Visualizing Software Repositories through Requirements Trace Links

Anonymous Authors

Anonymous

***Abstract*—Tracking the status of the requirements throughout the software development cycle is essential to the success of software development projects. Requirements trace links relate requirements with other software development artifacts which indicates the progress on their related requirements. Manually identifying these trace links, understanding and aggregating them as an indicator of the requirement progress is a daunting task. This paper implements and evaluates trace links from requirements to issues, pull requests, and commits using keyword matching, TF-IDF vectors, and word vectors. Extracted links are used to create an interactive visualization of the repository in a dashboard for retrospective and real-time analysis. Our preliminary evaluation suggests that the word vector method outperforms the other methods. Our main contribution is the interactive dashboard that utilizes trace links to visualize a software repository to support project management and analysis. We also lay out details for our future evaluation plan. Our replication package contains the code and the evaluation data.**

***Index Terms*—software management, repository visualization, requirements traceability, trace graph**

I. Introduction

The success of a software development project is determined by whether the requirements are satisfied or not. Project managers need to check the progress throughout the development cycle to ensure the timely delivery of requirements. Software development is a complex process where numerous artifacts are produced. These artifacts, such as commits, issues, and pull requests, are related to one or more requirements and traceability of requirements [1] to these artifacts indicates the progress of the project. However, due to their sheer volume and dynamic nature, attempting to manually track and link these artifacts to specific software requirements and understanding the status of the project can be a daunting task.

Emerging automated approaches are used to provide more efficient software requirements traceability [2]–[6]. Natural language processing(NLP) techniques, together with information retrieval and machine learning methods increase the time efficiency of the task of requirements tracing [2].

This paper aims at visualizing software development repositories to support real-time and retrospective analysis of software development projects based on the trace links between requirements and other software development artifacts. We implement a dashboard aggregating several visual and statistical data of the project and specific requirement progress through the trace links. We analyze a given software repository and extract directed trace links from requirements to issues, pull requests, and commits using the textual attributes of these software development artifacts such as summary and message. We implement and evaluate three methods to extract trace links: *i.* keyword matching, *ii.* TF-IDF vectors, *iii.* word vectors. We store the artifacts together with their attributes and the trace links among them in a graph database and use this database to visualize the repository for software management.

The main contribution of this paper is the dashboard designed to support software development management. The dashboard presents data visualizations of repository statistics, requirements trace links, and their temporal distribution to support real-time and retrospective analysis of software repositories. We publicly share our implementation and evaluation data in our replication package.

The rest of the paper is structured as follows. Sec. II presents the related work from the literature. Sec. III details our approach for identifying trace links and selected information visualization for our dashboard. Our preliminary evaluation and future evaluation plans are described in Sec. IV. Sec. V discusses our observations, limitations of our work, and threats to validity. Finally, Sec. VI concludes the paper and lays out future work.

II. Related Work

This section presents the selected related work from the literature. We group the relevant studies based on their methods, namely, traceability visualization, information retrieval, automated classification, knowledge organization, and feature extraction.

a) Traceability Visualization: In a study, Madaki *et al*. [7] introduce a visual framework and tool named VizTraceArtefacts, which utilizes a node-link diagram where traceability information is represented by color-coded symbols. Beier *et al*. [8] develop a prototype visualization tool "Ariadne's Eye", which has an interactive design that utilizes vertical and balloon tree layouts. In addition to node-link diagram approaches, Merten *et al*. [9] study the applicability of NetMap and Sunburst diagrams for visualizing traceability information of requirements. Similarly, Chen *et al*. [10] propose a global framework of traces between source code and documentation which combines Treemap and hierarchical tree visualization techniques. Moreover, Rodrigues *et al*. [11] propose the MultiVisioTrace tool, which provides a selection between sunburst, graph, tree, and matrix views to select the most appropriate visualization method for a given traceability context.

Although many approaches are taken for visualizing traceability information and software artifacts, Li and Maalej [12] present the comparison of different visualization techniques and identify traceability graph as the most suitable approach to support management tasks.

b) Information Retrieval: Utilizing Information Retrieval methods for trace link recovery is a popular approach. Capobianco *et al*. [13] demonstrate that focusing on the nouns of software artifacts could improve the accuracy of IR-based methods in their study, which was conducted on five software artifact repositories. Bonner *et al*. [5] develop a tool using the vector-space model to identify trace links between requirements and model-based designs. They integrate their tool into the Siemens toolchain for Application Lifecycle Management (ALM). Al-Msie'Deen *et al*. [6] develop the tool named YamenTrace, which combines Latent Semantic Indexing (LSI) and Formal Concept Analysis (FCA) methods to identify traceability links. LSI is an IR technique that operates on similarities of documents to generate indicators, while FCA enables clustering and extraction of ordered concepts from datasets that have attribute-based objects. Marcus *et al*. [14] presents latent semantic analysis to extract the semantics of the documentation and the source code. A vector-space model is created for each artifact, and traceability links are captured using similarity techniques on the model. Lucia *et al*. [15] introduce a traceability recovery tool integrated within the ADAMS system, ADAMS Re-Trace. The tool utilizes the Latent Semantic Indexing method and they demonstrate the effectiveness of it in a case study involving seven student projects. Cleland-Huang *et al*. [2] develop a tool, utilizing a probabilistic-network (PN) based model which has a confidence score. The model is used to produce a list of candidate traces, sorted by confidence scores. A human analyst is required to discard the false positives. Hayes *et al*. [16] present a tool named RETRO, which utilizes the TF-IDF algorithm with relevance feedback to trace requirement specifications to defect reports. They experiment with NASA scientific instrument dataset and demonstrate a recall rate of around 85% and precision rate of around 69%.

A limitation of these methods is that the list of possible trace links they produce is not exact and further human effort is required to filter the candidates.

c) Automated Classification: Another approach is to view trace link recovery as a binary classification problem. Mills [3] examines the performance of various machine learning algorithms for trace link classification. The results show that Random Forests is the most suitable classifier for this task.

LCDTrace, a tool developed by van Oosten *et al*. [4], leverages ML classifiers. It performs trace link recovery from MDD(Model Driven Development) models to JIRA issues. Cleland-Huang *et al*. [17] approach the traceability link recovery problem with a method that combines machine learning methods and structural analysis. They train a tracing algorithm with various open-source software projects to provide automated trace-link recovery for supporting software management. In another study, Cleland-Huang *et al*. [18] present two Machine-learning methods, one requires manually created traceability matrices and the other employs web-mining techniques. These methods improve the quality of traces identified between requirement specifications and regulatory codes in their study. Mills *et al*. [19] propose an approach they named TRAIL, which also utilizes previously captured knowledge about traceability to train a classifier. However, the classifier is then used to identify new traceability links and update the existing ones, therefore addressing the challenge of maintaining updated trace information.

d) Knowledge Organization: To utilize knowledge organization techniques, Li and Cleland-Huang [20] present a method to incorporate general and domain-specific ontologies into the tracing process. They extract phrases from software artifacts and compute similarity "if a phrase found in the source artifact can be matched via concepts in the ontology to a different phrase in a target artifact". In the domain of traceability, the objective becomes labeling the source and target artifacts with nodes from a taxonomy and associating the artifacts with similar labels [21]. In the labeling process, Abdeen [22] creates a vector of concepts for both artifacts and taxonomy nodes using Wikipedia Indices and produced a list of recommended taxonomy labels for each artifact using vector similarity.

Schlutter *et al*. [23] develop a trace link recovery approach that uses semantic relation graphs and the spreading activation method. They employ an NLP pipeline to extract information from requirement specifications, create a semantic relation graph, and use spreading activation to search traces from requirements to target artifacts.

e) Feature Extraction: In addition to automated trace link recovery studies, Gallego *et al*. [24] develop a tool that identifies and extracts potential features for mobile applications, using application-related text documents such as descriptions, changelogs, and user reviews. They implement a custom NLP pipeline, utilizing POS tagging and dependency parsing techniques, to extract noun phrases. In another study, Zisman *et al*. [25] use heuristic rules that match syntactically related terms to capture the traces between requirement specifications and object models automatically. These rules create various traceability links when matches are found. In another study, Von Knethen *et al*. [26] utilize a conceptual trace model that provides semi-automatic trace link recovery and consists of a traceability technique and a supporting tool environment. Their technique provides guidance on what traces should be recorded to support later impact analyses, and they utilize the tool environment for applicability.

These studies have integrated relatively more modern technologies to minimize the effort and maximize the quality of requirement traceability.

III. Method

This section details our approach to discovering trace links in a software repository. Our approach takes a software repository and requirements as input and extracts trace links

between requirements and the software issues, commits, and pull requests (PRs) by analyzing the textual fields of these artifacts. Our prototype tool visualizes the trace links and other information on the repository. Figure 1 presents the main steps of our approach. We publicly share the implementation of our approach in our replication package[1]

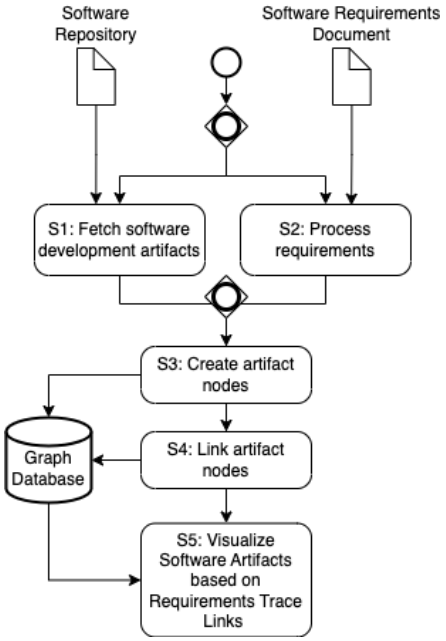

Fig. 1: Steps of our approach

The first two steps, S1 and S2, concern processing the two inputs of our approach, a software repository and natural language requirements, respectively. S1 fetches issues, pull requests, and commits from a repository whose URL is given. Our prototype expects a GitHub repository, however, our approach is general and can be applied to other repositories where the aforementioned development artifacts are present. Our prototype expects a text file containing requirements. It does not enforce specific requirements and is able to process the requirements that are written in a hierarchical structure, which is a common practice.

In S3 nodes are created for each of the requirements, issues, pull requests, and commits. The features in Table I are added as the node attributes. Our prototype uses Neo4j[2] as a graph database.

S4 links the requirements to artifacts by creating edges between their associated nodes. Two types of relationships are captured between the artifacts, namely *tracesTo* and *relatedCommit*. The *tracesTo* relationship represents a trace link between a requirement node and a software development artifact node. The *relatedCommit* is a relation between a commit and a pull request node. This relation captures provides insight into how the commits are organized by the team. In practice, requirements can trace directly to commits or via pull requests (as seen in Figure~6).

[1]https://zenodo.org/record/8076982

[2]https://neo4j.com

TABLE I: Attributes of software development artifacts used in our approach

	Requirement	Issue	PR	Commit
ID	✓	✓	✓	✓
Title	-	✓	✓	-
Description	✓	✓	✓	-
URL	-	✓	✓	✓
Number	✓	✓	✓	✓
State	-	✓	✓	-
Creation Date	-	✓	✓	-
Completion Date	-	✓	✓	✓
Message	-	-	-	✓
Comment Count	-	✓	✓	-
Comment List	-	✓	✓	-
Parent	✓	-	-	-
OID	-	-	-	✓
Text	✓	✓	✓	✓

We implement and evaluate three methods to extract trace links, which are represented with the *tracesTo* relation. The first method extracts keywords from the requirements and development artifacts and links the requirements to the artifacts that share keywords. The other methods are based on the *term frequency-inverse document frequency* (TF-IDF) vectors and *word vectors* obtained from a pre-trained model. For these methods, requirements are linked to the artifacts with similar vectors. The overview of the processing of software artifacts to extract the trace links is shown in Algorithm 1.

Algorithm 1 Trace links graph construction

Input: RSD ▷ Requirement Specification Document
Input: GRU ▷ Github Repository URL
Input: M ▷ Trace Extraction Method
Input: τ_e ▷ Threshold for Vector-Based Methods
Output: TG : `graph` ▷ Trace Graph

Phase 1 – Fetch Software Artifacts

issueList← getIssues(GRU)
prList← getPRs(GRU)
commitList← getCommits(GRU)
reqList← getRequirements(RSD)

Phase 2 – Create Graph with Artifacts

sdaList← $issueList + prList + commitList$
$TG \leftarrow$ `graph`
for each a **in** sdaList **do**
    TG.addNode(a)
end for
for each r **in** reqList **do**
    TG.addNode(r)
end for

Phase 3 – Preprocess Artifacts for Trace Extraction

$reqList \leftarrow$ lemmatize($reqList$)
$sdaList \leftarrow$ lemmatize($sdaList$)

```
if method = "vector-based" then
    reqList ← removeStopwords(reqList)
    sdaList ← removeStopwords(sdaList)
end if
```

Phase 4 – Extract Trace Links

```
for each r in reqList do
    for each a in sdaList do
        switch method do
            case keyword
                keywords ← extractKeywords(r)
                if contains(a.text, keywords) then
                    TG.addEdge(r,a)
                end if
            case tf-idf
                r-v ← createTfidfVector(r.text)
                a-v ← createTfidfVector(a.text)
                if sim(r-v, a-v) ≥ τ_e then
                    TG.addEdge(r,a)
                end if
            case word-vector
                r-v ← createWordVector(r.text)
                a-v ← createWordVector(a.text)
                if sim(r-v, a-v) ≥ τ_e then
                    TG.addEdge(r,a)
                end if
    end for
end for
    return TG
```

The *getIssues, getPRs, getCommits* functions take a project repository (GRU) and make API calls to the GitHub API[3] to fetch a list of issues, PRs, and commits respectively. It fetches the properties shown in Table I for these software software artifacts. During preprocessing phase, requirements and software artifacts are lemmatized and english stopwords are removed for vector-based methods. Thereafter, the trace links are determined according to desired method.In the keyword based method shared keywords between the requirements and software artifacts are sought. In vector based methods the requirements and artifacts are vectorized and their similarities are compared. When similarities exceed a given threshold edges are formed. For vector-based methods we utilize TF-IDF and word vectors. The following details the trace extraction methods referred to in the algorithm.

a) Keyword Case: To identify the most relevant keywords of the requirements, the following NLP methods are utilized:

- *Tokenization*: Each requirement is tokenized to obtain its words.
- *Part-of-Speech Tagging*: Each word is categorized according to its part-of-speech (POS) tags. This work specifically focuses on nouns and verbs when identifying relevant keywords.
- *Dependency parsing*: The dependency trees of the requirement sentences are obtained using spacy[4] . Figure 2 shows a dependency tree for a requirement. The verbs and nouns that are related to objects via the direct object and the object of preposition relations are used to create *verb-object* and *noun-object* pairs. The remaining verbs and nouns are also captured. All of these are used to find the artifacts that are relevant to the requirements.
- *Stopword removal*: English stopwords are used to remove the singleton nouns and verbs which are not distinguishing.
- *Project stopwords removal*: Users are allowed to provide project-specific stopwords.

Using these NLP tasks we identify the significant keywords from requirement specifications and prepare a base for identifying trace links. Figure 3 shows the extracted keywords for a given requirement.

> *1.1.3.3.3 Users shall be able to post comments on events which they attended.*
> **Extracted keywords**:
> - *verbs*: ['attend']
> - *verb-objects*: ['post comment']
> - *nouns*: ['event']
> - *noun-objects*: ['comment on event']

Fig. 3: Keyword extraction from a requirement.

The extracted keywords are matched against the textual attributes of the software artifacts to determine the trace links. These traces are stored in a graph database with the relation type *tracesTo*.

b) TF-IDF Vectors: Initially, a set of all the words in the requirements and the software development artifacts is created. Stopwords are removed from this set, from which TF-IDF vectors are created for each requirement and artifact using equations 1, 2, and 3. The requirements are linked with artifacts that have a similarity score above a given threshold (see Section IV). Figure 4 presents the steps for this method.

$$TF(t,d) = \frac{\text{frequency of t in d}}{\text{total number of terms in d}} \quad (1)$$

$$IDF(t) = log\frac{N}{1+df} \quad (2)$$

$$TF\text{-}IDF(t,d) = TF(t,d) * IDF(t) \quad (3)$$

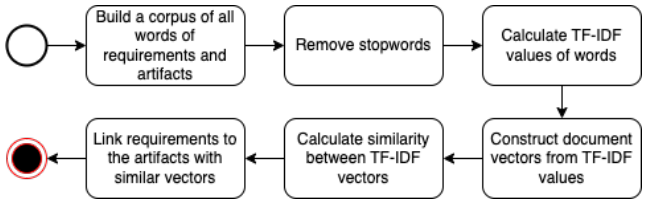

Fig. 4: Steps for extracting trace links based on TF-IDF vectors

[3]https://docs.github.com/en/graphql

[4]https://spacy.io/api/dependencyparser

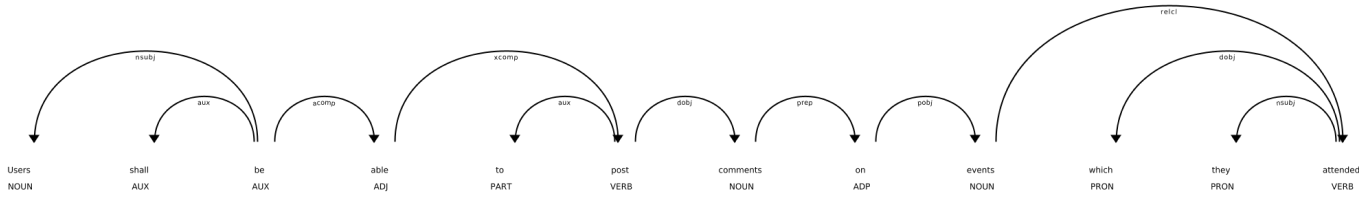

Fig. 2: The dependency tree of a requirement.

c) Word Vectors: To generate a vector for each artifact, we use a pre-trained word embeddings model (word2vec-google-news-300 [5]). We represent requirements as the average of their word vectors.

Just like in TF-IDF method, cosine similarity is used to link requirements to artifacts. The artifacts that have similarities above a predefined threshold are considered to be linked. Figure 5 presents the steps of the word vector method.

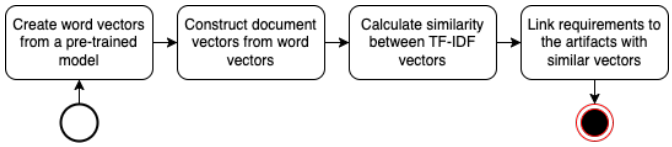

Fig. 5: Steps of trace link extraction based on word vectors

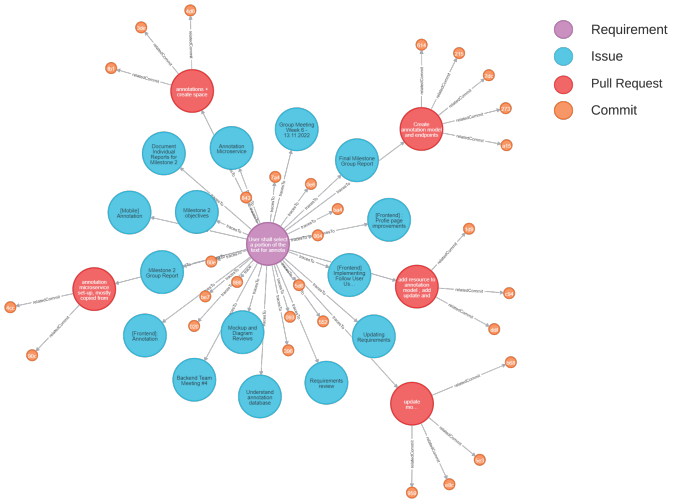

Fig. 6: A segment of a trace graph.

The obtained trace links are added to the trace graph with the *tracesTo* edge type. The relations between pull requests and commit nodes are stored with the *relatedCommit* edge type. Figure 6 illustrates a segment of a trace graph of a requirement.

The trace graph not only visualizes the software development artifacts (SDA) traced from requirements, but it also can display the lifetime of the activities over time. Figure 7 shows the SDAs related to a requirement which are arranged along a

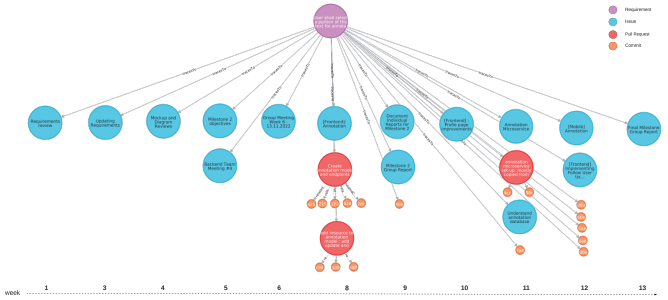

Fig. 7: A segment of a trace graph on a time axis.

time axis based on their *creationDate* property. Here, we see that first pull request related to the requirement was created in week 8, while the issues concerning the planning of the requirement were created in the early stages of the development. Such visualization is useful during the development phase of software projects as well as a while performing retrospective analyses.

Finally, S5 provides a visualization where the user can browse the software development artifacts based on the trace links. We report several types of information extracted from the repository to support software development project management. Neodash[6] is integrated into our Neo4j graph database to provide an interactive dashboard to explore the trace graph.

The dashboard presents information about a software repository and enables exploration based on trace links. First, an overview of the software artifacts for a software repository. The number of issues and pull requests is visualized in a stacked bar chart per week to view the project's progress over time. Similarly, the number of issues closed per week is visualized with a stacked bar chart. The dashboard also shows the total number of open/closed issues, open/merged PRs, and the average number of trace links per requirement. These statistics serve as a snapshot of the current state of the project. Figure 8 shows the dashboard for a repository.

The dashboard displays the trace links for a requirement over the course of the project on a weekly basis to track the progress of a requirement. It is presented using a line chart as shown in Figure 9. The project manager can visualize the data of a single requirement or compare the progress of multiple requirements. This comparison allows the users to observe the

[5]https://huggingface.co/fse/word2vec-google-news-300

[6]https://neo4j.com/labs/neodash/

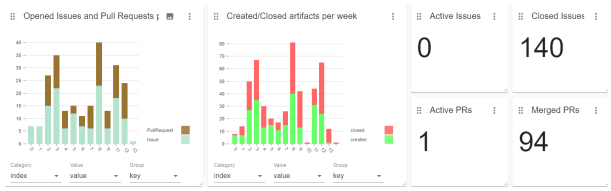

Fig. 8: Information about the software artifacts of a project. The first bar chart shows the weekly issues and pull requests created. The second bar chart shows opened and closed artifacts per week. On the right, is the total number of currently active issues and PRs and completed tasks (issues and PRs).

effort associated with each requirement since it displays the number of software development artifacts traced to them.

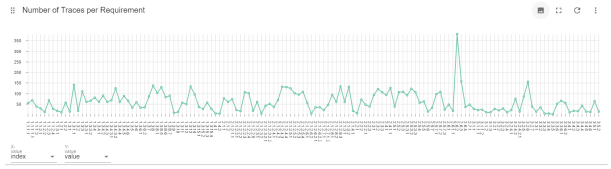

Fig. 9: The number of trace links for each requirement.

The dashboard shows a comparative view of two requirements with their weighted relations to software artifacts using a Sankey diagram (Figure 10). Here, thicker lines represent a stronger relation between the requirements to their traced artifacts. Thus, one can see how requirements are related via their associated software artifacts.

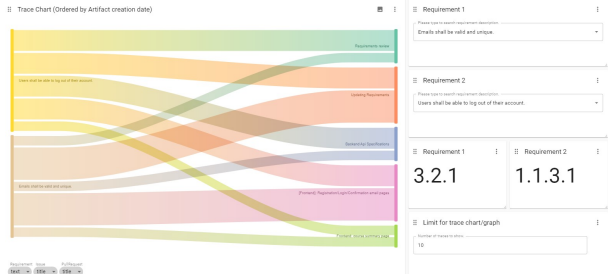

Fig. 10: The weighted relations between requirements and their associated software artifacts.

Users can interact with the dashboard to gain insights on selected requirements as seen in Figure~11. The identified traces are presented in graphical and tabular formats for selected requirements. Their current status and weekly activities are displayed in the last section of the dashboard.

IV. Evaluation

This section presents the results of our preliminary evaluation and detailed evaluation plan for the future.

We evaluate our approach on a public GitHub repository of a group of computer engineering students for their software engineering course[7]. The repository includes the project of

[7]Link Anonymized.

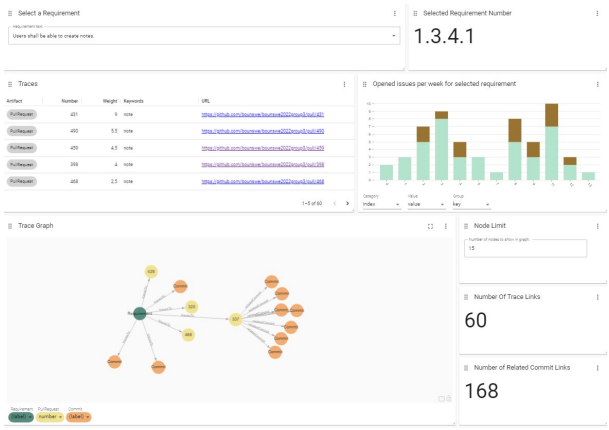

Fig. 11: Detailed information about a selected requirement.

an online learning platform. The requirements of the project are also in the repository written in a mixed format where some requirements are written as short phrases to describe a functionality whereas others as full shall statements. The requirements are structured hierarchically where a parent requirement is further refined into other requirement items.

The first two authors who are familiar with the project but not involved in the project manually extracted the trace links to serve as the ground truth. Below we report the performance of our keyword matching method in Table II and vector-based methods with various similarity thresholds in Table III.

TABLE II: Performance of the Keyword Matching Method

Method	Recall	Precision	F1 Score
Keyword extraction	0.865	0.212	0.340

TABLE III: Performance of the Vector-based Methods

Similarity Threshold	Word-vector			TF-IDF vector		
	Rec.	Prec.	F1	Rec.	Prec.	F1
0.05	1	0.043	0.082	0.839	0.121	0.211
0.15	1	0.043	0.082	0.573	0.256	0.354
0.25	1	0.043	0.082	0.244	0.43	0.311
0.35	1	0.043	0.082	0.095	0.392	0.153
0.45	0.965	0.071	0.132	0.025	0.125	0.042
0.55	0.865	0.1	0.179	0.013	0.121	0.023
0.65	0.294	0.3	0.297	0	0	–

Based on the F1 scores, the TF-IDF vector-based method has the best F1 score followed by the keyword-matching method on this repository followed by keyword matching method. The performance of the vector-based methods significantly varies according to the threshold value. Setting a low threshold links requirements with many artifacts yet few of these links are actually valid.

We do not reach any conclusions based on our preliminary evaluation. This evaluation demonstrates that our approach is applicable for tracing requirements in a software repository but we refrain to be conclusive on the performance of the

methods. In the near future, we plan two additional evaluation studies.

The first study concerns evaluating our approach again in an educational setting where we analyze the repositories of student groups, report on the perceived usefulness and usability of our dashboard, and study any correlations between the statistics reported by our dashboard and the performance of the groups in the course.

The second study is a case study with an industry partner where we ask for the ground truth from the experts from the industry and report the performance of different methods to extract trace links as well as the perceived usefulness and usability of our dashboard in comparison to trace matrix, which is widely used in the industry for tracing requirements.

V. Discussion

Observations. Our experiments on the evaluation of the tool have led us to observe that the quality and consistency of the requirement statements and the software development artifacts (SDA) significantly impact the effectiveness of our approach in identifying trace links. Well-written requirement specifications that are self-explanatory and granulated lead to better performance of its traces along with software development artifacts that have textual data (e.g. title, description, comments) that is semantically related to the feature they implement.

Limitations. The current implementation of our approach traces requirements to issues, pull requests, and commits. A more thorough trace should include other artifacts such as the architecture models, code, and test cases. Our implementation handles English only, other languages are not supported. The keyword matching method focuses on the syntax rather than the semantics of the word, yet consistent use of terms and agreeing on a glossary should mitigate this limitation.

Implications in practice. Our approach aims to visualize software repositories based on trace links. Improving the performance of trace link extraction would provide more accurate data for the dashboard visualization. We hypothesize that software engineering educators and students can benefit from this visualization to track the progress of the software projects both during and after the development phase. Visualization of traces of a requirement presents the maturity of the implementation of a requirement and the contribution patterns of students. Similarly, companies can track the progress of their projects and can gain retrospective insights on projects for good and bad practices based on the visualizations provided in our dashboard.

Threats to validity. Our evaluation presented in this paper is preliminary. To mitigate the threats to internal validity, we selected a repository the authors were not involved and two authors collaborated on building the ground truth to reduce personal bias. We cannot reach any conclusions about the external validity of the results. We need to conduct more studies to generalize the results. We share our replication package and invite the community to replicate our work for reliability.

VI. Conclusions

This paper aims to visualize software repositories through the trace links from requirements to other software development artifacts that are issues, pull requests, and commits. We extract trace links using three different methods: *i.* keyword matching, *ii.* TF-IDF vectors, and *iii.* word vectors and store the repository information and traces in a graph database. We query the database to visualize the information from the repository such as the number of traces of a requirement per week, traces of a requirement, created and closed issues, and other statistics in a dashboard to have a better understanding of the development process. We present the results of a preliminary evaluation of the performance of trace link extraction methods and our future evaluation plans. We envision our dashboard to be used for educational purposes as well as in the industry.

Our immediate future work includes a thorough evaluation of the performance of the trace link extraction methods and the usefulness and usability of our dashboard. The approach itself can be extended by incorporating other artifacts to be analyzed from the repositories. The performance of the techniques can be enhanced by using bigger and software-related corpora for the word-vector method. Our own keyword-matching technique can be compared against large language models for the same task.

References

[1] O. C. Gotel and C. Finkelstein, "An analysis of the requirements traceability problem," in *Proceedings of IEEE international conference on requirements engineering*, 1994, pp. 94–101.

[2] J. Cleland-Huang, R. Settimi, E. V. Romanova, B. Berenbach, and S. M. Clark, "Best practices for automated traceability," *IEEE Computer*, vol. 40, no. 6, pp. 27–35, 6 2007.

[3] C. Mills, "Automating traceability link recovery through classification," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 1068–1070.

[4] W. van Oosten, R. Rasiman, F. Dalpiaz, and T. Hurkmans, "On the effectiveness of automated tracing from model changes to project issues," *Information and Software Technology*, vol. 160, p. 107226, 2023.

[5] M. Bonner, M. Zeller, G. Schulz, D. Beyer, and M. Olteanu, "Automated traceability between requirements and model-based design," in *Joint Proceedings of REFSQ-2023 Workshops, Doctoral Symposium, Posters & Tools Track, andJournal Early Feedback Track. Co-located with REFSQ 2023*, 4 2023.

[6] R. Al-Msie'Deen, "Requirements traceability: Recovering and visualizing traceability links between requirements and source code of object-oriented software systems," *International Journal of Computing and Digital Systems*, pp. 1–17, 05 2023.

[7] A. Madaki and W. M. N. Zainon, "A visual framework for software requirements traceability," *Bulletin of Electrical Engineering and Informatics*, vol. 11, pp. 426–434, 2022.

[8] G. Beier and R. Müller, "Visualizing dependencies between digital product artefacts-creating a visualization layout based on a user study," *International Journal of Engineering Research & Technology*, vol. 6, pp. 8–13, 2017.

[9] T. Merten, D. Jüppner, and A. Delater, "Improved representation of traceability links in requirements engineering knowledge using sunburst and netmap visualizations," in *International Workshop on Managing Requirements Knowledge, 2011. Proceedings.*, 2011, pp. 17–21.

[10] X. Chen, J. Hosking, and J. Grundy, "Visualizing traceability links between source code and documentation," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 2012. Proceedings.*, 2012, pp. 119–126.

[11] A. Rodrigues, M. Lencastre, and G. A. De A. Cysneiros Filho, "Multi-visiotrace: Traceability visualization tool," in *10th International Conference on the Quality of Information and Communications Technology (QUATIC), 2016. Proceedings.*, 2016, pp. 61–66.

[12] Y. Li and W. Maalej, "Which traceability visualization is suitable in this context? a comparative study," in *International Conference on Requirements Engineering: foundation for software quality, 2012. Proceedings.*, vol. 7195, 03 2012, pp. 194–210.

[13] G. Capobianco, A. D. Lucia, R. Oliveto, A. Panichella, and S. Panichella, "Improving IR-based traceability recovery via noun-based indexing of software artifacts," *Journal of Software: Evolution and Process*, vol. 25, no. 7, pp. 743–762, 2013.

[14] A. Marcus and J. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Proceedings - International Conference on Software Engineering*, 06 2003, pp. 125–135.

[15] A. Lucia, F. Fasano, R. Oliveto, and G. Tortora, "ADAMS re-trace: A Traceability Recovery Tool," in *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, 04 2005, pp. 32–41.

[16] S. Yadla, J. Hayes, and A. Dekhtyar, "Tracing requirements to defect reports: An application of information retrieval techniques," *Innovations in Systems and Software Engineering*, vol. 1, pp. 116–124, 09 2005.

[17] M. Mirakhorli, Y. Shin, J. Cleland-Huang, and M. Cinar, "A tactic-centric approach for automating traceability of quality concerns," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 639–649.

[18] J. Cleland-Huang, A. Czauderna, M. Gibiec, and J. Emenecker, "A machine learning approach for tracing regulatory codes to product specific requirements," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 05 2010, pp. 155–164.

[19] C. Mills, J. Escobar-Avila, and S. Haiduc, "Automatic traceability maintenance via machine learning classification," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 369–380.

[20] Y. Li and J. Cleland-Huang, "Ontology-based trace retrieval," in *2013 7th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE)*, 2013, pp. 30–36.

[21] M. Unterkalmsteiner, "TT-RecS: The Taxonomic Trace Recommender System," in *2020 IEEE Seventh International Workshop on Artificial Intelligence for Requirements Engineering (AIRE)*, 2020, pp. 18–21.

[22] W. Abdeen, "Taxonomic trace links recommender: Context aware hierarchical classification," in *Joint of REFSQ-2023 Workshops, Doctoral Symposium, Posters and Tools Track and Journal Early Feedback, REFSQ-JP 2023, Barcelona, 17 April 2023 through 20 April 2023*, vol. 3378. CEUR-WS, 2023.

[23] A. Schlutter and A. Vogelsang, "Trace link recovery using semantic relation graphs and spreading activation," in *2020 IEEE 28th International Requirements Engineering Conference (RE)*, 2020, pp. 20–31.

[24] A. G. Marfà, Q. Motger, X. Franch, and J. Marco, "TransFeatEx: a NLP pipeline for feature extraction," in *Joint Proceedings of REFSQ-2023 Workshops, Doctoral Symposium, Posters & Tools Track, andJournal Early Feedback Track. Co-located with REFSQ 2023*, 2023.

[25] G. Spanoudakis, A. Zisman, E. Pérez-Minana, and P. Krause, "Rule-based generation of requirements traceability relations," *Journal of systems and software*, vol. 72, no. 2, pp. 105–127, 2004.

[26] A. Knethen and M. Grund, "QuaTrace: a tool environment for (semi-) automatic impact analysis based on traces," in *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, 10 2003, pp. 246– 255.