On the Effect of Incompleteness to Check Requirement-to-Method Traces

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

Requirement-to-method traces reveal the code location(s) where a requirement is implemented. This is helpful to software engineers when they have to perform tasks such as software maintenance or bug fixing. Indeed, being aware of the method(s) that implement a requirement saves engineers' time, as it pinpoints the exact code region that needs to be edited to perform a bug fix or a maintenance task. Engineers produce traces manually as well as automatically. Nevertheless, traces are incomplete. This limits the amount of information that could be used by an automated technique to check further traces. Therefore, since traces are incomplete, we would like to study the effect of incompleteness on the automated assessment of requirement-to-method traces. In this paper, we apply machine learning on either incomplete or complete tracing information and we evaluate the effect of incompleteness on checking trace information. We demonstrate that the use of complete traces might yield a higher precision but yields a lower recall. Also, the use of incomplete traces yields a higher recall but a lower precision.

CCS CONCEPTS

• **Software and its engineering** → **Software creation and management**;

KEYWORDS

Traceability, Machine Learning, Requirement-to-method traces

ACM Reference Format:
Mouna Hammoudi, Christoph Mayr-Dorn, Atif Mashkoor, and Alexander Egyed. 2021. On the Effect of Incompleteness to Check Requirement-to-Method Traces. In *The 36th ACM/SIGAPP Symposium on Applied Computing (SAC '21), March 22–26, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3412841.3442021

SAC '21, March 22–26, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8104-8/21/03.
https://doi.org/10.1145/3412841.3442021

1 INTRODUCTION

Traceability reveals the code region where a requirement is implemented. Traceability facilitates program understanding and change impact analysis. It is most beneficial during software evolution when engineers tend to be the least informed about code. Researchers indicate that engineers who have little understanding of the code tend to apply changes to inappropriate code regions, which leads to errors, software decay, and wasted effort [20, 26]. Tracing relationships between code and requirements are reported under the form of requirement-to-code matrices (RTMs) [13]. Multiple standards (such as CMMI level 3) [5, 9, 22] recommend the creation of requirement-to-method trace matrices indicating the code region where every requirement is implemented. Nevertheless, the benefits of requirement-to-method traces are contingent on their availability, completeness, and correctness. Correct traces are needed to ensure that engineers are making changes to the appropriate code regions. Complete traces include the tracing relationship of every requirement to every code region. Regrettably, there is no automated technique generating complete and correct traces at an acceptable quality [6, 10, 12, 21, 24]. Thus, as of today, the most widespread technique is to manually gather traces. However, this technique is inefficient as it induces errors, entails high effort, and is not complete. For instance, one of our case studies is iTrust, which is a client server application that has 4,907 Java methods. Considering 34 of its requirements, engineers would need to manually produce $4,907 \times 34 = 166,838$ requirement-to-method traces to precisely indicate the tracing value for every requirement-to-method entry for iTrust. Reporting the trace value for every entry within a trace matrix [13] is so overwhelming that engineers tend to create incomplete requirement-to-method trace matrices, leaving many requirement-to-method entries undefined.

Researchers have devised manual as well as automated techniques to check requirement-to-method traces and ensure their correctness. However, requirement-to-method traces have various levels of incompleteness [13]. Therefore, we would like to study the effect of incompleteness on the ability

to check requirement-to-method traces. To the best of our knowledge, no other researchers have considered the effect of incompleteness on checking requirement-to-method traces and we believe our work to be the first one that does so.

Researchers demonstrated that there is a correlation between the code structure (method calls) and requirement-to-method traces [13]. For instance, if all of a method's callers and all its callees have a Trace to a requirement, then there is a high probability that the method under consideration also traces to the same requirement. Therefore, we can learn from such correlations between the code structure and the traces to check further requirement-to-method traces. Nevertheless, since traces are incomplete, we would like to investigate how this learning process is impacted by the presence of incompleteness within requirement-to-method traces. In other terms, we would like to study the variations in the precision and recall depending on the presence or the absence of incompleteness within our traces.

In this paper, we investigate how incompleteness affects the ability of our technique to learn about the correlations between the code structure and the requirement-to-method traces. We show that complete trace data might yield a higher precision but yields lower recall. Also, we demonstrate that incomplete trace data yields a higher recall but a lower precision. As input, our technique takes (1) a training set: (some) requirement-to-method traces and (2) characteristics about the code structure (i.e., mainly method calls) which are derived automatically by parsing the source code. As output, our technique provides trace checking for the requirement-to-method entries within the test set. We evaluate our technique on four open-source systems made of 7.2-72 KLOC representative of software developed in industry.

The remainder of this paper is organized as follows. Section 2 introduces our technique, Section 3 evaluates our technique, Section 4 presents the related work and finally, Section 5 provides conclusions.

2 TECHNIQUE

In the following section, we first define our trace types, we provide a motivating example and we define the features used by our technique to learn about trace information.

2.1 Trace Types

Requirement-to-method traces are represented under the form of matrices- called requirement-to-method trace matrices (RTMs). Table 1 represents an example of such an RTM for a Train Ticket Management system (the example we use in this paper). The rows in the RTM represent the different methods of the system and the columns represent the different requirements under consideration. We take into consideration requirement 1: "Proceed to payment and get ticket checked by controller" and requirement 2: "Pay fine". Every RTM entry shows the tracing relationship for a given method and a given requirement. We use the terms "T trace", "N trace", and "U trace" to respectively designate Trace, NoTrace and, Undefined tracing relationhips between

Table 1: Requirement-to-Method traces for the Train Ticket-Management 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

a requirement and a method. A T trace signifies that the method implements the requirement under consideration. For instance, the methods *2-proceedToPayment* and *7-getReceipt* have T traces to requirement 1. An N trace signifies that the method of interest does not implement a given requirement. For instance, method *3-letPassengerIn* has an N trace to requirement 1. A third possible state specific to our work is a U trace, which signifies that it is uncertain if a requirement has a T trace or an N trace to a given method. For instance, it is uncertain whether *2-proceedToPayment* and *3-letPassengerIn* have T traces or N traces to requirement 2. U traces are ignored in literature as automated techniques only focus on the generation of T traces assuming that all of the remaining requirement-to-method entries are N traces. However, U traces are frequent as engineers besides automated techniques rarely generate a complete requirement-to-method trace matrix. This motivates our technique's output of T traces, N traces, and U traces. Since our technique might not generate T trace or N trace checking for some requirement-to-method entries, it would be better to assign U traces to such entries rather than enforcing a T trace or an N trace decision. Our use of U traces and N traces represents the most important distinction between our technique and others, as traditional techniques only generate T traces, assuming that all of the remaining requirement-to-method traces are N traces.

2.2 Motivating Example - Code Structure

We apply our technique on the different features characterizing the code structure of our case studies. Our technique relies on the different features characterizing the code structure and learns about the correlations between this code structure and the requirements.

Our technique requires little information about the code structure that is easily derived automatically. The Train Ticket Management system is represented in Figure 1 and contains five classes and seven methods. Each class is represented by a rectangle with its top part showing the class name and its bottom part listing the methods within the class. The letters T, N, and U written next to each method name respectively represent T traces, N traces, and U traces to the requirement under consideration. We observe in Figure 1 that some methods call each other. For example, *1-letPassengerIn* calls *3-proceedToPayment* (1 and 3 are IDs that we use to differentiate same named methods within

distinct classes). We say that *1-letPassengerIn* is the caller of *3-proceedToPayment*, and reciprocally, *3-proceedToPayment* is the callee of *1-letPassengerIn*. The solid-line arrows in Figure 1 denote method calls, while the dashed line arrow denotes the relationship between an interface method and its implementation. Our technique checks requirement-to-method traces based on the learned correlations between the requirement-to-method trace value and the trace values of the method's calling relationships. We notice that we could in ideal cases have complete trace information for all the requirement-to-method traces as shown in Figure 1 when considering requirement 1 as shown in Table 1. However, in other cases, we could have U traces for some requirement-to-method traces as shown in Figure 2 (Requirement 2 in Table 1), in which the tracing information is only specified for one method out of seven. We define three keywords in what follows:

Definitions:

- **Complete trace Neighborhood** This refers to requirement-to-method entries that do not have any Undefined traces within their callers and callees. For instance, in Figure 1, method *6-stampTicket()* is called by *2-showTicket.()* and calls *7-getReceipt()*. Both methods *2-showTicket.()* and calls *7-getReceipt()* have T traces to requirement 1 as shown in Table 1. Thus, they do not have any Us within their callers and callees and we state that method *6-stampTicket()* has a complete trace neighborhood when considering requirement 1.
- **Incomplete trace Neighborhood** This refers to requirement-to-method entries that have one or more Undefined traces within their callers or callees. For instance, considering the same example of method *6-stampTicket()* in Figure 2, both methods *2-showTicketToController()* and *7-getReceipt()* have U traces to requirement 2. Thus, we state that method *6-stampTicket()* has an incomplete trace neighborhood when considering requirement 2.
- **Mixed Trace Neighborhood** This refers to requirement-to-method entries that are chosen randomly regardless of whether they have a complete or an incomplete trace neighborhood.

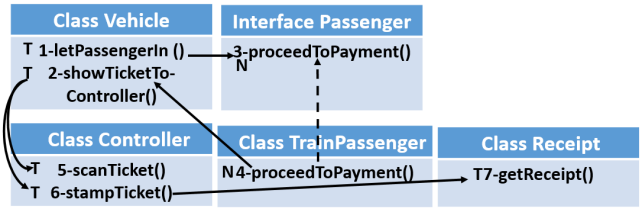

Figure 1: Code Structure of the Train Ticket Management System with Complete trace information (Requirement 1 in Table 1)

2.3 Random Forest Technique

We apply a machine learning technique to evaluate the impact of incompleteness on checking requirement-to-method traces.

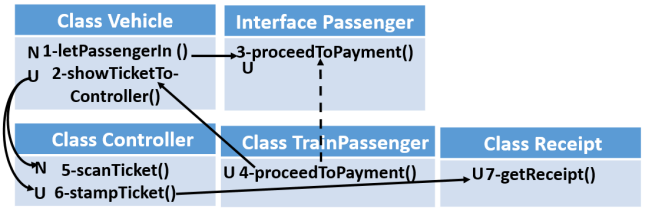

Figure 2: Code Structure of the Train Ticket Management System with Incomplete trace information (Requirement 2 in Table 1)

In order to choose which machine learning technique to use, we compared the performance of decision trees, naive bayes, K nearest neighbors, VSM, LSI, and random forest. The best precision and recall was obtained after using random forest [8]. Thus, we choose to apply the random forest technique in order to evaluate the impact of incompleteness on checking requirement-to-method traces. The source code along with the input file used for data processing are available online[1].

Our training set consists of a set of features to learn about the correlations between the code structure and the requirement-to-method traces. These features characterize the code structure. For instance, one rule that our technique could learn is that if a method has all its callers having T traces to a requirement, then the method under consideration also has a T trace to the same requirement. In the following, we enumerate the features that our technique relies on, in order to derive the set of correlations between these feature values and the trace value for a given requirement-to-method entry.

- **Class NTU:** This refers to the trace value of the owner class of a method considering a given requirement. In other words, this refers to the trace value of the requirement-to-class entry.
- **Quantities of T, N and U within Callers:** This feature characterizes the amounts of T traces, N traces, and U traces within a method's callers.
- **Quantities of T, N and U within Callees:** This feature characterizes the amounts of T traces, N traces, and U traces within a method's callees.
- **Quantities of T, N and U within Callers' Callers:** This feature characterizes the amounts of T traces, N traces, and U traces within the union of the callers' callers of a method.
- **Quantities of T, N and U within Callees' Callees:** This feature characterizes the amounts of T traces, N traces, and U traces within the union of the callees' callees of a method.

3 EVALUATION

3.1 Research Questions

In order to evaluate the correctness, completeness, and applicability of our technique, the following research questions are addressed:

[1] https://github.com/jku-isse/SAC2021

- **RQ1:** What is the distribution of incompleteness within the requirement-to-method trace matrices of our case studies? Since incompleteness is widespread within requirement-to-method trace matrices, it is important to quantify this incompleteness, as this could help us understand how to improve the precision and recall of techniques used for automated checking.
- **RQ2:** What is the effect of an incomplete versus a complete trace neighborhood on the performance of our technique? As previously stated, requirement-to-method traces have various levels of incompleteness. Therefore, it is crucial to evaluate how our technique performs when applied to an incomplete versus complete training or test sets.
- **RQ3:** How does the amount of incompleteness within a requirement-to-method trace matrix affect the use of our technique for cross project learning? Since we have four case studies and since the second research question does not distinguish among case studies, we are interested in how our technique performs when applied separately on each case study. This will help us understand if the technique using data with high, respectively low incompleteness affects the quality of estimates when applied to the remaining case studies.

3.2 Study Design

Figure 3 shows the process followed in order to conduct our study. First of all, the source code of each system is automatically parsed using the open source library Spoon [23]. This allows us to extract the features used by our technique, notably the list of methods as well as the method calls within the source code. Then, the methods are provided to the developers who produce requirement-to-method traces (RTM) for each system, meaning that they specify the tracing information for each method considering the different requirements for the given system. The RTMs for each system can then be divided into mixed, incomplete, and complete sub-RTMs, which respectively represent requirement-to-method traces with mixed, incomplete, and complete trace neighborhoods. For RQ1, we quantify the amount of incompleteness within the trace neighborhoods for our requirement-to-method traces as shown in Tables 4 and 5. In order to answer RQ2, we randomly select 50% of our RTMs as a training set and we select the remaining 50% as a test set. In order to answer RQ3, we select one random system as the training set and we consider the remaining three systems as the test set. Our *TraceChecker* technique learns about the correlations between trace values and features within the training set and checks further traces within the test set.

3.3 Case Studies

To address these research questions, we applied our technique on four case studies. The investigated systems are written in Java and are open source. The systems under consideration are Chess, Gantt, iTrust, and JHotDraw (Table 2). Chess [1] is an application of the chess game in which two players compete on a 2D board. Gantt [2] is a system that allows one to manage calendars and resources. iTrust [3] is a system that allows patients to monitor their medical history. JHotDraw [4] is a 2D graphics system that allows its user to draw 2D graph structures, such as architecture and design models.

Table 2: 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
#Sample Reqs	8	18	34	21
rtm_m Size	6016	90234	167042	136920

Table 3: 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

We selected these case studies as they are nontrivial 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. Furthermore, 81 functional requirements were available for these systems along with their requirement-to-method traces. In the following, we refer to these ground-truth traces as our gold standard and we use them to evaluate the correctness of our technique's output.

We evaluate our technique by comparing its estimations against the gold standard. For Chess, we hired a master student at Johannes Kepler University and we prompted him to produce requirement-to-method traces. This was an easy task to achieve for the student given the small size of Chess. Indeed, the student could easily familiarize himself with the source code of Chess and specify which group of methods trace to any given requirement. For Gantt and JHotDraw, we hired the key developers of these systems and we asked them to list the key requirements for these systems and produce requirement-to-method traces for these requirements. The developers were given an entire week to create the requirement-to-method traces (our gold standard). The developers were paid for performing these tasks. For iTrust, the list of the code's core functionalities as well as the list of requirement-to-method traces were all made public on the project's website by the systems' developers [29]. Given that iTrust is a commonly used system in traceability research, we conjecture that the quality of these traces is high. For all of our case studies, developers did not assign trace values for

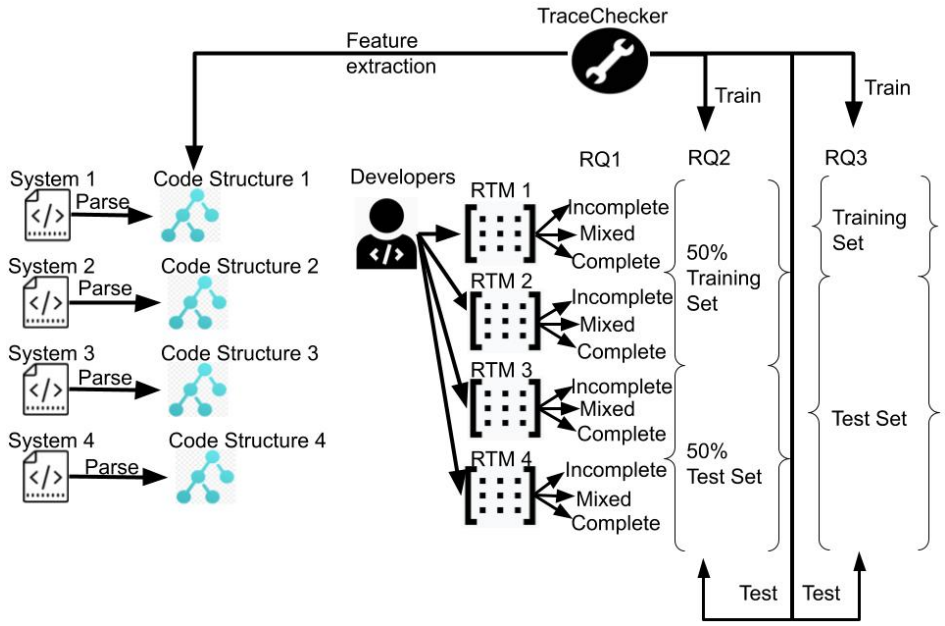

Figure 3: Study Design Steps

all requirement-to-method traces and left some undefined (U traces). Indeed, trace information was not specified for all the methods within inner Java classes, interfaces, and abstract classes.

Table 3 presents details about the amount of requirement-to-method traces available to us (gold standard available for comparison with the output of our technique). As is often the case, the requirement-to-method rtm_m (the subscript m stands for method) tends to be incomplete. The high percentage of U traces for our systems is a normal phenomenon since engineers do not specify complete tracing information between requirements and code [13]. There could be a high proportion of U traces as is the case for iTrust. In spite of this incompleteness, the gold standard presents us with a quantity of useful data that is more than sufficient. For instance, for JHotDraw's rtm_m, 12,658 of 136,920 entries have T/N trace information. Also, we notice that we have a low number of requirement-to-method entries having T traces compared to requirement-to-method traces having N traces as shown in Table 3. The reason for this is that a small area of the code (i.e, a few methods) implements a given requirement and the majority of the remaining methods do not. For instance, there could only be two methods implementing the requirement and 1,000 other methods not implementing it. This justifies the imbalance between the proportion of T and N traces at the method level. The traces used for each case study can be found as Supporting Online Material.[2]

[2]https://doi.org/10.5281/zenodo.4047965

3.4 Data Preparation

In order to quantify the amounts of callers and callees of a requirement-to-method trace, we consider the traces of the callers and callees of every requirement-to-method trace and we categorized them into four categories: H (High), M (Medium), L (Low), and N (None). We define the following four combinations of callers and callees (NoCallersUNoCalleesU, Low, Medium, and High Combinations) depending on the different amounts of U traces within the callers and callees.

Combination	Definition
$NoCallersU\ \ NoCalleesU$	$Callers^N\ \&\ Callees^N$
$LowCombination$	$\big(Callers^L\ \&\ \big(Callees^L \parallel Callees^N\big)\big) \parallel \big(Callers^N\ \&\ Callees^L\big)$
$MediumCombination$	$\big(Callers^M\ \&\ \big(Callees^M \parallel Callees^L \parallel Callees^N\big)\big) \parallel \big(Callers^N \parallel \big(Callers^L\ \&\ Callees^M\big)\big)$
$HighCombination$	$\big(Callers^H\ \&\ \big(Callees^H \parallel Callees^M \parallel Callees^L \parallel Callees^N\big)\big) \parallel \big(Callers^H\ \&\ \big(Callees^M \parallel Callees^L \parallel Callees^N\big)\big)$

We analyzed the patterns and the general distribution of the amount of U traces within our callers and callees and we created the following definitions. These definitions represent the variables used in the formulas above:

$$\text{Callees}^N \parallel \text{Callees}^N = 0$$
$$\text{Callers}^L \parallel \text{Callees}^L = 1$$
$$1 < \text{Callers}^M \parallel \text{Callees}^M \leq 5$$
$$5 < \text{Callers}^H \parallel \text{Callees}^H$$

3.5 Results

3.5.1 Quantities of U traces within callers and callees. Tables 4 and 5 both show the percentages of requirement-to-method traces having either non-existing (No Callers U No Callees U), low, medium, or high combinations of U traces within their callers and callees. Table 4 shows the percentages of T or N requirement-to-method traces having different amounts of Undefined callers and callees. Table 5 shows the percentages of U requirement-to-method traces having different amounts of Undefined callers and callees.

As we can notice from Table 4, except for Chess, the majority of requirement-to-method T or N traces have either low, medium, or high combinations of U traces within their callers and callees. Indeed, for Chess, we notice that 70% of the traces do not have any U traces within their callers and callees. Also, the traces for Gantt and JHotDraw respectively have 43% and 36% of their callers and callees not having any U traces. However, we notice that iTrust's traces only have 3% of their callers and callees not having any U traces. Also, we notice that the majority of traces for iTrust have a medium combination (52%) of U traces within their callers and callees.

From Table 5, we notice that the majority of U requirement-to-method traces have either low, medium, or high combinations of U traces within their callers and callees. Again, the traces for iTrust have the lowest percentage (12%) of U traces within their callers and callees. On the other hand, Chess, Gantt, and JHotDraw have higher percentages of traces with no U traces within their callers and callees. Indeed, Chess', Gantt's, and iTrust's traces respectively have 42%, 31%, and 24% of *NoCallers U NoCallees U* traces within their callers and callees. Thus, this demonstrates the strong presence of incompleteness within requirement-to-method traces. This motivates the need for us to investigate the effect of incompleteness on the ability to check requirement-to-method traces.

3.5.2 Evaluating the impact of incompleteness on trace checking. We selected all entries within our test set that our technique estimated to be either T or N traces and we compared them with the gold standard. We then derived the number of true positives, true negatives, false positives, and false negatives, separately for T traces and N traces, which in turn gave us the precision and recall values of our technique (see Table 6).

We will clarify our evaluation from the T trace perspective first. If our estimation and our gold standard are T traces, then this is a True Positive (TP_T). If our estimation is an N trace and our gold standard is an N trace, then this is a True Negative (TN_T). If our estimation is a T trace and our gold standard is an N trace, then this is a False Positive (FP_T). If our estimation is either an N trace or a U trace and our gold standard is a T trace, then this is a False Negative (FN_T).

From the N trace perspective, we have a True Positive (TP_N) when both our estimation and our gold standard are N traces. We have a True Negative (TN_N), when both our estimation and our gold standard are T traces. We have a False Positive (FP_N) when our estimation is an N trace and our gold standard is a T trace. We have a False Negative (FN_N) when our estimation is either a T trace or a U trace and our gold standard is an N trace.

We then apply the standard formulas for calculating precision (Prec.), recall (Rec.), and the F1 measure (F1), once for T traces ($Prec._T$, $Rec._T$, $F1_T$), and once for N traces ($Prec._N$, $Rec._N$, $F1_N$) as shown in Table 6.

Table 4: Percentages of Requirement-to-Method T traces or N traces with Undefined trace Combinations within their Callers and Callees

Sys.	NoCallersU NoCalleesU	Low Comb.	Medium Comb.	High Comb.
Chess	70	17	8	5
Gantt	43	28	23	6
iTrust	3	11	52	35
JHot.	36	22	29	12

Table 5: Percentages of Requirement-to-Method U traces with Undefined trace Combinations within their Callers and Callees

Sys.	NoCallersU NoCalleesU	Low Comb.	Medium Comb.	High Comb.
Chess	42	36	20	3
Gantt	31	33	28	8
iTrust	12	22	48	17
JHot.	24	29	34	13

3.5.3 The use of different combinations. We apply our technique using different combinations of complete and incomplete trace neighborhoods within the training set and the test set. Table 6 shows the average precision, recall, and F1 measures of our technique when using methods with different combinations of complete and incomplete trace neighborhoods over 10 runs of each combination. We also used equal sizes for the training set and the test set.

3.5.4 Technique Performance (Table 6). Table 6 shows the average T trace and N trace precision, recall, and F1 measure results obtained after applying our technique using different combinations in the types of the training and the test sets, as explained in Section 3.5.3 across ten iterations. The values between parentheses within each cell apart from the headers represent the standard deviations obtained for the precision, recall, and F1 measures across ten iterations.

Table 6: Average Precision (Prec.$_T$), Recall (Rec.$_T$), and F1 score $_T$ of our technique across ten iterations using different combinations of complete, incomplete, and mixed trace neighborhoods. Standard deviation given in brackets.

#	Combination	Prec.$_T$	Rec.$_T$	F1$_T$	Prec.$_N$	Rec.$_N$	F1$_N$
1	Mixed Train. Set+Mixed Test Set	77 (3.7)	66 (2.8)	71 (1.6)	99 (0.1)	99 (0.2)	99 (0.1)
2	Comp. Train. Set+Incomp. Test Set	77 (2.1)	60 (3.0)	68 (2.0)	99 (0.1)	99 (0.1)	99 (0.1)
3	Incomp. Train. Set+Comp. Test Set	69 (8.1)	73 (14.3)	70 (4.5)	99 (0.5)	99 (0.6)	99 (0.1)
4	Incomp. Train. Set+Mixed Test Set	74 (3.4)	71 (7.7)	72 (3.2)	99 (0.3)	99 (0.2)	99 (0.1)
5	Comp. Train. Set+Comp. Test Set	87 (1.6)	63 (1.7)	73 (1.2)	99 (0.1)	100 (0.1)	99 (0.1)
6	Incomp. Train. Set+Incomp. Test Set	77 (2.5)	71 (3.6)	74 (2.4)	99 (0.1)	99 (0.1)	99 (0.1)

Table 7: Precision (Prec.$_T$) and Recall (Rec.$_T$) and F1$_T$ of the technique when applied separately on each project using either a Mixed Training Set or a Complete Training Set

TrainingSet	TestSet	Separate Project Learning with Mixed Training Set			Separate Project Learning with CompleteTraining Set		
		Prec.$_T$	Rec.$_T$	F1$_T$	Prec.$_T$	Rec.$_T$	F1$_T$
Chess	Gantt	61	**55**	58	**79**	46	58
	iTrust	85	**70**	77	**88**	66	76
	JHotDraw	79	**49**	61	**89**	41	56
Gantt	Chess	**73**	**73**	73	72	68	70
	iTrust	**93**	**62**	74	85	46	60
	JHotDraw	79	**53**	63	**89**	38	53
iTrust	Chess	69	**88**	77	**77**	68	72
	Gantt	46	**86**	60	**81**	43	57
	JHotDraw	76	**88**	81	**89**	36	51
JHotDraw	Chess	67	**74**	70	**68**	74	71
	Gantt	40	72	52	**43**	**81**	56
	iTrust	46	**82**	59	**64**	75	69

Precision$_T$, Recall$_T$ and F1$_T$:

Default Combination (Combination 1): The first combination in Table 6 shows the results obtained when using a default combination with a mixed training set and a mixed test set. This corresponds to randomly selecting a training set and a test set without enforcing them to have complete or incomplete trace neighborhoods. The T trace precision, recall, and F1 values are respectively 77%, 66%, and 71%.

Complete Training Set (Combinations 2 and 5): We notice that using a complete trace neighborhood as a training set might yield a higher precision but yields a lower recall. Indeed, we notice through combinations 2 and 5 that the T trace precision is respectively 77% and 87%. However, the recall is respectively 60% and 63%. These recall values constitute the two lowest ones across all of our six combinations. The reason for this lower recall is the absence of important information that can only be acquired thanks to the use of an incomplete training set. This leads to a lower amount of overall estimations made when using a complete training set, which in turn leads to a higher precision and a lower recall.

Incomplete Training Set (Combinations 3, 4, and 6): Higher recall values are obtained when using an incomplete training set (combinations 3, 4, and 6) given that the incompleteness within the training set allows a richer learning process that is later applied to the test set. Using an incomplete training set and a complete test set (combination 3) allows us to achieve 69% of precision and 73% of recall. The recall using combination 3 is higher than the one obtained using combinations 2 and 5 as we are correctly retrieving a higher amount of T traces. However, the precision is lower, since the nature of our training set is different than the one of the test set. The training set is incomplete while the test set is complete, thus, this explains that some estimations are incorrectly made.

Also, we notice that the use of an incomplete training set and a mixed test set (Combination 4) yields higher values of precision and recall, which are respectively 74% and 71%. Again, these higher values are obtained thanks to the information acquired from the incomplete training set that are also applicable to the test set.

Finally, we notice that the last combination (combination 6), which is specific to using both incomplete training and test sets, yields 77% of precision and 71% of recall, and yields the highest F1$_T$ score (74%). Again, this is explained by the relevance of the information learned from the training set to the test set. Thus, since the training and test sets are similar (both incomplete), the information learned from the training set is directly applicable to the test set.

Precision$_N$, Recall$_N$ and F1$_N$: We notice that the N trace precision, recall, and F1 measures are excellent (99%) using any of the six combinations shown in Table 6. This is explained by the fact that N traces are ten times more frequent than T traces [13], which leads to an easier checking process and results in an increase of the precision, recall, and F1 measures of these estimations.

3.5.5 Technique Performance for Cross Project Learning (Table 7). Table 7 shows the results obtained after using one project as the training set and each of the remaining three projects as the test set. Furthermore, we show the variations of the precision (Prec.$_T$), recall (Rec.$_T$), and F1 (F1$_T$) measures depending on (1) the use of a mixed training set or (2) the use of a complete training set. We omit N traces, given that the precision and recall of N traces is close to 100% as shown in Table 6.

On average, we notice that the use of a mixed training set compared to a complete training set yields a lower T trace precision (an average of 68% against 77%) and a higher T trace recall (an average of 71% against 57%). This confirms the conclusions drawn from Table 6. We notice that the T trace precision when using a mixed training set is always lower than the T trace precision when using a complete training set, except for two outliers encountered when using Gantt as a training set and using either Chess or iTrust as a test set.

Similarly, we notice that the recall obtained when using a mixed training set is always greater than or equal to the recall obtained when using a complete training set, except for one outlier encountered when using JHotDraw as a training set and Gantt as a test set.

3.6 Discussion and Lessons Learned

The use of a complete training set allows us to yield a higher precision given that we are more successful in making correct estimations. However, this leads to a decrease in the recall given that we are able to retrieve less requirement-to-method traces.

Using an incomplete training set allows us to increase the recall obtained after applying our technique on the test set. This is thanks to the information acquired through the presence of incomplete trace neighborhoods. However, we achieve a lower precision when using an incomplete trace neighborhood compared to a complete one.

Table 7 reconfirms the results previously drawn from Table 6. The use of a complete training set yields an increase in the precision but a decrease in the recall, while the use of a mixed training set yields a decrease in the precision and an increase in the recall. As such, we do not notice an effect of the dataset's amount of incompleteness on whether a mixed or a complete training set performs better.

On the one hand, the mixed training allows our technique to learn how to deal with U traces and thus matches more situations, hence ultimately recommending a higher amount of T traces, which explains the higher recall. On the other hand, the use of a complete training set limits the amount of information learned with respect to incompleteness, which results in the estimations of a lower amount of requirement-to-method T traces as our technique only matches situations with low incompleteness. Having learned on situations with little or no incompleteness, the technique can give then more accurate estimations in these situations (hence the higher precision).

3.7 Threats to Validity

Internal Validity We counter researcher bias by relying on data originating from different open source systems and created by developers instead of this paper's authors. The correlations between the features for the requirement-to-method entries and their trace values were learned based on a training set that is different from the test set.

External Validity The case studies were created by different developers. Furthermore, these systems 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, our technique is not only applicable to Java. Indeed, our technique is primarily based on method calls, which are programming constructs encountered in other not necessarily object oriented programming languages, such as Python, C, etc. The only cases in which our technique is not applicable are relative to situations in which information is not transmitted via method calls. For example, our technique cannot be applied if information is transmitted via message passing or data access. For all these reasons, there is no major limitation in applying our technique to other case studies making use of method calls and written in a language different than Java.

4 RELATED WORK

To the best of our knowledge, no existing work addresses the effect of incompleteness on the estimation of requirement-to-method traces by applying machine learning in relation with the code structure. However, there are similarities between our technique and others. Indeed, Ghabi and Egyed [13] present a technique for specifying incorrect or missing traces based on method calls within the source code. Nevertheless, Ghabi and Egyed do not investigate the effect of incompleteness on the estimation of traces. Furthermore, our technique is different from theirs, given that our technique makes estimations based on 14 features specific to calling relationships. This allows us to consider all the possible cases of trace value combinations for a method's calling relationships. On the contrary, Ghabi and Egyed [13] only consider four possible combinations for a method's calling relationships. Therefore, our technique makes more accurate and precise estimations by considering all possible trace value combinations for a method's calling relationships. Another difference between our technique and Ghabi's and Egyed's is that our technique considers parsed method calls, as opposed to Ghabi's and Egyed's technique which considers input method calls resulting from program execution. Using parsed method calls is

cheaper than using executed method calls as the program execution is not required. Furthermore, parsed method calls represent a superset of the executed ones since parsed method calls constitute the total calling relationships of a method, while executed method calls are specific to the execution path undertaken during the program execution. For all these reasons, our technique has a richer set of method calls as an input compared to Ghabi's and Egyed's technique. This richer input allows our technique to make more accurate and precise estimations. This explains our technique's higher precision and recall compared to Ghabi's and Egyed's technique.

It has been demonstrated that manually capturing traces is an expensive and time consuming process [10, 12]. This has prompted the need for automated trace generation techniques that rely on information retrieval [6, 11]. **Nevertheless, all these techniques only focus on the creation of requirement-to-class T traces and assume that all of the remaining unchecked traces are N traces.** [6, 10–12, 21]. Furthermore, these techniques have a lower precision and recall compared to our technique, since they rely on text similarities among requirements and source code [10].

Ali et al. present Trustrace [6], which is a technique that uses both information retrieval and mining software repositories to generate requirement-to-class traces. However, Trustrace is limited since its success is contingent on the quality of the data within software repositories [6]. Other researchers [21] rely on the use of feedback from engineers to produce requirement-to-class traces derived from information retrieval techniques. Again, all of these techniques only focus on the creation of T traces, assuming that all of the unchecked traces are N traces. Also, these techniques do not generate complete and correct requirement-to-class traces.

Some techniques are specialized in the recovery of traces between code and artifacts other than requirements; or between requirements and artifacts other than code. For instance, Cleland-Huang et al. propose a technique using web mining by relying on the internet to get a relevant set of indicator terms to generate traces between requirements and regulatory codes [10]. Other techniques are specialized in the automatic recovery of traces between requirements and architecture [14, 19], while others recover traces between source code and documentation [7]. Guo et al. [15] suggest a technique based on a tracing network architecture that relies on Word Embedding and Recurrent Neural Network (RNN) models to generate traces among any class of artifacts (test cases, documentation, source code, etc).

Different techniques focus on haptic feedback - for example, Walters et al. [31] suggest using developers' eye gazes to generate traces. Also, Sharif et al. [28] propose generating traces between bug reports and source code using developers' eye gazes while they perform their work on an IDE. Such techniques generate useful results.

Some researchers also focus on maintaining and evolving traces across different versions of a software. For instance, Rahimi and Cleland-Huang [24, 25] present an automated technique for evolving requirement-to-class traces across successive versions of a system.

Some researchers have assessed the advantages of traceability in the software engineering lifecycle. Research shows that traceability eases change impact analysis, regression testing, and reverse engineering [17]. Also, trace correctness and completeness have received a lot of attention in the community. For instance, Kong et al. [12, 16] motivate our work as they have assessed the quality of traces after evolving them. They assume the presence of trace information and assess the consequences of trace maintenance on the quality of traces. Kong et al. observe that subjects checking requirement-to-class traces tend to worsen the quality of traces instead of improving it. Such papers demonstrate that manual trace capture is limited and should be assisted by (semi) automated trace checking [13] if possible.

Our technique is specialized in the recovery of traces between requirements and methods, regardless of the dependencies among requirements. Thus, our technique is different from feature interaction techniques [18], concept lattices [30], or concern graphs [27].

All the techniques previously enumerated do not investigate the effect of incompleteness on trace checking. Also, they focus on the study of requirement-to-class traces and do not consider requirement-to-method traces. Our technique focuses on finer-grained code regions, which offers more precise information to engineers. Indeed, our technique focuses on the exact method implementing a requirement, as opposed to other researchers focusing on the entire class implementing a requirement. Furthermore, all these techniques assume that requirement-to-method traces are complete, which is not the case in practice [13]. Studying the effect of incompleteness is important as the latter could influence the technique's performance. Thus, understanding the impact of incompleteness on trace checking could help us improve the precision and recall of all the techniques previously enumerated.

5 CONCLUSION

We presented a study about the effect of incompleteness on the ability to check requirement-to-method traces. To the best of our knowledge, our study is the first and only one that measures the variations in the precision and recall after applying machine learning on a complete versus an incomplete training or test set. In this paper, we show that using a complete training set might yield higher precision. However, it yields lower recall. Conversely, the use of an incomplete training set might yield lower precision but yields higher recall. Indeed, the presence of incompleteness within the training set offers more information for the learning process that allows the retrieval of a higher amount of requirement-to-method traces.

ACKNOWLEDGMENTS

The research reported in this paper has been funded by Austrian Science Fund (FWF) under the grant numbers P31989000 and P29415-NBL as well as the Federal Ministry of Transport, Innovation and Technology, the Austrian Federal Ministry for Digital and Economic Affairs, and the Provinces

of Upper Austria and Styria in the frame of the COMET Competence Centers for Excellent Technologies Programme managed by Austrian Research Promotion Agency FFG (K1-Centres Pro^{2}Future and SCCH).

REFERENCES

[1] [n. d.]. https://github.com/warpwe/java-chess.

[2] [n. d.]. https://sourceforge.net/projects/ganttproject.

[3] [n. d.]. https://sourceforge.net/projects/itrust.

[4] [n. d.]. https://sourceforge.net/projects/jhotdraw.

[5] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni. 2006. Model Traceability. *IBM Syst. J.* 45, 3 (July 2006), 515–526. https://doi.org/10.1147/sj.453.0515

[6] N. Ali, Y. Guéhéneuc, and G. Antoniol. 2013. Trustrace: Mining Software Repositories to Improve the Accuracy of Requirement Traceability Links. *IEEE Transactions on Software Engineering* 39, 5 (May 2013), 725–741. https://doi.org/10.1109/TSE.2012.71

[7] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. 2002. Recovering Traceability Links Between Code and Documentation. *IEEE Trans. Softw. Eng.* 28, 10 (Oct. 2002), 970–983. https://doi.org/10.1109/TSE.2002.1041053

[8] Leo Breiman. 2001. Random Forests. *Mach. Learn.* 45, 1 (Oct. 2001), 5–32. https://doi.org/10.1023/A:1010933404324

[9] Lionel C. Briand, Yvan Labiche, and Tao Yue. 2009. Automated Traceability Analysis for UML Model Refinements. *Inf. Softw. Technol.* 51, 2 (Feb. 2009), 512–527. https://doi.org/10.1016/j.infsof.2008.06.002

[10] J. Cleland-Huang, A. Czauderna, M. Gibiec, and J. Emenecker. 2010. A machine learning approach for tracing regulatory codes to product specific requirements. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. 155–164. https://doi.org/10.1145/1806799.1806825

[11] M. Eaddy, A. V. Aho, G. Antoniol, and Y. Guéhéneuc. 2008. CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis. In *2008 16th IEEE International Conference on Program Comprehension*. 53–62. https://doi.org/10.1109/ICPC.2008.39

[12] A. Egyed, F. Graf, and P. Grünbacher. 2010. Effort and Quality of Recovering Requirements-to-Code Traces: Two Exploratory Experiments. In *2010 18th IEEE International Requirements Engineering Conference*. 221–230.

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

[14] Arda Goknil, Ivan Kurtev, and Klaas van den Berg. 2010. Tool Support for Generation and Validation of Traces Between Requirements and Architecture. In *Proceedings of the 6th ECMFA Traceability Workshop (ECMFA-TW '10)*. ACM, New York, NY, USA, 39–46. https://doi.org/10.1145/1814392.1814398

[15] Jin L.C. Guo, Jinghui Cheng, and Jane Cleland-Huang. 2018. Semantically Enhanced Software Traceability Using Deep Learning Techniques. (04 2018). https://doi.org/10.1109/ICSE.2017.9

[16] Wei-Keat Kong, Jane Huffman Hayes, Alex Dekhtyar, and Jeff Holden. 2011. How Do We Trace Requirements: An Initial Study of Analyst Behavior in Trace Validation Tasks. In *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE '11)*. ACM, New York, NY, USA, 32–39. https://doi.org/10.1145/1984642.1984648

[17] Patrick Mader and Alexander Egyed. 2015. Do developers benefit from requirements traceability when evolving and maintaining a software system? *Empirical Software Engineering* 20, 2 (01 Apr 2015), 413–441. https://doi.org/10.1007/s10664-014-9314-z

[18] Marius Marin, Arie Van Deursen, and Leon Moonen. 2007. Identifying Crosscutting Concerns Using Fan-In Analysis. *ACM Trans. Softw. Eng. Methodol.* 17, 1, Article 3 (Dec. 2007), 37 pages. https://doi.org/10.1145/1314493.1314496

[19] M. Mirakhorli and J. Cleland-Huang. 2016. Detecting, Tracing, and Monitoring Architectural Tactics in Code. *IEEE Transactions on Software Engineering* 42, 3 (March 2016), 205–220. https://doi.org/10.1109/TSE.2015.2479217

[20] P. Mäder and A. Egyed. 2011. Do software engineers benefit from source code navigation with traceability? — An experiment in software change management. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. 444–447. https://doi.org/10.1109/ASE.2011.6100095

[21] A. Panichella, C. McMillan, E. Moritz, D. Palmieri, R. Oliveto, D. Poshyvanyk, and A. De Lucia. 2013. When and How Using Structural Information to Improve IR-Based Traceability Recovery. In *2013 17th European Conference on Software Maintenance and Reengineering*. 199–208. https://doi.org/10.1109/CSMR.2013.29

[22] David Lorge Parnas. 1994. Software Aging. In *Proceedings of the 16th International Conference on Software Engineering (ICSE '94)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 279–287. http://dl.acm.org/citation.cfm?id=257734.257788

[23] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2015. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience* 46 (2015), 1155–1179. https://doi.org/10.1002/spe.2346

[24] Mona Rahimi and Jane Cleland-Huang. 2018. Evolving software trace links between requirements and source code. *Empirical Software Engineering* 23, 4 (01 Aug 2018), 2198–2231. https://doi.org/10.1007/s10664-017-9561-x

[25] M. Rahimi, W. Goss, and J. Cleland-Huang. 2016. Evolving Requirements-to-Code Trace Links across Versions of a Software System. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 99–109. https://doi.org/10.1109/ICSME.2016.57

[26] M. P. Robillard, W. Coelho, and G. C. Murphy. 2004. How effective developers investigate source code: an exploratory study. *IEEE Transactions on Software Engineering* 30, 12 (Dec 2004), 889–903. https://doi.org/10.1109/TSE.2004.101

[27] Martin P. Robillard and Gail C. Murphy. 2002. Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*. ACM, New York, NY, USA, 406–416. https://doi.org/10.1145/581339.581390

[28] Bonita Sharif, John Meinken, Timothy Shaffer, and Huzefa Kagdi. 2017. Eye movements in software traceability link recovery. *Empirical Software Engineering* 22, 3 (01 Jun 2017), 1063–1102. https://doi.org/10.1007/s10664-016-9486-9

[29] Yonghee Shin and L. Williams. 2006. 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*. IEEE Computer Society, Los Alamitos, CA, USA, 30–31. https://doi.org/10.1109/FIE.2006.322599

[30] Paolo Tonella. 2003. Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis. *IEEE Trans. Softw. Eng.* 29, 6 (June 2003), 495–509. https://doi.org/10.1109/TSE.2003.1205178

[31] Braden Walters, Timothy Shaffer, Bonita Sharif, and Huzefa Kagdi. 2014. Capturing Software Traceability Links from Developers' Eye Gazes. In *Proceedings of the 22Nd International Conference on Program Comprehension (ICPC 2014)*. ACM, New York, NY, USA, 201–204. https://doi.org/10.1145/2597008.2597795