Identifying Failure Inducing Developer Pairs Within Developer Networks

Jordan Ell
University of Victoria, Victoria, British Columbia
jell@uvic.ca

***Abstract*—Software systems have not only become larger over time, but the amount of technical contributors and dependencies have also increased. With these expansions also comes the increasing risk of introducing a software failure into a pre-existing system. Software failures are a multi-billion dollar problem in the industry today and while integration and other forms of testing are helping to ensure a minimal number of failures, research to understand full impacts of code changes and their social implications is still a major concern. This paper describes how analysis of code changes and the technical relationships they infer can be used to detect pairs of developers whose technical dependencies may induce software failures. These developer pairs may also be used to predict future software failures as well as provide recommendations to contributors to solve these failures caused by source code changes.**

I. Introduction

Large software projects are created using highly modular and reusable code. This creates technical dependencies between methods or functions that can be used in a wide variety of locations throughout the project. This causes changes to any given method to have a rippling effect across the rest of the project [1]. The larger these effects are, the more likely they are to cause a software failure inside the system during the project's life span [2]. These observations of technical dependencies open the door to types of analysis on the developer networks they infer and preventing software failures by improving coordination amongst dependant developers.

Technical dependencies in a project can be used to predict success or failure in builds or code changes [3], [2]. However, most research in this area is based on identifying central modules inside a large code base which are likely to cause software failures or detecting frequently changed code that can be associated with previous failures [4]. This module-based method also results in predictions at the file or binary level of software development as opposed to a code change level and often lack the ability to provide recommendations for improved coordination other than test focus.

With the power of technical dependencies in predicting software failures, the question I investigated in this project is : "*Is it possible to identify pairs of developers whose technical dependencies in code changes statistically relate to bugs?*"

This paper explains the approach used to locate these pairs of developers in developer networks. The process utilizes code changes and the call hierarchies effected to find patterns of developer relationships in successful and failed code changes. These developer relationships can also be used to promote the idea of leveraging socio-technical congruence, a measure of coordination compared to technical dependencies amongst stakeholders, to provide coordination recommendations.

II. Technical Approach

A. Extracting Technical Networks

The basis of this approach is to create a technical network of developers based on method ownership and those methods' call hierarchies effected by code changes. These networks will provide dependency edges between contributors caused by code changes which may be identified as possible failure inducing pairings (Figure 1). To achieve this goal, developer owners of methods, method call hierarchies (technical dependencies) and code change effects on these hierarchies must be identified. This approach is described in detail by illustrating its application to mining the data in a Git repository although it can be used with any software repository.

To determine which developers own which methods at a given code change, the Git repository is queried. Git stores developers of a file per line, which was used to extrapolate a percentage of ownership given a method inside a file. If developer A has written 6/10 lines of method foo, then developer A owns 60% of said method.

A method call graph is then constructed to extract method call hierarchies in a project at a given code change. Unlike other approaches such as Bodden's et al. [5] of using byte code and whole projects, call graphs are built directly from source code files inside of a code change, which does not have the assumptions of being able to compile or have access to all project files. It is important to not require project compilation at each code change because it is an expensive operation as well as code change effects may cause the project to be unable to compile. Using source files also allowed an update to the call graph with changed files as opposed to completely rebuilding at every code change. This creates a rolling call graph which is used to show method hierarchy at each code change inside a project opposed to a static project view. As some method invocations may only be determined at run time, all possible method invocations are considered for these types of method calls while constructing the call graph.

The code change effect, if any, to the call hierarchy is now found. The Git software repository is used to determine what changes were made to any give file inside a code change. Specifically, methods modified by a code change are searched for. The call graph is then used to determine which methods

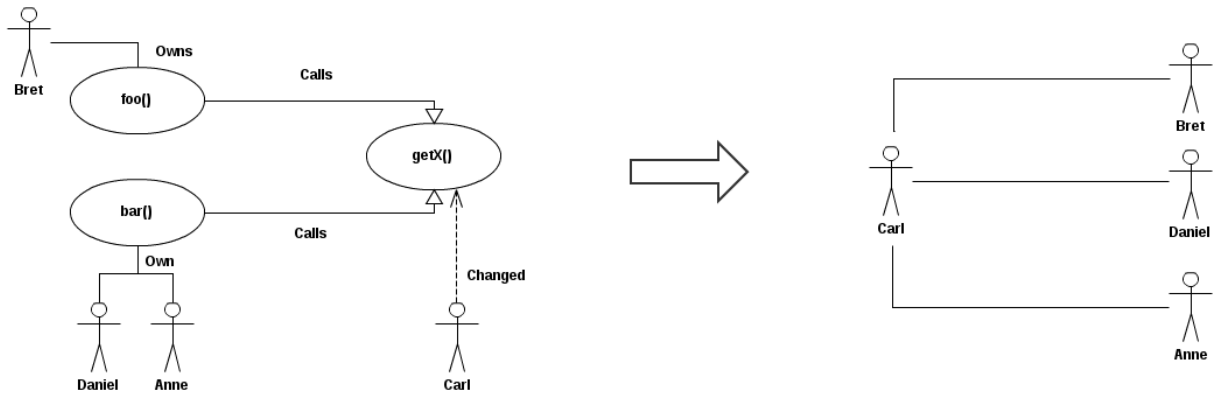

Fig. 1. A technical network for a code change. Carl has changed method getX() which is being called by Bret's method foo() as well as Daniel and Anne's method bar().

call those that have been changed, which gives the code change technical dependencies.

These procedures result in a technical network based on contributor method ownership inside a call hierarchy effected by a code change (Figure 1 left hand side). The network is then simplified by only using edges between developers, since I am only interested in discovering the failure inducing edges between developers and not the methods themselves (Figure 1 right hand side). This is the final technical network.

B. Identifying Failure Inducing Developer Pairs

To identify failure inducing developer pairs (edges) inside technical networks, edges in relation to discovered code change failures are now analysed. To determine whether a code change was a success or failure (introduce a software failure), the approach of Sliwerski et al. [6] is used. The following steps are then taken:

1) Identify all possible edges from the technical networks.
2) For each edge, count occurrences in technical networks of failed code changes.
3) For each edge, count occurrences in technical networks of successful code changes.
4) Determine if the edge is related to success or failure.

To determine an edge's relation to success or failure, the value FI (failure index) which represents the normalized chance of a code change failure in the presence of the edge, is created.

$$\mathrm{FI} = \frac{\mathrm{edge}_{failed}/\mathrm{total}_{failed}}{\mathrm{edge}_{failed}/\mathrm{total}_{failed} + \mathrm{edge}_{success}/\mathrm{total}_{success}} \quad (1)$$

Developer pairs with the highest FI value are said to be failure inducing structures inside a project. These edges are stored in Table I. A Fisher Exact Value test is also preformed on edge appearance in successful and failed code changes, and non-appearance in successful and failed code changes to only consider statistically significant edges (Table I's p-value).

III. Results

To illustrate the use of the approach, I conducted a case study of the Hibernate-ORM project, an open source Java application hosted on GitHub[1] with issue tracking performed by Jira[2].

This project was chosen because the tool created only handles Java code and it is written in Java for all internal structures and control flow and uses Git for version control. Hibernate-ORM also uses issue tracking software which is needed for determining code change success or failure [6].

In Hibernate-ORM, 27 statistically significant failure inducing developer pairs (FI value of 0.5 or higher) were found out of a total of 46 statistically significant pairs that existed over the project's lifetime. The pairings are ranked by their respective FI values (Table I).

Pair	Successful	Failed	FI	P-Value
(Daniel, Anne)	0	14	1.0000	0.0001249
(Carl, Bret)	1	12	0.9190	0.003468
(Emily, Frank)	1	9	0.8948	0.02165

TABLE I
TOP 3 FAILURE INDUCING DEVELOPER PAIRS FOUND IN THE STUDY.

IV. Conclusion and Future Work

Technical dependencies are often used to predict software failures in large software system [3], [2], [4]. This paper has presented a method for detecting failure inducing pairs of developers inside of technical networks based on code changes. These developer pairs can be used in the prediction of future bugs as well as provide coordination recommendations for developers within a project.

In future work, the adding of communication networks on a per code change basis is planned. In this, the investigation of social and technical congruence on these networks and its effects on software quality will be studied.

[1]https://github.com/
[2]http://www.atlassian.com/software/jira/overview

REFERENCES

[1] M. Acharya and B. Robinson, "Practical change impact analysis based on static program slicing for industrial software systems," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 746–755. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985898

[2] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 531–540. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368161

[3] M. Pinzger, N. Nagappan, and B. Murphy, "Can developer-module networks predict failures?" in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, ser. SIGSOFT '08/FSE-16. New York, NY, USA: ACM, 2008, pp. 2–12. [Online]. Available: http://doi.acm.org/10.1145/1453101.1453105

[4] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead, "Automatic identification of bug-introducing changes," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 81–90. [Online]. Available: http://dx.doi.org/10.1109/ASE.2006.23

[5] E. Bodden, "A high-level view of java applications," in *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '03. New York, NY, USA: ACM, 2003, pp. 384–385. [Online]. Available: http://doi.acm.org/10.1145/949344.949447

[6] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of the 2005 international workshop on Mining software repositories*, ser. MSR '05. New York, NY, USA: ACM, 2005, pp. 1–5. [Online]. Available: http://doi.acm.org/10.1145/1082983.1083147