of multilingual thesauri," in *The Semantic Web. Latest Advances and New Domains*. Springer, 2015, pp. 38–53.

[23] M. Luczak-Rösch, G. Coskun, A. Paschke, M. Rothe, and R. Tolksdorf, "Svont-version control of owl ontologies on the concept level." *GI Jahrestagung (2)*, vol. 176, pp. 79–84, 2010.

[24] E. Jiménez-Ruiz, B. C. Grau, I. Horrocks, and R. B. Llavori, "Contentcvs: A cvs-based collaborative ontology engineering tool." in *SWAT4LS*. Citeseer, 2009.

[25] I. Zaikin and A. Tuzovsky, "Owl2vcs: Tools for distributed ontology development." in *OWLED*. Citeseer, 2013.

---

[31] http://www.rdaregistry.info

[32] http://www.semver.org