Poster: Identifying TraIn: A Neglected Form of Socio-Technical Incongruence

Xiao Wang [*], Lu Xiao [*], Ye Yang [*], Xinyi Xu [†], Yu Jiang [‡]
xwang97, lxiao6, yyang4@stevens.edu, xuesu@bupt.edu.cn, jy1989@mail.tsinghua.edu.cn
[*]Stevens Institute of Technology, [†]Beijing University of Posts and Telecom., [‡]Tsinghua University

ABSTRACT

Socio-Technical Congruence (STC) indicates that social interactions among developers should be congruent with technical dependencies among their tasks. Prior research discovered that the lack of the "should-happen" communication will lead to integration errors and productivity decrease. However, the opposite scenario, excessive communication not matched by any technical dependencies, has been largely neglected. This paper terms such scenario as *Transgressive Incongruence* (*TraIn*). To automatically pin-point source files involved in *TraIn*, this paper defines a new form of coupling between files, called *communication coupling*. It measures the communication traffic among developers working on two files. Evaluation on 6 Apache open source projects reveals: 1) the *communication coupling* between files with structural dependencies is 3 to 10 times higher than that between files independent from each other; and 2) source files involved in *TraIn* are usually very bug-prone. This implies that *TraIn* may have negative impact on the quality of software systems, and thus should merit due attention.

KEYWORDS

Social-Technical Congruence, Communication Coupling, Design Structure Matrix

1 INTRODUCTION

Software development is both a social and technical activity. Socio-Technical Congruence (STC) indicates that the actual communication links among developers should be congruent with the expected communication needs implied by the technical dependencies among their tasks. Prior research has reached a general consensus that high STC is beneficial for improving software development productivity and quality [1, 4].

There are two scenarios where STC is violated: 1) the "should" happen communication between developers is lacking despite the existing technical dependencies between their tasks; and 2) to the opposite, there exists excessive communication traffic between developers who work on mutually independent tasks. Prior research has only focused on the first scenario that leads to integration errors and productivity decrease [1, 4]. Research on the second scenario has been largely neglected.

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3195008

In this paper, we term the second scenario as *Transgressive Incongruence* (*TraIn*). To identify *TraIn*, this paper defines and measures a new form of coupling between files, called *communication coupling*. The *communication coupling* between two source files measures the relevant communication traffic (e.g. number of email threads) among developers contributing to the files. Investigation of 6 Apache projects (Cassandra, Camel, Hadoop, CXF, PDFBox, HBase) shows that on average the *communication coupling* congruent with structural dependencies is 3 to 10 times of the *communication coupling* not matched by any dependencies. In other words, developers working on structurally dependent files indeed are more likely to communicate with each other compared to those working on independent files. Oppositely, high *communication coupling* without matching structural dependencies, i.e. *TraIn*, may reveal suspicious and implicit connections that merit attention. Leveraging *communication coupling*, we developed an automatic approach to pin-point source files that involved in *TraIn*. Evaluation on the 6 projects shows that files involved in *TraIn* are also likely to be bug-prone. The implication is that *TraIn* is likely to cause nagative impact on the quality of software systems, and thus merit attention.

2 APPROACH

This section introduces our three-step approach to identify *TraIn*.

Step1: Extracting Structural Dependencies. We leverage a commercial code analysis tool, Understand [1], to reverse-engineer the source code. It extracts the various types of structural dependencies among files. It captures 9 common types of dependencies between files in objected-oriented programming languages:

f_a *Implement* f_b:	f_a (concrete class) implements f_b (interface)
f_a *Extend* f_b:	f_a (child class) extends f_b (parent class)
f_a *Call* f_b:	f_a calls the methods defined in f_b
f_a *Throw* f_b:	f_a throws an exception in the type of f_b
f_a *Cast* f_b:	f_a is casted into the type of f_b
f_a *Create* f_b:	f_a creates an instance of f_b
f_a *Aggregate* f_b:	f_b is an attribute of f_a
f_a *Use* f_b:	f_a uses a value of f_b
f_a *Import* f_b:	f_b are imported at the headers of f_a

Step2: Measuring Communication Coupling. Developers frequently exchange emails to discuss potential issues that are relevant to the tasks (i.e. source files) they work on. The communication traffic between developers reflects a form of coupling between the files they contribute to. We measure the *communication coupling* between two files f_x and f_y as the sum of the number of *relevant* email threads between each pair of developers working on f_x and

[1]https://scitools.com/

f_y. The challenge is that there is no effective way to figure out exactly which emails are relevant to which files. As an approximation heuristic, we use the the developers' effort distribution on files as a proxy for the percentage of relevant communication. For example, if a developer spends 30% of his effort (measured by the lines of code written by him) on a file, we assume that 30% of his emails are relevant to that file. The *communication coupling* between any two files can be calculated by mining the mailing list and the code repository.

Step3: Identifying Transgressive Incongruence. We can automatically identify *TraIn* by comparing the structural dependencies and *communication coupling* among files. Given a pre-specified threshold Thd, two files f_x and f_y are defined to be involved in *TraIn* if and only if: 1) the *communication coupling* between them is above Thd; and 2) f_x and f_y are structurally independent from each other. The input threshold controls the severity of the identified *TraIn*: the higher the threshold, the more severe is the *TraIn*.

3 EVALUATION

To evaluate the effectiveness of the proposed *communication coupling* and *TraIn* identification approach, we studied 6 Apache open source projects: Camel, Cassandra, CXF, Hadoop, HBase, and PDFBox. These projects are long-lived (survived 7 to 10 years) and well-accepted projects in the Apache community. They are all in industry-scale with 635 (PDFBox) to 9865 (Camel) source files and 3971 (CXF) and 19613 (HBase) email records. We evaluate our measurement and approach using two research questions below:

RQ1: Is communication coupling an effective measurement for analyzing TraIn*?* The baseline of *STC* is that the technical dependencies among software artifacts imply the communication needs among the participating developers. This RQ tests whether *communication coupling* reflects such baseline. If not, *communication coupling* is not an effective measurement for investigating *TraIn*. To answer this question, we calculate the ratio of the average *communication coupling* between files with structural dependencies divided by the average *communication coupling* of files mutually independent from each other. Figure 1 shows such ratios associated with each dependency type in the 6 projects. The x-axis is the structural dependency type. The y-axis is the ratio. We observe that 1) in all the 6 projects, *communication coupling* congruent with a *general* type (i.e. any combinations of structural dependency types) is averagely 3.37 to 10.86 times of the *communication coupling* without matching structural dependencies; 2) despite the various characteristics of the 6 projects, the *Use*, *Import*, and *Call* dependency types, always imply higher (1.35 and 11.86 times of) communication among developers compared to no structural dependencies. Therefore, we conclude that *communication coupling* is an effective measurement for investigating *TraIn*.

RQ2: Are source files involved in TraIn *also bug-prone?* We hypothesize that files involved in *TraIn* are also likely to be bug-prone. This is because files sharing implicit connections (which are not captured by structural dependencies) are likely to cause problems and thus excessive communication. To test this hypothesis, we use the Cohen's Kappa [2, 3] to evaluate the agreement between files involved in *TraIn* and the top 30% most bug-prone files in a project. We investigate *TraIn* of four levels of severities by setting the threshold to be the top 3%, 5%, 10%, and 20% highest *communication coupling*. The result is shown in Table 1. We can make two observations. First, the *TraIn* files (when threshold is the top 3% *communication coupling*) and the top 30% bug-prone files have substantial (agreement = 0.61 for Hadoop) to almost perfect (agreement = 0.95 for Cassandra) agreement. Second, as the *TraIn* threshold increases, the agreement also increases, implying that the higher the severity of *TraIn*, the more likely the files are bug-prone. Therefore, we conclude that files involved in *TraIn* are also highly likely to be bug-prone. As the severity of *TraIn* increases, the *TraIn* files are more likely to be bug-proneness.

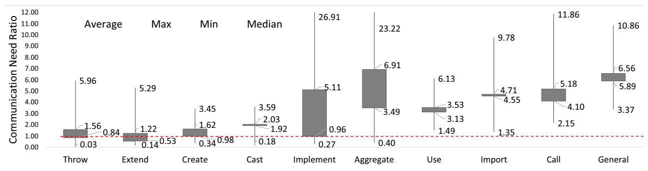

Figure 1: Communication Need Ratio for Dependencies

Table 1: Agreement between *TraIn* and Bug-proneness

Subject	Thd=20%	Thd=10%	Thd=5%	Thd=3%
Camel-2.12.4	0.32	0.7	0.6	0.77
Cassandra-2.1.2	0.86	0.9	0.92	0.95
CXF-3.0.0	0.62	0.69	0.73	0.77
Hadoop-2.5.0	0.39	0.52	0.51	0.61
HBase-0.98.2	0.7	0.74	0.79	0.74
PDFBox-1.8.7	0.59	0.66	0.7	0.63

4 CONCLUSION

In conclusion, this paper investigated *TraIn*, a largely neglected form of Socio-Technical incongruence in existing literature. This paper defined and measured a new form of coupling among files, called *communication coupling*, as well as contributed an approach to automatically identify the occurrences of *TraIn*. A study of six Apache open source projects reveals that the *communication coupling* among files with structural dependencies are indeed significantly higher than those without structural dependencies. Files involved in the occurrences of *TraIn* are highly likely to suffer from bug-proneness. The higher the severity of *TraIn*, the more likely the involved files suffer from quality problem. The implication of this study is that *TraIn* is also an important form of Socio-Technical incongruence that deserve serious attention.

REFERENCES

[1] M. Cataldo and J. D. Herbsleb. 2013. Coordination Breakdowns and Their Impact on Development Productivity and Software Failures. *TSE* 39, 3 (March 2013), 343–360. https://doi.org/10.1109/TSE.2012.32
[2] J.L. Fleiss. 1981. *Statistical Methods for Rates and Proportions*. Wiley. https://books.google.com/books?id=8Z8QAQAAIAAJ
[3] Joseph L. Fleiss, Bruce Levin, and Myunghee Cho Paik. 2004. *The Measurement of Interrater Agreement*. John Wiley and Sons, Inc., 598–626. https://doi.org/10.1002/0471445428.ch18
[4] John C. Georgas and Anita Sarma. 2011. STCML: An Extensible XML-based Language for Socio-technical Modeling. In *4th CHASE*. ACM, New York, NY, USA, 61–64. https://doi.org/10.1145/1984642.1984657