

Specification and Validation of Quality Criteria for Git Repositories using RDF and SHACL

Leon Martin¹, Andreas Henrich¹

¹University of Bamberg, An der Weberei 5, 96047 Bamberg, Germany

Abstract

As part of quality management, it is important to ensure that Git repositories meet certain quality criteria depending on the type of the respective project. In academia, for instance, repositories of finished research projects should always comprise a readme file providing usage instructions. The range of heterogeneous resources within modern Git repositories leads to a large number of conceivable quality criteria, thus increasing the effort for validating Git repositories against the project type specific quality criteria. At the same time, there is a lack of approaches and tools supporting this validation. Hence, this paper proposes and discusses an approach based on RDF and SHACL for validating a given Git repository against a set of quality criteria defined for its intended project type. QUARE, a working prototype implementing the approach, is provided and discussed.

Keywords

Applications of knowledge management, Representation of Git repositories, Git repository quality

1. Introduction

Today, the version control software Git¹ and related platforms like GitHub² are standard tools for developing and sharing software. Version control systems store data within so called repositories in the form of a filesystem tree [1]. Developers can connect to the repository and edit the files in the tree as needed. Every committed change is registered and stored such that older revisions of the files can be viewed retrospectively. On top of this core functionality, platforms like GitHub provide a continuously growing number of powerful features like issue tracking, release management, wikis, CI/CD [2], and many more. As a result, modern Git repositories comprise many heterogeneous resources beyond pure software. Further increasing the complexity, different relationships exist between the resources and the users of a repository: There are one-to-one relationships, e.g., one repository can have exactly one description, one-to-many relationships, e.g., one repository can comprise one or more branches, and many-to-many relationships, e.g., one or more issues can be assigned to one or more contributors.

In software engineering, there are numerous tools and practices for producing high-quality software [3]. Similarly, Git repositories should also comply with certain quality criteria depending on the intended type of the respective project. Due to the complexity of modern


LWDA'22: *Lernen, Wissen, Daten, Analysen*. October 05–07, 2022, Hildesheim, Germany

✉ leon.martin@uni-bamberg.de (L. Martin); andreas.henrich@uni-bamberg.de (A. Henrich)

🆔 0000-0002-6747-5524 (L. Martin); 0000-0002-5074-3254 (A. Henrich)



© 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

¹<https://git-scm.com> (visited 2022/09/07)

²<https://github.com> (visited 2022/09/07)

Git repositories, there is a range of conceivable quality criteria. In fact, all resources of a Git repository can theoretically be the subject of some quality criteria, thus making it difficult to evaluate the quality of one or even multiple Git repositories that have to follow certain criteria across an organization consistently. At the same time, there is a lack of tools supporting the validation of Git repositories against such quality criteria, thus motivating the present paper.

In this paper, an approach based on the Resource Description Framework (RDF) [4] and the Shapes Constraint Language (SHACL) [5] is proposed for validating a Git repository against a set of quality criteria defined for its intended project type. The approach follows a three-step process to check whether a Git repository meets the quality criteria of its intended project type:

1. A representation of the project type and the corresponding quality criteria is created.
2. A representation of the repository and its heterogeneous resources is created with respect to its intended project type.
3. An engine is leveraged to validate the repository representation against the representation of the quality criteria of its intended project type.

We provide a prototypical tool called QUARE³ implementing the approach. The tool allows users to easily check whether Git repositories of interest comply with the quality criteria they should fulfill according to their intended project type. Even though this paper focuses on repositories of academic projects, the approach and thereby the tool can readily be applied to other kinds of Git repositories.

The remainder of this paper is organized as follows: Section 2 investigates important foundations and related work. In Section 3, the approach is explained thoroughly. Then, Section 4 focuses on the implementation of the approach by introducing our working prototype. Subsequently, the approach and the implementation are discussed in Section 5, before drawing a conclusion in Section 6.

2. Foundations & Related Work

In the beginning, this section introduces a set of useful quality criteria with respect to literature and our academic background, which is the first contribution of this paper. Afterwards, the candidate technologies for implementing the proposed approach are investigated.

2.1. Quality Criteria for Git Repositories

Although Git and related platforms like GitHub are standard tools in software development today, there is few scientific literature explicitly discussing quality criteria for Git repositories. Instead, the available literature mainly discusses topics around so called data repositories for storing datasets. Examples of such data repositories provide interactivity and support collaboratively working on the data [6, 7]. Regarding quality criteria, there is, for instance, a paper concerned with the consistency of data within such repositories [8]. Typical areas of application for data repositories are the general scientific domain [6] and its various disciplines like medicine [7].

³<https://github.com/uniba-mi/quare> (visited 2022/09/07); pronounced like *quare*, the Latin word for *why*, which is a reference to the tool's ability to provide explanations why a repository's validation did not succeed.

Actually, one could understand a modern Git repository as a highly collaborative and feature-rich form of a data repository mainly for the software development domain. This is supported by the fact that papers like [9] mention code as a part of the data stored in research data repositories. Regarding quality indicators for code, the paper mainly elaborates on code reproducibility. To achieve this, lists of required dependencies, documentation, user guides, and in the best case virtual runtime environments provided via tools like Docker⁴ or platforms like Code Ocean⁵ are recommended.

Apart from the scientific literature, there are posts on blogs and Q&A sites addressing quality indicators of Git repositories. For instance, answers to questions on StackOverflow⁶ and Quora⁷ state that comprehensive documentation, ease of installation, up-to-dateness, the activity on the repository, the friendliness of the community, the extensibility of the provided code, and the usage of software testing are important indicators.

In the end, the repository maintainers decide which quality criteria they want to enforce, thus calling for a flexible approach. Based on our academic expertise and the literature, we compiled a first set of generic quality criteria categories that can be useful for academic Git repositories (and others). Table 1 gives an overview of the considered quality criteria categories on the *y*-axis and the affected Git repository resources on the *x*-axis. As shown, we distinguish five categories⁸: *Existence* criteria check if a resource is available at all, *quantity* criteria check if the correct number of a resource is available, *naming* criteria check if a resource is named appropriately, *content* criteria check if a resource comprises the correct contents, and *status* criteria check a resource’s status.

Table 1

An overview of the five considered quality criteria categories (*y*-axis) and some of the affected Git repository resources (*x*-axis). The × symbols indicate if there are quality criteria from the respective categories that are applicable for the respective resources.

	<i>Repository</i>	<i>Topics</i>	<i>Description</i>	<i>Branches</i>	<i>Issues</i>	<i>Releases</i>	<i>License</i>	<i>Readme file</i>
Existence		×	×		×	×	×	×
Quantity				×	×			
Naming	×			×	×			
Content			×				×	×
Status	×			×	×			

⁴<https://docker.com> (visited 2022/09/07)

⁵<https://codeocean.com> (visited 2022/09/07)

⁶See <https://stackoverflow.com/questions/22527438> (visited 2022/09/07)

⁷See <https://www.quora.com/How-do-you-measure-a-good-github-repository> (visited 2022/09/07)

⁸There are numerous other conceivable quality criteria and criteria categories which, however, cannot be covered here comprehensively. For instance, one could impose an exotic criterion addressing the quality of writing of some file’s content (cf. [10]) or require criteria with respect to the FAIR data principles (<https://www.go-fair.org>; visited 2022/09/07). The representations of constraints and Git repositories used by our approach allow adding other criteria categories and specific criteria in the future.

We define that a repository R is of high quality iff $\forall q \in Q_T(\text{complies}(R, q))$, where Q_T is the set of all quality criteria of the intended project type T and complies is a predicate that evaluates to true if R complies with q .

2.2. RDF, SHACL & OWL

One strength of RDF [4] is its ability to semantically link heterogeneous data. For this purpose, RDF encodes knowledge as subject-predicate-object-triples, i.e., statements that specify binary relations between entities that are either Internationalized Resource Identifiers (IRIs), literal values, or blank nodes. The predicates defining the binary relations are also referred to as properties. Collections of such triples span so called RDF graphs, which can be queried using the SPARQL Protocol And RDF Query Language (SPARQL). On top of this, the SPARQL Inferencing Notation (SPIN) [11] allows specifying business rules in SPARQL against which a given RDF graph can be directly validated. Said rules include various types of constraints on the nodes and properties in the graph. It's modern and more powerful successor, SHACL [5], uses so called shapes graphs to model constraints. To validate an RDF graph, SHACL validators like `pySHACL`⁹ check whether the graph of interest complies with the constraints specified in the shapes graph.

An alternative for specifying constraints in the form of restrictions on RDF graphs is to leverage the OWL 2 Web Ontology Language (OWL) [12] and class expressions. Building on top of RDF, OWL greatly increases the expressiveness of RDF by introducing classes, properties, individuals, and data values such that semantic reasoners can be applied to perform reasoning on the data. One popular semantic reasoner is `PELLET` [13]. In this context, one typical inference problem is testing the satisfiability of class expressions, which are used to impose certain restrictions on the properties of classes [14]. Informally speaking, a class expression CE is not satisfiable if there exists an individual of the class defined using CE that does not comply with the restrictions specified by CE , for instance.

The following section describes how these RDF-related technologies can be employed to tackle the problem motivating this paper.

3. Concept

Modern platforms like GitHub typically provide comprehensive APIs and wrappers for accessing and manipulating repositories in code. Given such an API, the straightforward option for testing whether a Git repository complies with sets of quality criteria is to directly access the API or employ one of the available wrappers to check the individual quality criteria directly on the raw repository data. However, this approach has the drawback that an appropriate representation of the quality criteria and an engine for validating the repository against them has to be implemented single-handedly. For flexibility, it is thus more expedient to exploit already available technologies that suffice for the use case. In our specific case, RDF is well-suited for creating a generic and abstract intermediary representation of a Git repository because 1) platforms like GitHub already provide explicit Uniform Resource Locators (URLs), a subset

⁹<https://github.com/RDFLib/pySHACL> (visited 2022/09/07)

of IRIs, for most of the repositories’ resources and 2) the graph structures of RDF are able to naturally model the complex relationships between the heterogeneous repository resources and its users.

Regarding the actual validation, both OWL in combination with a semantic reasoner and SHACL in combination with a SHACL validator are capable of checking whether an RDF representation of a Git repository complies with certain quality criteria. In the former case the quality criteria would be modelled as class expressions, in the latter as a separate shapes graph. However, there is a decisive advantage of SHACL: Semantic reasoners for OWL like PELLET terminate as soon as one violation of a class expression is encountered. As a result, only the first violation can be reported to the user. Applied to our use case, users would only receive a report mentioning one violated quality criterion even if the Git repository of interest potentially comprises many more. In contrast, SHACL validators like pySHACL continue even if one or more constraints are violated and ultimately provide a full report of all violations. Therefore, SHACL was chosen for the approach that is mainly proposed and discussed in this paper. For completeness, our tool also features an OWL-based approach, which will be mentioned in Sections 4 and 5, too.

The remainder of this section uses the project type *finished research project* (T_F) as a running example to explain the SHACL approach. With respect to the currently supported criteria from Table 1, we define that repositories of finished research projects, have to comply with the quality criteria depicted in the upper section of Table 2. Note that the selected criteria reflect our understanding of a finished research project, i.e., a project whose development has finished but whose results, code etc. should still be usable by others. For comparison, the table shows another project type, namely *internal documentation* (T_I). Since repositories with this intended project type are for internal use, we impose less constraints for it.

The following sections will explain how the three-step process from Section 1 is implemented using the SHACL approach. Using this approach, the goal of the process is to evaluate the formula $\forall q \in Q_T(\text{complies}(R, q))$ from Section 2.1 by checking the compliance of the repository’s RDF representation with a shapes graph.

3.1. Creating a Representation of Project Types and Quality Criteria

As described previously, SHACL uses a separate shapes graph to specify the constraints for another RDF graph. In the context of this paper, the created shapes graph provides specifications for the supported project types. Every project type specification includes constraints on the properties that entities of the project type have to comply with as defined by $q_i \in Q_T$. To give an example, Figure 1 (a) shows the fragment of the shapes graph defining $q_5 \in Q_{T_F}$. The depicted properties with the leading sh prefix originate from the SHACL vocabulary [5]. More specifically, the sh:property property links a project type entity to so called property shapes which are structures for defining the constraints. The sh:path property defines the path in the RDF graph to which the constraint applies. In the example, the properties sh:minCount and sh:maxCount are leveraged to define the constraint that an entity with T_F as its intended project type has to possess exactly one instance of a property has_branch that links the entity to a branch entity¹⁰. Similarly, shapes for all $q_i \in Q_{T_F}$ are created. The exact design of each shape has

¹⁰Section 4.2 provides more examples regarding the implementation of quality criteria.

Table 2

The sets of quality criteria Q_{T_F}/Q_{T_I} that repositories of projects with the type *finished research project* (T_F)/*internal documentation* (T_I) have to comply with.

$q_i \in Q_{T_F}$	Quality Criterion	Category
q_1	The repository shall be public.	State
q_2	There shall be at least one topic assigned to the repository.	Quantity
q_3	There shall be a repository description.	Existence
q_4	The repository shall provide at least one software release.	Quantity
q_5	There shall be exactly one branch.	Quantity
q_6	The branch from q_5 shall further be called <i>main</i> .	Naming
q_7	There shall be no open issues.	Existence
q_8	There shall be a license.	Existence
q_9	The license from q_8 shall be a GNU GPL 3.0 license.	Content
q_{10}	There shall be a readme file.	Existence
q_{11}	The readme file from q_{10} shall include the sections <i>Installation</i> and <i>Usage</i> .	Content

$q_i \in Q_{T_I}$	Quality Criterion	Category
q_1	The repository shall be private.	State
q_2	There shall be a repository description.	Existence
q_3	There shall be a readme file.	Existence
q_4	The readme file from q_3 shall include the sections <i>Purpose</i> .	Content

to sufficiently capture the constraint but can otherwise be chosen according to taste because the structure of the RDF representation is not determined yet. Nevertheless, we recommend designing the shapes as concise as possible.

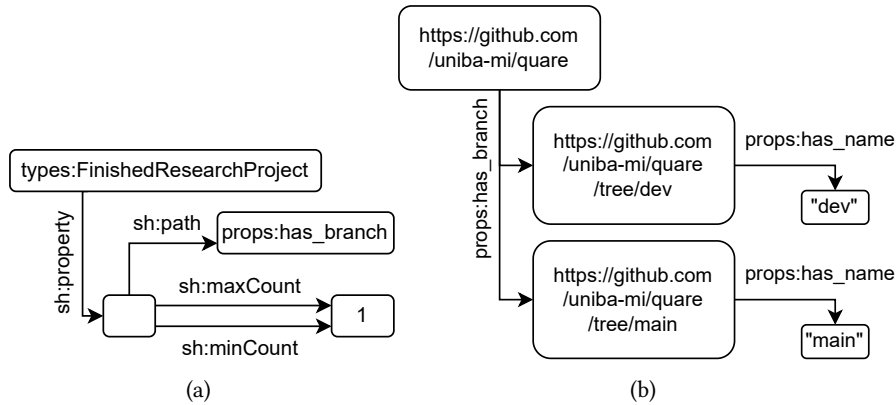


Figure 1: (a) The fragment of the shapes graph defining $q_5 \in Q_{T_F}$. (b) A fragment of the RDF representation of QUARE's GitHub repository.

3.2. Creating an RDF Representation of the Git Repository

In SHACL, the shapes graph is independent from the RDF graph that is validated against the shapes graph. To ensure compatibility between the shapes graph and the repository representation, the repository representation has to be created based on the same properties that are used for the constraints in the shapes graph and their structure. The naive option would be to just create the representation such that all q_i of all supported Q_T in the shapes graph could be validated based on them. This way, the time to create the representation becomes unnecessarily long and the representation itself unnecessarily big due to the diversity of possible quality criteria, though, without any benefit. To mitigate this problem, the RDF representations of repositories could be cached by exploiting RDF's serialization capabilities. However, caching comes with the risk that older versions of repositories are validated. This is problematic because the most recent versions might have new problems regarding the quality criteria while some problems might already have been fixed. For this reason, it is more expedient to choose another option which is to create a *minimal* representation just-in-time that only comprises the information that is necessary to check the constraints of the intended project type. Due to the diversity of possible quality criteria, one cannot rely on the API to directly provide exactly the information that is required to represent each $q_i \in Q_T$ with respect to the shape structures determined in the first step, though. Instead, the derivation of each triple from the available repository data has to be implemented manually as the algorithm in Algorithm 1 shows: After initialization, the wrapper is used to fetch the repository data that is relevant for each q_i . Based on the retrieved data, RDF triples are derived that sufficiently reflect the characteristics of the repository with respect to the SHACL constraints. The derived triples constitute the *minimal* RDF representation of the repository.

Algorithm 1 Generating a minimal RDF representation of a Git repository.

Require: The *url* of the repository, the set of quality criteria of the intended project type Q_T , and an empty set of triples G , i.e., the RDF representation of the repository.

wrapper \leftarrow *initApiWrapper(url)*

for $q_i \in Q_T$ **do**

data \leftarrow *wrapper.fetchRelevantData(q_i)*

triples \leftarrow *deriveTriples(data)*

 ▷ The derivation depends on the respective q_i .

$G \leftarrow G \cup \{triples\}$

end for

return G

To give an example, Figure 1 (b) shows a fragment of the RDF representation of QUARE's GitHub repository³. As depicted, two instances of the `props:has_branch` property are created that link the repository entity to entities representing the branches available in the repository, i.e., one branch entity with the name *dev* and one branch entity with the name *main*. The result of the second step is therefore an RDF representation of the Git repository that is adequate in the sense that it reflects the repository appropriately with respect to the properties used in the shapes graph such that it can be validated against the shapes graph in the upcoming third step.

3.3. Validation

Finally, a SHACL validator implementation, in our case `pySHACL`, is used to validate the Git repository's RDF representation from the second step against the shapes graph from the first step. If the RDF representation complies with the project type specific constraints defined in the shapes graph, the repository is of high quality as specified by the formula in Section 2.1. As depicted in Figure 1, `QUARE`'s repository features two branches while only exactly one branch is permitted by $q_5 \in Q_{T_P}$, thus causing `pySHACL` to report that this constraint is violated when the repository is validated against Q_{T_P} .

4. Implementation

Based on the SHACL (and the OWL) approach, the single-page application `QUARE`³ was implemented, which allows testing one or multiple GitHub repositories against the quality criteria of their intended project type via a web interface. The web interface is implemented using `Svelte`¹¹ and is connected to a backend written in Python which is executed on the same machine. For working with RDF and ontologies, the backend employs the `rdflib`¹² and `Owlready2`¹³ libraries as well as `pySHACL` and `PELLET`. For interaction with the GitHub API, the `PyGithub`¹⁴ wrapper is leveraged. In its current stage, the application comprises two pages, which are described in the following sections.

4.1. The Validation Page

The *Validation* page (cf. Figure 2) provides a form for entering an arbitrary number of GitHub repositories and their respective intended project types. By pressing the + and – buttons users can add or remove input fields as needed. Users can also provide their personal GitHub access token, which is required if they want to validate private repositories or make multiple validation requests in short succession. Via a switch, users can choose if the backend should use the SHACL or the OWL approach. When the form is submitted using the designated button, one individual request for each repository name and project type combination is sent to the backend triggering the execution of the chosen approach. If the validation in the backend succeeds, a green check mark is displayed. Otherwise, a red error symbol is shown accompanied by a button revealing the `pySHACL` or `PELLET` explanation when pressed. Since the raw explanations are not useful for users unfamiliar with SHACL and OWL, a basic template-based verbalization feature was implemented that extracts the parts of the explanation addressing the reasons why the validation failed and transforms them in natural language. The verbalized explanations are presented alongside the raw explanations. Furthermore, there is a button for saving the form inputs locally such that they do not have to be entered every time.

¹¹<https://svelte.dev> (visited 2022/09/07)

¹²<https://github.com/RDFLib/rdflib> (visited 2022/09/07)

¹³<https://github.com/pwin/owlready2> (visited 2022/09/07)

¹⁴<https://github.com/PyGithub/PyGithub> (visited 2022/09/07)

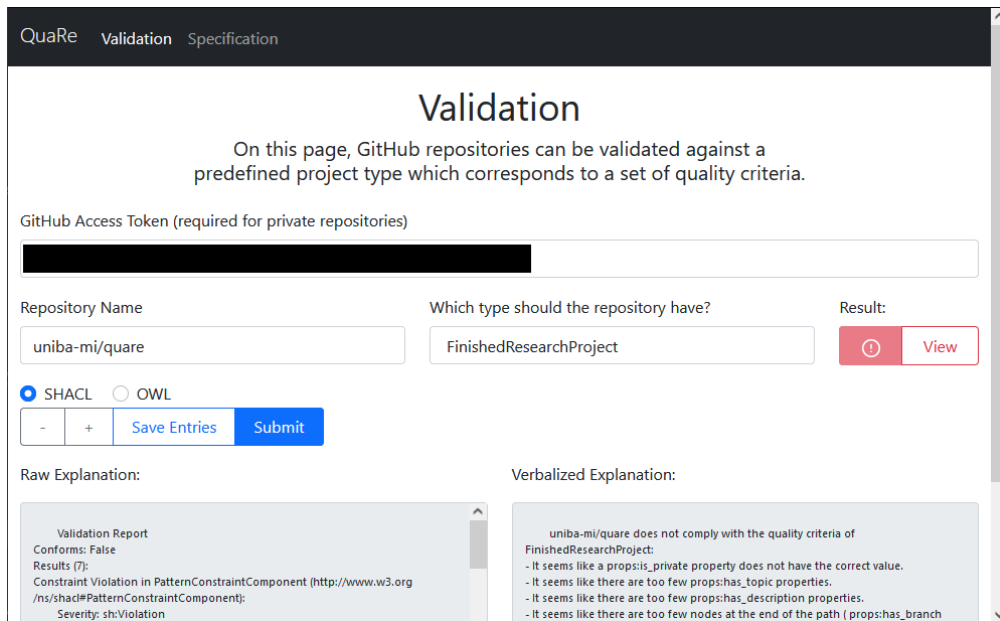


Figure 2: A screenshot of QUARE’s Validation page after a request to validate QUARE’s repository against T_F using the SHACL approach was issued. As shown, the validation has failed and the button revealing the raw and verbalized explanations has been pressed. Note that the screenshot was taken when the repository was still almost empty which is why basically all constraints of T_F were violated.

4.2. The Specification Page

The other page provided in the web interface is the *Specification* page (s. Figure 3) where the available project type specifications can be viewed. Users are able to choose if they want to view the SHACL specifications, as defined by the employed shapes graph, or the OWL specifications, as defined by the used class expressions. The screenshot shows that there are currently four supported projects types, namely *finished research project*, *internal documentation*, i.e., T_F and T_I from above, *ongoing research project*, and *teaching tool*. To give an example of a project type specification, the tab of the *teaching tool* project type is expanded revealing its quality criteria as defined by the SHACL shapes graph in the form of a list. Each list entry corresponds to another quality criterion. Using the `has_branch` example from Section 3, the fourth entry in the list defines that the RDF representation of a Git repository containing a *teaching tool* has to comprise at least two instances of the `has_branch` property. The final entry shows a more sophisticated constraint stating that there has to be a node that is 1) reachable via the `has_readme` and `has_section` properties and 2) has the literal value *Usage*. The combination of the `sh:qualifiedValueShape`, `sh:pattern`, and `sh:qualifiedMinCount` properties define a so called `sh:QualifiedMinCountConstraintComponent` [5]. In other words, the criterion states that there has to be a readme file with a *Usage* section in the repository.

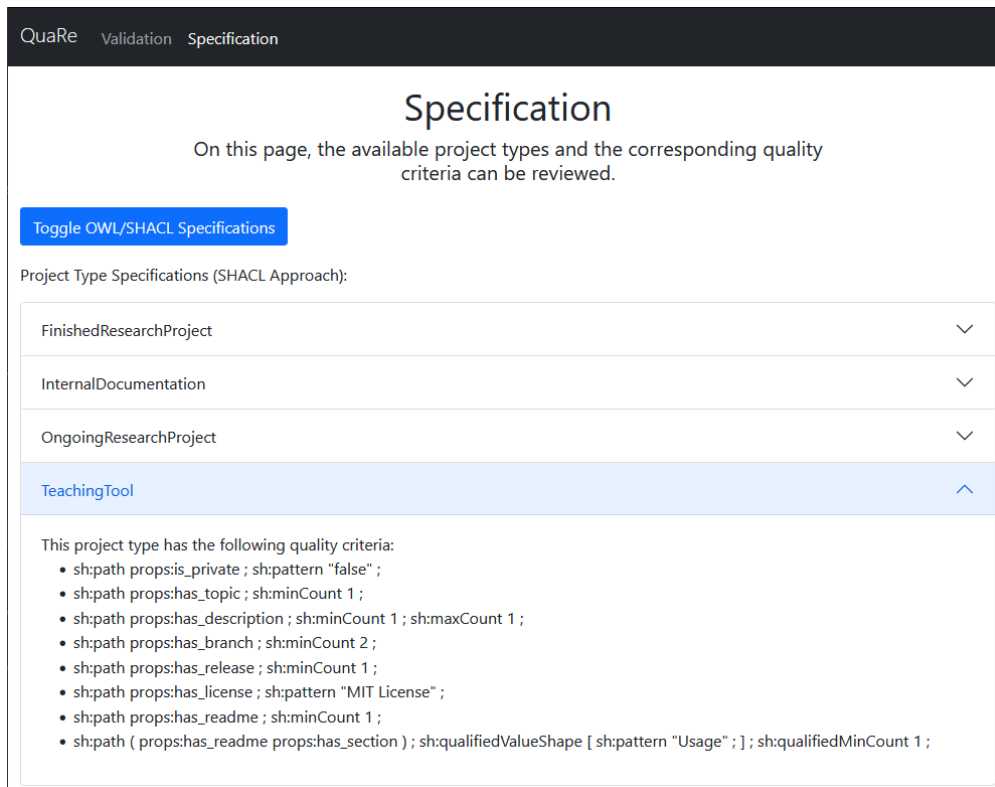


Figure 3: A screenshot of QUARE’s Specification page where the project types and the associated quality criteria can be viewed. At the moment, there are four supported project types. Editing the given specifications or adding additional ones currently requires manual changes in the backend.

5. Discussion

In Section 2, we already mentioned the comprehensive validation reports of SHACL as one of its main advantages compared to the alternative OWL approach. Apart from this, SHACL’s underlying principle of separating constraints, i.e., the shapes graph, and actual data is another strong point because it facilitates reviewing constraints for the supported project types. In contrast, the ontology of the OWL approach includes both class expressions and the repository representation impeding the readability. Thus, the SHACL approach surpasses the OWL approach in terms of developer friendliness and maintainability, too.

Regarding our implementation, QUARE lacks certain features that are planned to be added in the future. First, we want to implement that users can edit the specifications directly in the app on the *Specification* page, instead of editing them manually in the backend. Although this feature is important for QUARE, its implementation was postponed because it addresses non-trivial topics beyond the focus of this paper, i.e., the mapping of SHACL and OWL expressions to and from user interfaces (cf. [15]). Similarly, the verbalization of the explanations has to be improved significantly. Verbalization capabilities are planned to be added to the *Specification* page, as well, such that inexperienced users can easily understand the shown quality criteria.

An assessment of the tool’s performance was also conducted. For this, the time it takes for the backend to perform the SHACL and the OWL approach was measured on a standard desktop PC. As repositories, the top twenty trending GitHub repositories¹⁵ were used, which are well maintained and rather large. As intended project types, we used T_F and T_I . Figure 4 shows that the SHACL approach is faster in both scenarios even though PELLET stops the validation as soon as a violation is encountered. Hence, the SHACL approach provides a complete picture of the violations *and* is also faster. Furthermore, the figure shows that the project type does not significantly affect the runtime. This is due to the fact that fetching the repository data from GitHub is the most time-consuming part of the process. In fact, the steps 1/2/3 from Section 1 account for 0.32%/91.75%/7.89% of the total duration using the SHACL approach and for 0.10%/76.49%/23.36% using the OWL approach on average, based on our measurements. Accordingly, comparatively large (in terms of branches, issues, etc.) repositories account for the depicted outliers¹⁶. Finally, QUARE only supports GitHub repositories at the moment. Support for other platforms can be implemented as long as APIs and/or wrappers are available.

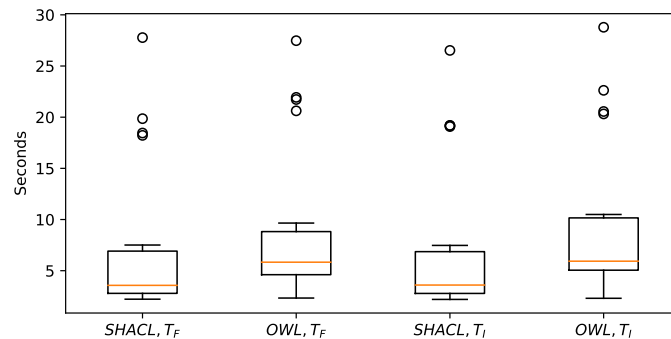


Figure 4: The distribution of the time required to process the top twenty trending repositories on GitHub. The labels on the x-axis indicate the used approach as well as the project types T_F and T_I against which the repositories were validated.

6. Conclusion

This paper proposed a SHACL based approach for validating a Git repository against a set of quality criteria specified by the type of project within the repository and compared it to an alternative OWL based approach. For this, we also provided and discussed an implementation of both approaches called QUARE. It should be emphasized that the general idea of specifying and validating requirements for certain artifacts based on RDF representations of them could be applied in other contexts as well, even for software and software systems. In fact, conversions and interfaces between the Unified Modeling Language (UML) and SHACL [16] have already been studied. Apart from the already mentioned leads, future work could thus also explore how such technologies can be leveraged to widen QUARE’s repertoire of quality criteria.

¹⁵Retrieved from <https://github.com/trending?since=monthly> (visited 2022/03/31).

¹⁶To mitigate the impact of their size, the RDF representations of repositories could be cached by exploiting RDF’s serialization capabilities. As discussed in Section 3.2, caching is not a good option for our use case, though.

References

- [1] C. M. Pilato, B. Collins-Sussman, B. W. Fitzpatrick, Version control with subversion - the standard in open source version control, O'Reilly, 2008. URL: <http://www.oreilly.de/catalog/9780596510336/index.html>.
- [2] M. Shahin, M. A. Babar, L. Zhu, Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices, IEEE Access 5 (2017) 3909–3943.
- [3] A. Gillies, Software quality: theory and management, Lulu. com, 2011.
- [4] R. Cyganiak, D. Hyland-Wood, M. Lanthaler, RDF 1.1 concepts and abstract syntax, W3C recommendation (2014). URL: <https://www.w3.org/TR/rdf11-concepts>, visited 2022/09/07.
- [5] H. Knublauch, D. Kontokostas, Shapes Constraint Language (SHACL), W3C recommendation (2017). URL: <https://www.w3.org/TR/shacl>, visited 2022/09/07.
- [6] R. A. Rossi, N. K. Ahmed, The network data repository with interactive graph analytics and visualization, in: Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, AAAI Press, 2015, pp. 4292–4293.
- [7] M. G. Antonio, K. Schick-Makaroff, J. M. Doiron, L. Sheilds, L. White, A. Molzahn, Qualitative data management and analysis within a data repository, Western Journal of Nursing Research 42 (2020) 640–648. URL: <https://doi.org/10.1177/0193945919881706>.
- [8] A. Henrich, D. Däberitz, Using a query language to state consistency constraints for repositories, in: Database and Expert Systems Applications, 7th International Conference, DEXA '96, volume 1134 of *Lecture Notes in Computer Science*, Springer, 1996, pp. 59–68. URL: <https://doi.org/10.1007/BFb0034670>.
- [9] A. Trisovic, K. Mika, C. Boyd, S. S. Feger, M. Crosas, Repository approaches to improving the quality of shared data and code, Data 6 (2021) 15.
- [10] T. B. Hashimoto, H. Zhang, P. Liang, Unifying human and statistical evaluation for natural language generation, CoRR (2019). URL: <http://arxiv.org/abs/1904.02792>.
- [11] H. Knublauch, SPIN – overview and motivation, W3C Member Submission (2011). URL: <https://www.w3.org/Submission/spin-overview>, visited 2022/09/07.
- [12] P. Hitzler, M. Krötzsch, B. Parsia, P. F. Patel-Schneider, S. Rudolph, et al., OWL 2 web ontology language primer, W3C recommendation (2009). URL: <https://www.w3.org/TR/owl2-primer>, visited 2022/09/07.
- [13] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, Y. Katz, Pellet: A practical OWL-DL reasoner, J. Web Semant. 5 (2007) 51–53. URL: <https://doi.org/10.1016/j.websem.2007.03.004>.
- [14] I. Horrocks, B. Parsia, U. Sattler, OWL 2 web ontology language direct semantics, W3C recommendation (2012). URL: <https://www.w3.org/TR/owl2-direct-semantics>, visited 2022/09/07.
- [15] J. Wright, S. J. R. Méndez, A. Haller, K. Taylor, P. G. Omran, Schímatos: A SHACL-based web-form generator for knowledge graph editing, in: The Semantic Web - ISWC 2020, volume 12507 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 65–80. URL: https://doi.org/10.1007/978-3-030-62466-8_5.
- [16] E. Stani, Metadata quality: Generating SHACL rules from UML class diagrams, in: Proceedings of the 2018 International Conference on Dublin Core and Metadata Applications, DCMI'18, Dublin Core Metadata Initiative, 2018, p. 63–64.