2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR) | 978-1-7281-8710-5/20/$31.00 ©2021 IEEE | DOI: 10.1109/MSR52588.2021.00073

A Traceability Dataset for Open Source Systems

Mouna Hammoudi
Johannes Kepler University
Linz, Austria
mouna.hammoudi@jku.at

Christoph Mayr-Dorn
Johannes Kepler University
Linz, Austria
christoph.mayr-dorn@jku.at

Atif Mashkoor
Johannes Kepler University
Linz, Austria
atif.mashkoor@jku.at

Alexander Egyed
Johannes Kepler University
Linz, Austria
alexander.egyed@jku.at

***Abstract*—Software engineers use requirement-to-method trace matrices to indicate the methods implementing different system requirements. Requirement-to-method trace matrices pinpoint the exact method implementing each requirement, which facilitates software maintenance and bug fixing. The code structure of a system can be used to make predictions about requirement-to-method traces. In this paper, we present a data set documenting the requirement-to-method traces as well as the code structure (methods, variables, etc.) for four open source systems. The code structure was obtained by parsing the systems under consideration and extracting the methods, variables, etc. The requirement-to-method trace matrices were obtained by resorting to students as well as to the original developers of the systems who provided us with the list of requirement-to-method traces.**

I. Introduction

Software engineers make use of requirement-to-method trace matrices to specify which methods implement which requirements [1]. The code structure of a system can be used to make conjectures about requirement-to-method traces [1]. Requirement-to-method traces help engineers save time while performing tasks such as software maintenance and bug fixing.

Background Table I represents the requirement-to-method trace matrix for a train ticket management system. A requirement-to-method trace matrix documents the tracing relationship (Trace, NoTrace, or Undefined) for each requirement and each method of the system. Thus, considering r requirements and m methods, the size of the requirement-to-method trace matrix would be $r \times m$. The columns in Table I represent the two different requirements under consideration. The cells within the leftmost column represent the seven different methods under consideration. Each entry within Table I apart from the headers represents the tracing relationship between a requirement and a method for a train ticket management system. We notice that each cell within Table I can take three values, namely T, N, or U. A T entry within a requirement-to-method trace matrix represents a tracing relationship between a method and the requirement under consideration. For instance, in Table I, we notice that methods *1-letPassengerIn* and *2-showTicketTo.* have T traces to requirement 1. This means that both methods *1-letPassengerIn* and *2-showTicketTo.* implement requirement 1. An N tracing relationship in Table I signifies that the method does not implement the requirement. For instance, we notice that method *1-letPassengerIn* has an N trace to requirement 2. This means that method *1-letPassengerIn* does not implement requirement 2. Another possible value for the tracing relationship between a method and a requirement is the U value, which means that it is unknown whether the requirement implements the given method. For instance, we notice that methods *2-showTicketTo.* and *3-proceedToPayment* have U traces to requirement 2. This means that it is unknown whether methods *2-showTicketTo.* and *3-proceedToPayment* implement requirement 2. The use of U traces is beneficial as it is better to assign a U trace to a given requirement-to-method entry instead of enforcing a T or an N trace value. The use of N and U requirement-to-method trace values represents the main difference between our dataset and the others. Indeed, other datasets solely focus on specifying T traces and assume that all of the unspecified requirement-to-method entries are automatically N traces. This is an assumption that might lead to incorrect conclusions as engineers rarely create complete and accurate requirement-to-method trace matrices [1]. Explicitly modeling undefined requirement-to-method entries can be used as an uncertainty indicator demonstrating that a developer is unsure whether there should be a Trace or a NoTrace.

TABLE I
Requirement-to-Method traces for the Train TicketManagement System (an illustration)

Method	Requirement 1	Requirement 2
1-letPassengerIn	T	N
2-showTicketTo.	T	U
3-proceedToPayment	N	U
4-proceedToPayment	N	U
5-scanTicket	T	N
6-stampTicket	T	U
7-getReceipt	T	U

Research Opportunities Our dataset allows researchers to investigate different traceability research problems such as automated trace prediction, trace maintenance, and trace repair. Specifically, researchers may investigate the correlations between the code structure and requirement-to-method trace values. This dataset can also be used to perform empirical studies by hiring software engineers and prompting them to fix bugs with or without the use of the requirement-to-method trace matrices. Such studies can help us understand the benefits of the use of requirement-to-method traces.

Related Available Datasets Some datasets such as the one provided by Qusef et al. [2], [3], [4] establish trace links between unit tests and classes. Some researchers provide data between code and non-functional requirements [5]. Other

978-1-7281-8710-5/21/$31.00 ©2021 IEEE
DOI 10.1109/MSR52588.2021.00073

TABLE II
INFORMATION ON THE FOUR STUDY SYSTEMS

	Chess	Gantt	iTrust	JHotDraw
Language	Java	Java	Java	Java
KLOC	7.2	41	43	72
#Methods	752	5013	4913	6520
#Interfaces	23	209	5	99
#Classes	104	666	718	663
#Superclasses	18	180	135	296
#Method Calls	1042	7578	12093	11413
#Class Fields	451	2452	2048	2300
#Sample Reqs	8	18	34	21
rtm_m Size	6016	90234	167042	136920

datasets provide trace links between source code and fixed bugs through patch analysis [6]. Our dataset focuses on the relationship between requirements and code. Cleland Huang et al.[7] also provide a dataset for requirement-to-code traces. Their focus is on requirement-to-class traces, but not at the method level. Requirement-to-method traces are beneficial given that they pinpoint more exactly the region of code implementing a given requirement [1]. Also, Cleland-Huang et al.'s dataset focuses on specifying the portions of code that trace to a given requirement. On the contrary, our dataset specifies the methods that trace to a requirement, the ones that do not trace to a requirement, and the ones that are unknown to trace to a given requirement. This allows different types of analysis that measure uncertainty or take uncertainty into account. In addition, our data set includes details about the code structure besides requirement-to-method traces.

II. DATASET CONSTRUCTION

A. System Selection

Candidate systems in our study are selected according to the following criteria: Systems need to exhibit a minimal complexity (i.e., at least 7 KLOC), but to be not too large to allow for a significant part of methods to be traced. If such traces were not available, we contacted the systems' developers who were willing to produce the traces. Also, we needed the systems to be open source in order to parse the source code and extract information relative to the code structure. Classes, methods, method calls, interfaces, implementations, superclasses, and subclasses were extracted using the open source library Spoon [8]. Furthermore, data dependencies were extracted for variables declared within classes (field classes), class variables used within methods (field methods), and parameters using the open source library Soot [9]. Our dataset documents the requirement-to-method traces across 81 requirements and 17,198 methods. Table II presents information about our four systems and their code structure.

B. Case Studies

1) Case Studies: Our four case studies are written in Java and are open source. The systems under consideration are Chess, Gantt, iTrust, and JHotDraw (Table II). The Chess source code is publicly available on Github and the source code for Gantt, iTrust, and JHotDraw is available on Sourceforge. Chess [10] is an application of the chess game in which two players play on a 2D board. Gantt [11] is a system that allows calendar and resource management. iTrust [12] is a system that lets patients monitor their medical history. JHotDraw [13] is a 2D graphics system that allows its user to create 2D graph structures such as architecture and design models.

2) Required Criteria for Case Study Selection: We chose these case studies as they are complex with regards to their code sizes (between 7 and 72 KLOC in size); the high amount of lines of code (LOC) is representative of software developed in industry. Also, 81 functional requirements were available for these systems along with their requirement-to-method traces. In the following, we use the term gold standard to refer to these requirement-to-method traces.

3) Trace Data Collection Process: We obtained our gold standard by paying the original developers of Chess, Gantt, and JHotDraw. Also, we asked these developers to enumerate the key requirements of the systems and we prompted them to produce traces for these requirements. The developers were given an entire week to produce the requirement-to-method traces (the gold standard). For iTrust, the list of requirement-to-method traces as well as the list of the system's core requirements were all made available on the system's website [14]. Since iTrust is frequently used as a case study in traceability research, we expect the quality of these traces to be high. As previously mentioned, developers did not specify tracing information for all requirement-to-method entries and left some undefined (U traces). Among the entries for which trace information was not specified, we can list inner Java classes, interfaces, and abstract classes. Given that the traces and the requirements were produced by the original developers of our systems, we are confident that the requirement-to-method traces produced have high quality. Indeed, the developers are extremely familiar with the source code of the systems given that they wrote the code for these systems. Therefore, we can state that we have high quality requirement-to-method traces for these systems.

Our supporting online material (SOM) [15] includes for each system information about classes, superclasses, interfaces, fieldclasses, fieldmethods, parameters, methods, method calls, requirements, and traces. The data is available under the form of JSON files as well as MySQL scripts. The source code used to extract the files listed above is available online [16].

Figure 1 represents the process followed in order to produce our dataset for each system. We parsed the source code to extract the code structure (classes, methods, interfaces, implementations, superclasses, subclasses, etc) using Spoon [8]. The data dependencies for variables were extracted using the open source library Soot. Where not available, the developers recreated a set of requirements for the system, they wrote code for. Then, they specified the requirement-to-method traces for each system, which we refer to as "developer gold". To further assess the quality of the collected traces, we relied on an experiment conducted by researchers in our group [17] in

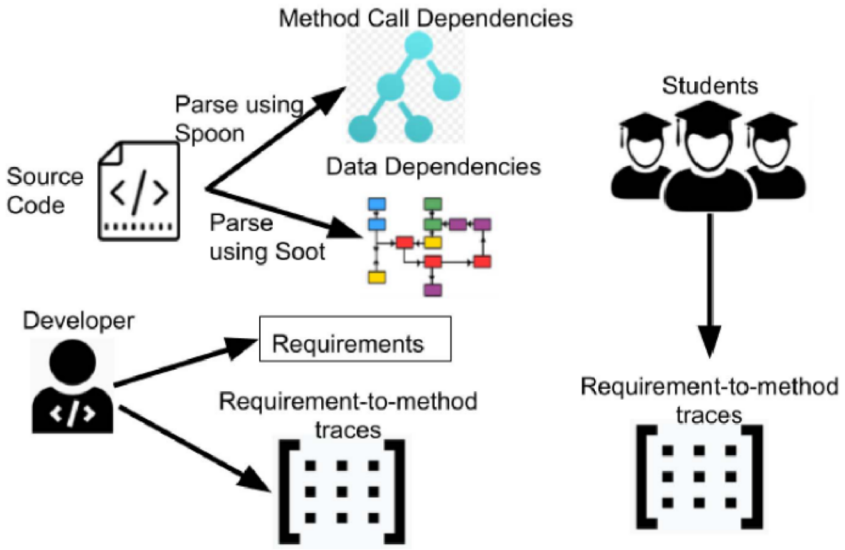

Fig. 1. Data Collection Process for each System

which they hired 100 master-level students at Johannes Kepler University and prompted them to produce tracing information for these systems. All students received training in the domain of trace recovery and had no background in regards to the case studies. About half of the students had 2-10 years of industrial experience while the other half had little or no industrial experience. All the students were not familiar with the code and had no prior knowledge about the implementation details of the case studies. Each student was given 90 minutes to complete his/her tasks. The range of student votes (*T*, *N*, or *U*) per requirement-to-method trace is 3 to 12. Also, some requirement-to-method traces did not receive any student votes, typically when they were left undefined by the original developers, as no comparison to their developer gold trace classification would be possible then.

4) Entity Relationship Diagram: Figure 2 shows the Entity Relationship Diagram (ERD) representing the different elements of the dataset and their relationships with each other. The $*$ symbol denotes a zero/one to many relationship. We notice that each class might have zero to many methods and each method is part of one class. Furthermore, each class might have one superclass and zero to many interfaces. Conversely, one interface has one to many implementations; one superclass has one to many subclasses. Each class might have zero to many variables declared (*classfield* in Figure 2). Furthermore, each method might have zero to many class field usages (*fieldmethod* in Figure 2). Each method might have zero to many parameters. The parameter's *isReturn* attribute is a Boolean flag that is set to true if it is used as the method's return value, or set to false to indicate its use as an input parameter. One method might be involved in zero to many method calls as shown by the *methodcall* entity in Figure 2. Finally, the *trace* entity specifies the gold value for each pair of requirement and method. The trace entity has *T_vote* and *N_vote* attributes that respectively refer to the number of students that voted for whether a method traces to a requirement or whether a method does not trace to a requirement.

5) Gold Standard Characteristics: Table III shows details about the amount of requirement-to-method traces collected from the original developers of our case studies. As expected, the requirement-to-method trace matrices for our four case studies are incomplete. The high percentage of *U* traces is a normal phenomenon given that engineers do not specify complete tracing information for every requirement-to-method entry within the trace matrix [1]. There could be a high percentage of *U* traces as is the case for iTrust. Despite this incompleteness, the gold standard presents us with a quantity of useful data that is more than sufficient. For example, considering JHotDraw's rtm_m, 12,658 of 136,920 of its entries have either *T* or *N* trace information. Furthermore, we observe that we have fewer *T* traces than *N* traces within our requirement-to-method entries as shown in Table III. The justification behind this imbalance is that a requirement is implemented by a small code portion (i.e., a few methods) and the majority of the remaining methods do not implement the requirement. For instance, we could only have two methods implementing a requirement and 2,000 other methods not implementing it. This explains the disparity among the percentages of *T* and *N* traces at the method level.

III. Data Usage: Example Research Questions

The gold standard for our four systems provides some ground-truth data that can be used by various researchers to explore traceability problems. We believe our dataset to be helpful to software engineers given its use of *N* traces

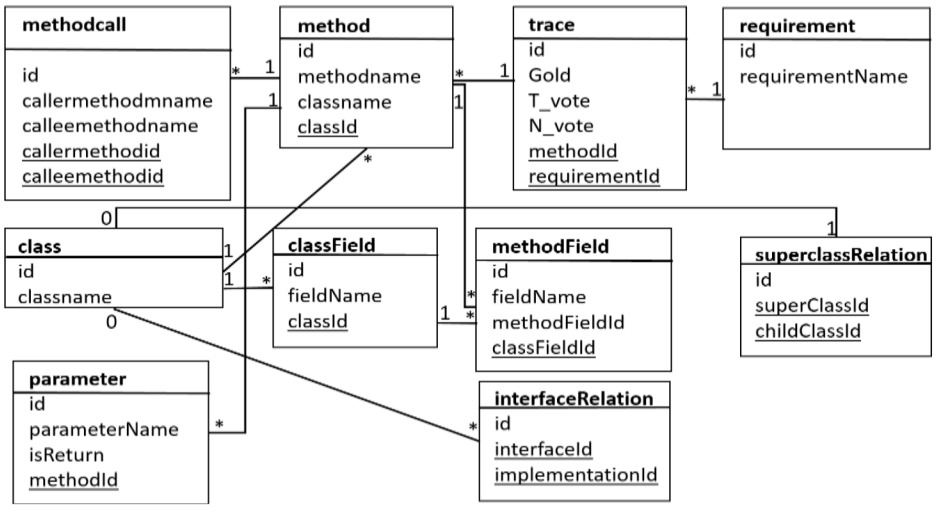

Fig. 2. Entity Relationship Diagram for our Dataset

TABLE III
QUANTIFYING THE REQUIREMENT-TO-METHOD RTM_M INPUT GOLD STANDARD

Sys.	T_m (#)	N_m (#)	U_m (#)	Total	T_m (%)	N_m (%)	U_m (%)
Chess	563	2389	3064	**6016**	9.36	39.71	50.93
Gantt	343	23166	66725	**90234**	0.38	25.67	73.95
iTrust	307	7173	159562	**167042**	0.18	4.30	95.52
JHot.	439	12219	124262	**136920**	0.32	8.92	90.76

and *U* traces besides *T* traces. This offers more accurate requirement-to-method tracing relationships to engineers, as opposed to other datasets that simply specify *T* traces within requirement-to-method trace matrices and assume that all of the remaining unpredicted traces are *N* traces. Our dataset could be used by researchers to investigate the presence of trace errors by analyzing the agreement in developer and student trace decisions [1], [18]. To this end, the students' data specifically can also be used to simulate realistic trace errors. Thus this data set allows evaluating a technique's ability to predict data in the presence of errors within requirement-to-method traces. Researchers could use automated techniques to predict requirement-to-method trace values using our dataset. Then, they could compare the predictions output by these automated techniques against our gold standard. Also, researchers could devise automated techniques involving machine learning, information retrieval, etc., using the code structure to make predictions and they could compare the output of these automated techniques against our gold standard.

IV. CHALLENGES AND LIMITATIONS

We avoid research bias by using data originating from different open source systems and produced by developers instead of this paper's authors. Furthermore, our case studies are representative of software developed in industry since their complexity is high. iTrust even includes network communication that obscures method calls (method calls cannot be observed between the client and the server). Even though all of our case studies have Java in common, the findings that can be drawn from the use of our dataset can be generalized to other programming languages different from Java. Indeed, our dataset provides trace information for methods. Also, we know that methods are a common programming construct that is encountered in other programming languages that are not necessarily object oriented, such as C, Python, C#, etc. Thus, the findings that can be reached via the use of our dataset can be generalized to other programming languages.

V. CONCLUSION

We presented a new dataset for traceability and described how it was constructed by professional software engineers. We hope that our dataset can help the software engineering community to explore different research problems related to requirement-to-code traceability.

ACKNOWLEDGEMENT

Part of this work was funded by the Austrian Science Fund (FWF) under the grant numbers P31989 and P29415-NBL and by the state of Upper Austria via LIT-2019-8-SEE-118 and the LIT Secure and Correct System Lab.

REFERENCES

[1] A. Ghabi and A. Egyed, "Code patterns for automatically validating requirements-to-code traces," in *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pp. 200–209, 2012.

[2] A. Qusef, R. Oliveto, and A. De Lucia, "Recovering traceability links between unit tests and classes under test: An improved method," in *2010 IEEE International Conference on Software Maintenance*, pp. 1–10, 2010.

[3] A. Qusef, G. Bavota, R. Oliveto, A. De Lucia, and D. Binkley, "Recovering test-to-code traceability using slicing and textual analysis," *Journal of Systems and Software*, vol. 88, pp. 147 – 168, 2014.

[4] A. Qusef, G. Bavota, R. Oliveto, A. De Lucia, and D. Binkley, "Scotch: Test-to-code traceability using slicing and conceptual coupling," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pp. 63–72, 2011.
[5] J. Cleland-Huang, R. Settimi, X. Zou, and P. Solc, "The detection and classification of non-functional requirements with application to early aspects," in *14th IEEE International Requirements Engineering Conference (RE'06)*, pp. 39–48, 2006.
[6] C. S. Corley, N. A. Kraft, L. H. Etzkorn, and S. K. Lukins, "Recovering traceability links between source code and fixed bugs via patch analysis," (New York, NY, USA), Association for Computing Machinery, 2011.
[7] "http://coest.org/."
[8] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A library for implementing analyses and transformations of java source code," *Software: Practice and Experience*, vol. 46, pp. 1155–1179, 2015.
[9] "http://www.cs.toronto.edu/ aamodkore/notes/dfa-tutorial.html."
[10] "https://github.com/warpwe/java-chess."
[11] "https://sourceforge.net/projects/ganttproject."
[12] "https://sourceforge.net/projects/itrust."
[13] "https://sourceforge.net/projects/jhotdraw."
[14] Y. Shin and L. Williams, "Work in progress: Exploring security and privacy concepts through the development and testing of the itrust medical records system," in *Frontiers in Education 36th Annual Conference*, (Los Alamitos, CA, USA), pp. 30–31, IEEE Computer Society, oct 2006.
[15] "https://doi.org/10.5281/zenodo.4453526."
[16] "https://github.com/jku-isse/tracegeneratorcdg/tree/master/tracetool."
[17] A. Egyed, F. Graf, and P. Grünbacher, "Effort and quality of recovering requirements-to-code traces: Two exploratory experiments," in *2010 18th IEEE International Requirements Engineering Conference*, pp. 221–230, 2010.
[18] M. Hammoudi, C. Mayr-Dorn, A. Mashkoor, and A. Egyed, "On the effect of incompleteness to check requirement-to-method traces," in *Proceedings of the 36th ACM/SIGAPP Symposium On Applied Computing*, SAC 2021.