How to cite:
Mosquera, D., Ruiz, M., Pastor, O., Spielberger, J., Fievet, L. (2022). OntoTrace: A Tool for Supporting Trace Generation in Software Development by Using Ontology-Based Automatic Reasoning. In: De Weerdt, J., Polyvyanyy, A. (eds) Intelligent Information Systems. CAiSE 2022. Lecture Notes in Business Information Processing, vol 452. Springer, Cham.
https://doi.org/10.1007/978-3-031-07481-3_9

OntoTrace: a Tool for Supporting Trace Generation in Software Development by using Ontology-based Automatic Reasoning

David Mosquera[1[0000-0002-0552-7878]], Marcela Ruiz[1[0000-0002-0592-1779]], Oscar Pastor[2[0000-0002-1320-8471]], Jürgen Spielberger[1[0000-0003-2617-3535]], and Lucas Fievet[3]

[1] Zürich University of Applied Sciences, Gertrudstrasse 15, Winterthur 8400, Switzerland
{mosq, ruiz, spij}@zhaw.ch
[2] PROS-VRAIN: Valencian Research Institute for Artificial Intelligence - Universitat Politècnica de València, València, Spain
opastor@dsic.upv.es
[3] LogicFlow AG, Butzenstrasse 130, Zürich 8041, Switzerland
lucas@logicflow.ai

Abstract. Traceability in software development has gained interest due to its software maintainability and quality assurance benefits. Artifacts such as code, requirements, mockups, test cases, among others, are feasible trace sources/targets during the software development process. Existing scientific approaches support tasks like identifying untraced artifacts, establishing new traces, and validating existing traces. However, most approaches require input existing traceability data or are restricted to a certain application domain hindering their practical application. This contemporary challenge in information systems engineering calls for novel traceability solutions. In this paper, we present OntoTrace: a tool for supporting traceability tasks in software development projects by using ontology-based automatic reasoning. OntoTrace allows software development teams for inferring traceability-related data such as i) which are the traceable source/target artifacts; ii) which artifacts are not yet traced; and iii) given a specific artifact, which are the possible traces between it and other artifacts. We demonstrate how OntoTrace works in the context of the Swiss startup LogicFlow AG, supporting the traceability between functional/non-functional requirements and user interface test cases. We conclude the paper by reflecting on the experience from applying the approach in practice, and we draw on future challenges and next research endeavors.

Keywords: Software traceability, Ontology, Automatic reasoning, Trace generation, Software traceability tool.

1 Introduction

Traceability in software development refers to creating traces between software artifacts [1]. A trace is a triplet comprising a source artifact, a target artifact, and a trace link [2]. Such artifacts include source code, requirements, mockups, test cases, among

others. Keeping traceability between software artifacts facilitates quality-assurance-related tasks such as maintenance, verification, and validation tasks, which are regular practices in information systems engineering [3, 4]. In practice, the effort required to maintain, validate, and generate traces between artifacts outweighs traceability benefits [5]. Therefore, some authors propose novel approaches that allow software development teams to create traces between artifacts [5–13]. Although such approaches are helpful, some of them require as input existing traceability data sets or existing traces between artifacts [5, 6, 8, 11], hindering their practical applicability by software development teams that do not currently trace their artifacts. On the other hand, other approaches limit their scope to specific artifacts [7, 9, 10, 12, 13], lacking generality.

In this paper, we propose OntoTrace: an ontology-based automatic reasoning tool for supporting trace generation in software development projects. OntoTrace uses software development teams' context-dependent traceability ontology, representing their specific context source/target artifacts and their traces. Moreover, our approach support software development teams when defining traceability links without relying on historical traceability data sets or limiting their scope to tracing specific software artifacts. Then, software development teams can use OntoTrace to infer traceability-related information such as: i) which are the traceable source/target artifacts; ii) which artifacts are not yet traced; and iii) given a specific artifact, which are the possible traces between it and other artifacts.

To evaluate the feasibility of our approach and exemplify its application, we instantiate our approach in the context of a real-world use case at LogicFlow AG: a Swiss startup that has a traceability gap between functional/non-functional requirements and test scenarios—mainly focused on user interface (UI) test cases. We present the use of OntoTrace by using the LogicFlow AG's traceability ontology, an automatic reasoner, and a graph-like UI to visualize software artifacts and traces. We show that OntoTrace allows for establishing and discovering traceability links. Furthermore, we discuss the next research challenges to a complete technology transference.

The paper is structured as follows: in Section 2, we review the related works; in Section 3, we set up the running example describing a use case at LogicFlow AG; in Section 4, we introduce OntoTrace in the context of our running example; and, finally, in Section 5 we discuss conclusions and future work.

2 Related Work

Trace generation and discovery have gained researchers' attention, generating novel and tool-supported approaches. Some authors propose historic-data-based approaches such as artificial neural networks [5, 8], Bayes classifier [13], and similarity-based algorithms [6] for automatically creating traces between artifacts. However, such proposals require large and well-labeled training data sets based on historical traceability data, which are not always available. This represents an entry barrier for software development teams that currently do not trace their artifacts.

On the other hand, some authors propose approaches that do not rely on historical-traceability data sets, such as domain ontology-based recommendation systems [7, 13],

pattern languages [9], expert systems [10], and metamodel-based ontologies [12]. Nevertheless, such approaches are limited to generating traces on the specific artifact, lacking generality. Some proposals [7, 8, 10] limit their source/target artifacts to text-based artifacts—e.g., such as textual requirements, source code, and standard norm documents. Therefore, mockups, models, UIs, and other non-textual artifacts are beyond their scope. Similarly, other approaches limit their artifacts to model-based artifacts [12], requirements [9, 13], and source code [9, 11, 13].

To address the gaps mentioned above, we propose an ontology-based automatic reasoning tool named OntoTrace that does not rely on historical-traceability data and is not restricted to a specific set of traceable artifacts. Although some authors base their approach on ontologies [7, 10, 12, 13], the sources describing their proposed ontologies are not available for reusing them. Therefore, OntoTrace also relies on a context-independent traceability ontology, making the sources available for reuse.

3 Running Example: LogicFlow AG case

In the rest of this paper, we will use as a running example the LogicFlow AG case, a Swiss startup whose main objective is to provide a platform to facilitate the generation of UI testing in software development projects. Currently, LogicFlow AG has a web platform that allows testers to record test scenarios of web-based applications (see Fig. 1). Such test scenarios are automatically transformed into Selenium Script [14], a domain-specific language used for modeling and executing UI test cases. Moreover, LogicFlow AG's platform automatically identifies changes in the UIs, comparing current web-based application version screenshots with former web-based application version screenshots—we refer to this module as UI automatic change identifier (UI-ACI) from now on. Despite the usefulness of the LogicFlow AG platform, startup members have identified that web-based application requirements are hardly traceable to the test scenarios. Such traceability gap hinders the maintainability of test scenarios, increasing the tester's effort to keep them consistent with the requirements. In Fig. 1, we show the LogicFlow AG platform setup and the missing traces between artifacts.

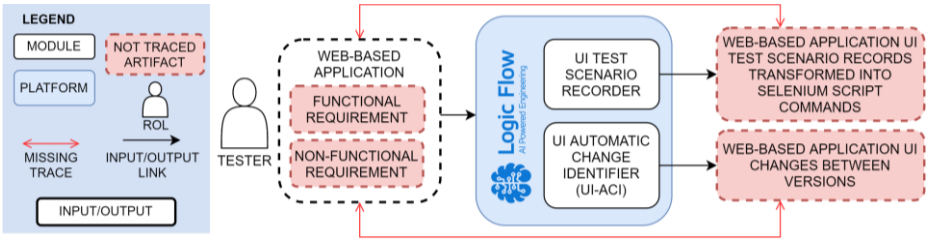

Fig. 1. LogicFlow AG platform setup and missing traces between artifacts.

For instance, a use case where such traceability gap is evident is the following: A Swiss insurance company wants to use the LogicFlow AG platform to generate test scenarios based on their web-based application for calculating insurance premiums. Therefore, the Swiss insurance company's testers create a test scenario based on the

company's requirements—i.e., the source artifacts—by using the LogicFlow AG platform. As a result, the testers create one test scenario comprising 63 Selenium Script commands. Moreover, the testers run the test scenario and compare the web-based application versions using the UI-ACI. Then, the LogicFlow AG platform's UI-ACI automatically identifies nine changes in the UI. As a result of using the LogicFlow AG platform, the testers have a set of 72 target artifacts in one test scenario. However, up to this point, the testers do not have any trace between the requirements and the test scenario, hindering the test scenario's maintainability. In Section 4, we show how this problematic case can improve by using OntoTrace.

4 OntoTrace: enabling ontology-based automatic reasoning for supporting trace generation in software development

In this section, we introduce OntoTrace and exemplify it through the running example. OntoTrace allows software development teams to infer traces among software artifacts using ontology-based automatic reasoning. To do so, OntoTrace relies on a domain-independent traceability ontology that has its foundation on general traceability definitions taken from [1, 2, 15], having terms as: trace, artifact, source artifact, target artifact, and traceability link. Therefore, as the first step to using OntoTrace, software development teams should extend such traceability ontology to their specific contexts. We fully extended the traceability ontology to the context of LogicFlow AG, including describing the source artifacts, target artifacts, and the traces between them. However, for the sake of space, in this paper, we show an excerpt of such extension (see Fig. 2).

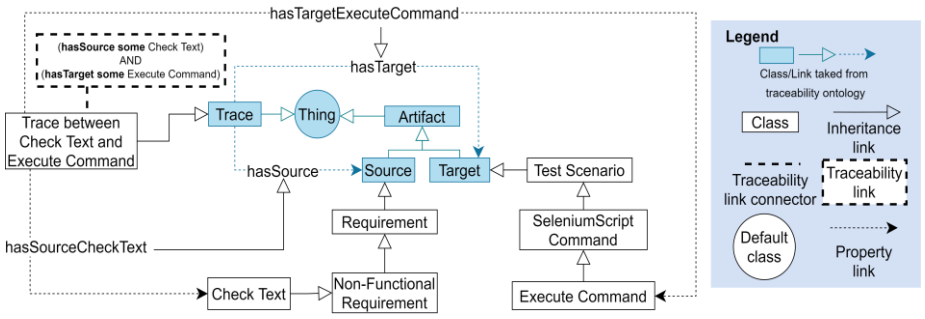

Fig. 2. Excerpt of the OntoTrace traceability ontology extension in the context of LogicFlow AG.

First, we extend the traceability ontology's *source and target* artifacts based on the LogicFlow AG context, having *requirements* as *source artifacts* and *test scenarios* as *target artifacts.* We continue increasing the class hierarchy until we identify two artifacts: *non-functional requirement check texts* as *source artifacts* and *SeleniumScript execute commands* as *target artifacts. Check text* is a non-functional requirement that checks if a text in a UI matches a specific format, font, or size. On the other hand,

LogicFlow AG testers use the *SeleniumScript execute command* to verify such non-functional requirements in a test scenario. Thus, the *trace between check text and execute command* arises between these artifacts.

Having defined the traceability ontology extension to a specific context, software development teams should use a computational-readable knowledge representation language as OWL (Ontology Web Language) [16] to describe such extended ontology. Software development teams can use OWL editors such as Protégé [17] to generate an OWL file describing the ontology. This OWL file is the primary input to use OntoTrace. Then, OntoTrace process the OWL file containing the context-dependent ontology by using three main modules: i) the automatic reasoner, ii) the SPARQL query engine, and iii) the trace graph-like visualizer (see Fig. 3).

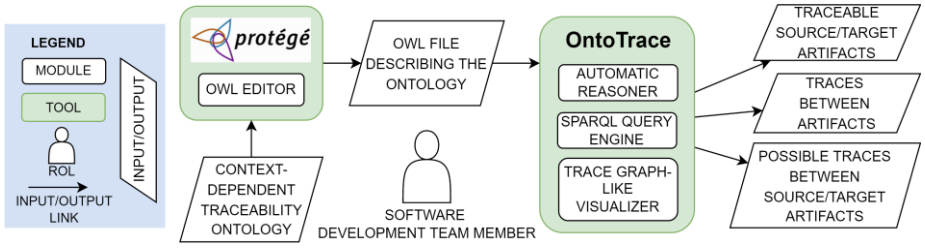

Fig. 3. OntoTrace overview.

To develop the OntoTrace modules, we use Apache Jena [18], a free-open-source Java framework for building ontology-based applications. Apache Jena allows us to integrate and develop the first two OntoTrace modules: the automatic reasoner and the SPARQL query engine. We select Pellet [19] as the OWL-based reasoner, allowing for inferring traceability-related data automatically from the context-dependent ontology. Then, we design a set of SPARQL queries to access the inferred data from the automatic reasoner. Apache Jena provides a default SPARQL query engine to execute such queries. For the sake of space, we do not show the SPARQL queries in this paper. However, we create a public GitHub repository[1] containing all the OWL files with the traceability ontology, the SPARQL queries, and the source code of OntoTrace.

After executing the SPARQL queries, the SPARQL query engine retrieves text-formatted triplets. However, we noticed that having just text-based information hinders the tool's usability. Therefore, we create a graph-like visualizer by using JgraphX [20] that allows software development teams for visualizing the following information: i) all the source/target artifact; ii) which artifacts are untraced; ii) possible traces between artifacts resulting from the automatic reasoning; and iv) the existing traces between artifacts. Thus, OntoTrace allows software development teams to generate traces between artifacts by using the information inferred through ontology-based automatic reasoning.

We test OntoTrace by using the Swiss insurance company use case in the context of LogicFlow AG. In the current status of OntoTrace, we manually create the source

[1] https://github.com/DavidMosquera/TraceabilityOntology

artifact individual instances, describing the functional and non-functional requirements. We do the same with the target artifacts, creating the individual instances that describe the test scenario. We manually populate all the ontology with individuals since OntoTrace is not yet integrated with the LogicFlow AG platform. However, in further versions of OntoTrace, we will automate populating the ontology individuals. After creating such individual instances, OntoTrace allows testers to generate the traces between the requirements and the test scenario based on the automatic reasoner inferred information. We show in Fig. 4 an excerpt of such information regarding the Swiss insurance company use case, showing the possible traces between a *non-functional requirement check text* and *target artifacts* in the *test scenario.*

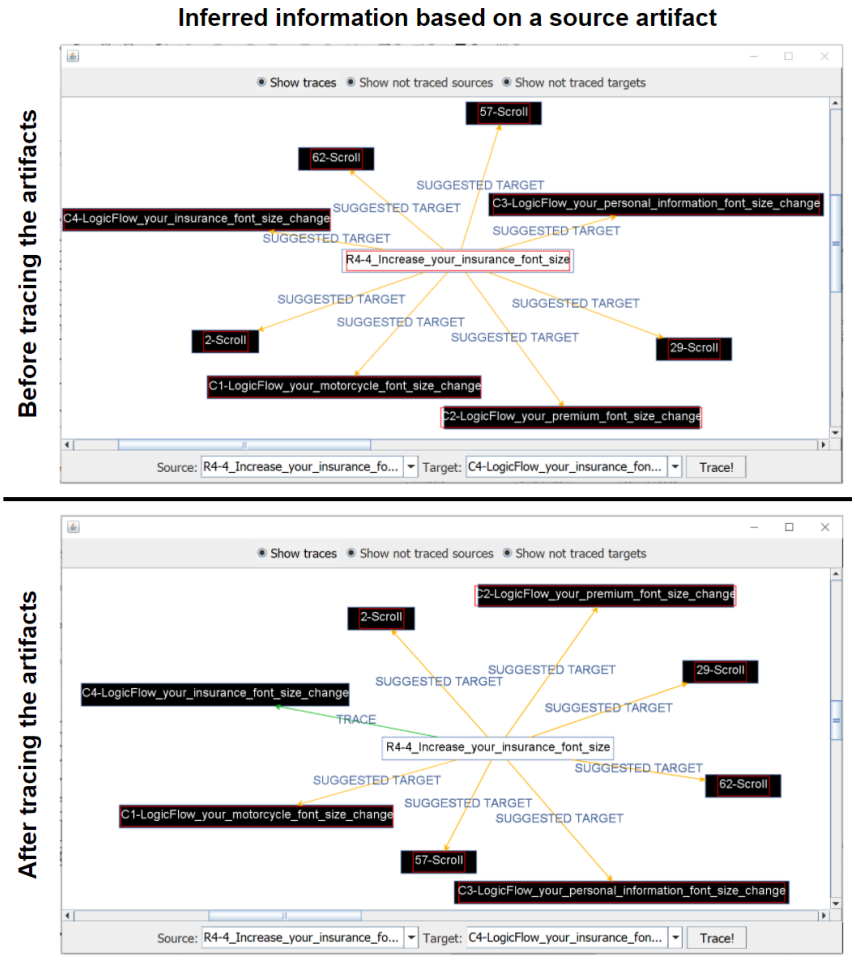

Fig. 4. Excerpt of OntoTrace showing the inferred use case information, representing the source artifacts as white boxes and the target artifacts as black boxes.

5 Conclusions and Further Work

Trace generation between software development artifacts benefits quality assurance and software maintenance [3, 4]. However, the effort required to generate such traces outweighs traceability-related benefits [5]. In this paper, we reviewed some approaches in the literature for supporting trace generation. Although such approaches are helpful, we observed some of them require historical traceability data, hindering their implementation by software development teams that do not currently trace their artifacts. On the other hand, some approaches lack generality, limiting the set of possible traceable artifacts. Consequently, in this paper, we proposed an ontology-based automatic reasoning tool for supporting trace generation named OntoTrace, which addresses the gaps mentioned above.

OntoTrace requires that software development teams extend a traceability ontology based on general traceability definitions in the literature to their software development context. Thus, software development teams describe context-dependent artifacts such as requirements, source code, test cases, among others, and the traces between them. Then, software development teams can use such ontology together with OntoTrace to automatically infer traceability information such as: i) which are the traceable source/target artifacts; ii) which artifacts are not yet traced; and iii) given a specific artifact, which are the possible traces between it and other artifacts. In this paper, we showed how OntoTrace is successfully implemented by using a running example: a Swiss startup named LogicFlow AG aiming to fulfill the traceability gap between functional/non-functional requirements and UI test cases.

As future research steps, we expect to extend OntoTrace in other directions. As the first remark, OntoTrace depends on several external tools such as Protégé, Pellet, and JgraphX. In practice, we should provide a workspace that integrates all the OntoTrace functionalities, aiming to automate steps of our approach, e.g., automatically creating individual instances. Moreover, as traces between artifacts evolve, we will include new techniques—such as machine learning algorithms—for automatically devising new traceability links while the software development team uses OntoTrace. Such techniques will support software development teams to maintain the context-dependent traceability ontology over time. Finally, other steps such as the user interaction design and empirical validation should be performed in future research endeavors.

Acknowledgments

This work has been supported by the Zürich University of Applied Sciences (ZHAW) – School of Engineering: Institute for Applied Information Technology (InIT). Moreover, we would like to thank LogicFlow AG for collaborating with us on providing data, time, and ideas during the development of this research.

References

1. Charalampidou, S., Ampatzoglou, A., Karountzos, E., Avgeriou, P.: Empirical studies on software traceability: A mapping study. Journal of Software: Evolution and Process. 33, (2021).
2. Cleland-Huang, J., Gotel, O., Zisman, A.: Software and Systems Traceability. Springer, London (2012).
3. Antoniol, G., Canfora, G., de Lucia, A.: Maintaining traceability during object-oriented software evolution: a case study. In: Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). pp. 211–219. IEEE (1999).
4. Sundaram, S.K., Hayes, J.H., Dekhtyar, A., Holbrook, E.A.: Assessing traceability of software engineering artifacts. Requirements Engineering. 15, 313–335 (2010).
5. Lin, J., Liu, Y., Zeng, Q., Jiang, M., Cleland-Huang, J.: Traceability Transformed: Generating More Accurate Links with Pre-Trained BERT Models. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). pp. 324–335. IEEE (2021).
6. Javed, M.A., UL Muram, F., Zdun, U.: On-Demand Automated Traceability Maintenance and Evolution. In: Lecture Notes in Computer Science. pp. 111–120. Springer Verlag (2018).
7. Huaqiang, D., Hongxing, L., Songyu, X., Yuqing, F.: The Research of Domain Ontology Recommendation Method with Its Applications in Requirement Traceability. In: 2017 16th International Symposium on Distributed Computing and Applications to Business, Engineering and Science (DCABES). pp. 158–161. IEEE (2017).
8. Guo, J., Cheng, J., Cleland-Huang, J.: Semantically Enhanced Software Traceability Using Deep Learning Techniques. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). pp. 3–14. IEEE (2017).
9. Javed, M.A., Stevanetic, S., Zdun, U.: Towards a pattern language for construction and maintenance of software architecture traceability links. In: Proceedings of the 21st European Conference on Pattern Languages of Programs. pp. 1–20. ACM, New York, NY, USA (2016).
10. Guo, J., Cleland-Huang, J., Berenbach, B.: Foundations for an expert system in domain-specific traceability. In: 2013 21st IEEE International Requirements Engineering Conference (RE). pp. 42–51. IEEE (2013).
11. Nagano, S., Ichikawa, Y., Kobayashi, T.: Recovering Traceability Links between Code and Documentation for Enterprise Project Artifacts. In: 2012 IEEE 36th Annual Computer Software and Applications Conference. pp. 11–18. IEEE (2012).
12. Narayan, N., Bruegge, B., Delater, A., Paech, B.: Enhanced traceability in model-based CASE tools using ontologies and information retrieval. In: 2011 4th International Workshop on Managing Requirements Knowledge. pp. 24–28. IEEE (2011).
13. Hayashi, S., Yoshikawa, T., Saeki, M.: Sentence-to-Code Traceability Recovery with Domain Ontologies. In: 2010 Asia Pacific Software Engineering Conference. pp. 385–394. IEEE (2010).
14. Selenium - Domain Specific Language, https://www.selenium.dev/documentation/guidelines/domain_specific_language/, last accessed 2021/11/29.
15. Guo, J., Monaikul, N., Cleland-Huang, J.: Trace links explained: An automated approach for generating rationales. In: 2015 IEEE 23rd International Requirements Engineering Conference (RE). pp. 202–207. IEEE (2015).
16. Web Ontology Language (OWL), https://www.w3.org/OWL/, last accessed 2021/11/29.
17. Protégé ontology editor, https://www.w3.org/2001/sw/wiki/Protege, last accessed 2021/11/29.
18. Apache Jena Home Page, https://jena.apache.org/, last accessed 2021/11/29.
19. Pellet OWL reasoner, https://www.w3.org/2001/sw/wiki/Pellet, last accessed 2021/11/29.
20. JgraphX github repository, https://github.com/jgraph/jgraphxm, last accessed 2021/11/29.