Is NOT Talking About Your Impact on Others Harmful?

Adrian Schröter
Univeristy of Victoria, Canada
schadr@uvic.ca

Daniela Damian
University of Victoria, Canada
danielad@cs.uvic.ca

ABSTRACT

Investigating the human aspect of software development is becoming prominent in current research. Studies found that the misalignment between the social and technical dimensions of software work leads to losses in developer productivity and defects. We study the communication and technical dependencies between developers involved in software integrations and relate their misalignment to the integration success. Using data from the IBM Jazz™ project we investigate socio-technical networks in relation to build failure. Our findings reveal that the overall socio-technical misalignment could not distinguish between failed and successful builds in Jazz™, and that only a small number of developer pairs that did not communicate about their dependencies were related to build failure. The influence of these pairs on the build failure was, however, very high. We found that if any one of these pairs is present in a social network of a build, the build had at least an 84% chance to fail.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*; D.2.9 [**Software Engineering**]: Management—*Programming Teams*; K.6.1 [**Management of Computing and Information Systems**]: Project and People Management—*Systems development*; K.6.3 [**Management of Computing and Information Systems**]: Software Management—*Software development*

General Terms

Human Factors, Measurement, Management

Keywords

Social Networks, Technical Networks, Socio-Technical Networks, Builds, Failures, Socio-Technical Congruence

1. INTRODUCTION

Ensuring a smooth progress during software development is key for a project to stay within budget and on schedule. Failures introduced into the source code often require an extensive amount of time to fix, especially if caught only at integration time. Time spent on fixing bugs uses up project budget and hinders development progress.

In the last decade, research has documented multiple reasons for software failures, both on the technical and the human side of software development. On the technical side, studies showed that technical dependencies in the code (e.g. [19, 49]) are powerful predictors of error. On the human side, human and organizational factors have been found to strongly affect how these technical dependencies are handled [13, 24] and thus affected software quality [6, 23]. Coordinating to handle technical dependencies in a project becomes ever more challenging as the level of interdependencies between tasks increases [18], especially in large [11] and distributed projects [12, 24].

Complementary to studies that relate coordination to software defects (e.g. [6]), we investigate the relationship between team coordination and the success of its code integration activity. In our previous work [43] we found that properties of the social networks that represent the communication behavior around software integrations can predict integration failure. Here we seek to further our understanding on the influence of communication by studying the misalignment between technical and social dependencies.

Therefore, we conduct a finer grain investigation at the level of developer-developer dependency in these work groups. Seeking to identify to what extent the misalignment of technical and social dimensions is linked to coordination failure, we pose the question "*Is NOT talking about your impact on others really harmful?*". It is important to gain more insights into the communication at the level of developers and not the entire team. Only these insights can be used to more precisely make recommendations that enhance coordination among team members before the integration fails and thus prevent a project slow down.

We study technical dependencies and communication in IBM's Jazz™ project in relation to software builds. Jazz™ is a development environment that focuses on collaboration support and tightly integrates programming, communication, and project management (http://www.jazz.net). One insight from our study is, for example, that out of 16 builds where developers Adam and Bart changed the same file without communicating about this dependency, 13 builds failed while 3 succeeded, and that this developer pair significantly increased the risk of a build to fail.

In our methodology we construct socio-technical networks that represent communication and technical dependencies among developers, and study them in relation to build outcomes in Jazz. We first seek to validate that the lack of misalignment between the social and technical dependencies leads to failure, and then investigate these socio-technical relationships in more detail at the level of developer-developer dependency.

WOODSTOCK '97 El Paso, Texas USA

Overall, we found that the misalignment between the social and technical dimensions of work in Jazz did not differentiate between successful and failed builds. We also identified that only a small number of developer pairs that did not communicate about their dependencies are statistically related to build failure. The influence of these pairs on the build failure was, however, very high. We found that if any one of these pairs is present in a social network of a build, the build had at least an 84% chance to fail.

We use these findings to discuss the design of collaborative systems that leverage information about the failure-related developer pairs to determine which dependencies, if not communicated about, are more critical to the upcoming build.

2. RESEARCH QUESTIONS

Recent research in socio-technical congruence [7], such as the study of the effects of alignment between social and technical elements on a development project, found evidence that this alignment influences the productivity of development teams [7]. The idea behind socio-technical congruence (STC) is that if people are technically dependent (e.g. working on interdependent tasks) they should coordinate [10]. If they do not coordinate appropriately, they will slow down progress of others dependent on their tasks. The slow down occurs when developers need to wait on others' tasks to complete or to make corrections, which in turn, requires more changes [16].

Complementary to results that imply that high socio-technical congruence is related to higher productivity, we are asking whether STC has a similar relationship to an outcome that is even more tightly related to developer coordination: integration failure. If two developers work on interdependent tasks and their changes impact one another, they should presumably coordinate. Without coordination, they may introduce failures into the software that make the upcoming integration fail. Therefore, our first research question:

RQ1: Can socio-technical congruence predict integration failure?

Socio-technical congruence implies the underlying concept of socio-technical *gap* between developers. When the coordination need of two developers is not met by social interaction, they are said to form a gap. Research has generally assumed that these gaps are problematic (e.g. [16]). They are responsible for a weak socio-technical congruence and have been said to be responsible for lowering productivity [7]. Gaps may also be responsible for developers introducing bugs into the source code, for instance if a develoepr changes the behavior of a method a co-worker uses, the co-workers code might break during integration. Hence, we investigate if gaps in STC can generally be related to integration failures.

RQ2: Do socio-technical gaps lead to integration failure?

Having related socio-technical gaps to integration failure, we also seek explanations as to why they may be failure-related. For example, research suggests that different factors, such as team distribution [3] and problem domain [40], influence software quality.

The answer to our two research questions and their additional explanation enable us to devise strategies to fill socio-technical gaps. These strategies leverage the inter-personal communication that are most important for the coordination outcome and are useful for both developers and managers.

3. RELATED WORK

Since we are building on the notion of socio-technical congruence we compare technical and social dependencies among developers. To represent such dependencies among developers a network with the respective dependency is most appropriate. Thus we first review the research that used socio-technical networks in software engineering. We also search for patterns in the project history that concern socio-technical relationships that can be related to build failure, and thus follow with a review of research in pattern mining in software engineering.

3.1 Networks in Software Engineering

Three different types of developer networks have been used in software engineering research: (1) social networks that capture ongoing coordination, such as communication, (2) technical networks that use source code dependencies to code owners, and (3) socio-technical networks, which combine social and technical networks.

3.1.1 Social Networks

Software engineering research is showing an increasing interest in the human side of software development. To study developer interactions in a software project several techniques have been borrowed from social sciences. A number of studies used social network analysis techniques to investigate the relation between developer social networks and different success measures. These measures range from software quality measures over productivity to project success.

The mining software repository community described different approaches to mine social networks from software repositories, like email lists (e.g. [2]). Gonzales-Barahona et al. [29] used social networks to characterize entire projects, in contrast to Yu and Ramaswamy [46] who investigated different roles developers take on in software projects. The Huang and Liu [28] study used a similar granularity level to draw conclusions about the learning processes in projects.

Wolf et al. [44] described a methodology for how to mine social networks from repositories and a study in which they used properties of these social networks to predict the outcome of integrating the software parts within teams [43]. Meneely et al. [31] found similar evidence by extracting developer networks on file level by using code churn information. Several studies at Microsoft [3, 34] showed that different kinds of distance between people that work together on a binary determine the binaries failure proneness.

Ehrlich et al. [15] investgiated how social networks can be used to leverage knowledge in distributed teams. Backstrom et al. [1] took a more general approach and investigated the evolution of large social networks and the information they hold. Chung et al. [8] reported in recent work about behavior of individuals while performing knowledge intensive tasks. There have been a number of studies that investigated communication structures to identify good practices (e.g. [4, 25–27, 45]). In contrast to studies of the general development process Marczak studied social networks to idenify best practices for requirements management processes [30].

We use social networks to describe the communication and coordination behavior of developers that contribute to a build.

3.1.2 Technical Networks

Much research has been concerned with the technical side of software development. This technical side is often concerned with the source code. Using code ownership we can use source code to connect developers constructing a technical network. In technical networks connections between people are derived from dependencies often extracted from source code. There are two major ways in which technical networks have been built: (1) Using explicit source code dependencies and (2) using implicit source code dependencies.

Explicit code dependencies have been used to construct dependency graphs between source code entities such as classes or meth-

ods. These dependency graphs can be constructed either for a complete project or per change made to the source code [21]. For instance, Nagappan et al. used several code complexity metrics to build failure prediction models [33].

Implicit code dependencies are often not visible in the source code itself. They can be aspects that connect different source code entities [41]. Source code management systems can make aspects or other implicit relationships visible by inspecting which files have been changed together [32].

Zimmermann et al. [48, 49] used technical networks to predict the failure probability of files. Similarly Pinzger et al. [36] build networks of developers connected via code artifacts to predict failures. Previous research has also used technical networks for failure prediction by extraction complexity metrics, such as cyclomatic complexity or object oriented metrics, that are derived from technical networks [33].

In our study we construct technical networks for each build in Jazz™ to describe relations between developers derived from co-changed files.

3.1.3 Socio-Technical Networks

In recent years research has started to investigate the effect of both social and technical relations of software developers. Socio-technical networks focus on developers and connect them with two kinds of edges, social and technical. The initial idea of investigating the alignment between the communication and technical dependency between developers was formulated by Conway [10].

Expanding on this idea Cataldo et al. [7] formulated a coefficient that measures the alignment of the social and technical networks defining the term of socio-technical congruence. Moreover they observed that higher socio-technical congruence leads to higher developer productivity [7]. Others picked up on that notion and coefficient to further investigate the effect of congruence (e.g. [42]). Ducheneaut [14] investigated the evolution of social and technical relationships of open source project participants to see how those participants become a part of the comunity.

From research on socio-technical congruence emerges the question about what role socio-technical gaps play. A gap in STC exists if two developers have a coordination need that is not met by actual coordination behavior. There is a general tendency to believe that gaps are problematic for projects. Ehrlich et al. [16] investigated gaps and found that files that are changed by developers which form a gap are more prone to change.

To leverage the relation of socio-technical congruence and optimizing task completion times Sarma et al. [37] developed TESSERACT to visualize and explore socio technical networks in a project.

In this work we study ways in which knowledge about socio-technical congruence can be made actionable [39]. We study the behavior of socio-technical networks in teams of developers invovled in a software build and draw recommendations to prevent build failures.

3.2 Pattern Mining in Software Engineering

Patterns in software engineering can be either project specific or more general. A pattern describes a reoccurring event, such as people updating source code documentation [38] after a project deadline. The patterns that are found to be beneficial are then used to form templates that can be used in the future. Similarly, patterns linked to problems are then used to identify harmful events in the future.

Project Specific Patterns. Schröter et al. [40] extracted package usage information and found that using certain packages increases the chance of a file containing a failure. Neuhaus et al. [35] extended on that approach and investigated used packages in relation to vulnerabilities in Mozilla Firefox. Zimmermann er al. [47] developed eRose, an Eclipse plug-in that scans a source code repository for co-changed lines and makes recommendations for future changes.

General Patterns. Programs such as FindBugs [9] isolated anti patterns in source code, such as code smells, that have often been observed with failures. The book "Design Patterns" [20] gives general guidelines to construct better software by giving examples on working designs.

In our study we focus on simple patterns that are comprised only of a pair of developers that are connected via a technical dependency and relate to build failure.

4. METHODOLOGY

Our methodology to answer our research questions constructs socio-technical networks in order to analyze coordination behavior in software teams. We use such networks to analyze socio-technical congruence and socio-technical gaps both in relation to integration outcome in software projects. Below we first describe our methods to collect data on coordination and integration from the Jazz™ repository. We then describe the methods we use to construct and analyze social networks associated with these integrations.

4.1 Data Collection

In the following we describe what data we use to extract social networks that we extend to socio-technical network.

4.1.1 Coordination and Integrations in Jazz

We analyze builds in the Jazz™ repository to study coordination and integrations in Jazz. The Jazz™ team integrates on different levels and in different intervals, for instance on the team or project level in a nightly and weekly interval. Each build and therefor integration includes a number of changesets. A *changeset* consists of changes to one or more files in a project, comparable to a transaction in the source code management system Subversion (http://subversion.tigris.org). Furthermore, information about a build is stored in the Jazz™ repository. This includes the time the building process started, if the build succeeded and passed its test cases, and the built product. But if we refer to the build outcome we are talking about whether it could be built and if it passed all test cases.

We investigate the Jazz™ builds between April and July 2008. We choose this time interval because this represents the interval with the most complete history of builds. The Jazz™ development team deletes builds that are less important due to space reasons. Thus older builds besides builds that represent important milestones have been deleted.

From this time interval we extracted a total of 244 builds (see Table 1 for details), specifically 70 failed builds and 174 successful builds. Each build has on average 32 changesets, with a changeset touching 29 files on average. Note that there are some builds that have only one changeset associated with them. These are project wide builds that accumulate all changes made by all teams into one changeset.

4.1.2 Extracting Social Networks

We first construct *social networks* to capture the coordination behavior of developers invovled in a build. We use the information contained in the Jazz™ workitems for the construction. A *workitem* in Jazz™ is the basic unit of work. It describes a general task which can be, but is not restricted to, a bug fix or feature request. Developers coordinate about work on workitems by posting

		Successful	Failed	Total
#WorkItem	min	1	1	1
	avg	16.68	26.52	19.63
	max	111	109	111
#ChangeSet	min	1	1	1
	avg	26.71	46.27	32.57
	max	227	194	227
#Developers	min	1	1	1
	avg	19.62	28	22.16
	max	64	71	71

Table 1: Statistics on Jazz data: change sets, work items, and developers over successful (227), failed (99) and total builds (328).

comments in a discussion board style which we use as conceptualization of their coordination behavior.

We are interested in constructing a social network for each build in Jazz™. To create a social network for a given build we proceed in six steps:

1. Select the build of interest.
2. Extracting changesets that are part of the build.
3. Extracting workitems linked to the retrieved changesets.
4. Extracting developers commenting on a workitem before the build's built time.
5. Connect all developers commenting on the same workitem.

These steps take us as illustrated in Figure 1 from a build through a changeset to a workitem. From the workitem we are able to see who contributed to the workitem discussion (see Figure 2). These developers become part of the social network and share a *social edge* if they made a comment on the same workitem. Note that all links we use to get from a build to a developer are explicitly contained in the Jazz™ repository.

4.1.3 Extending to Socio-Technical Networks

To construct *socio-technical networks* we use the steps described below (see Figure 3). We essentially add technical edges to the build's already constructed social network. In our conceptualization a *technical edge* is a source code dependency between two developers. A technical dependency between two developers exists if they changed the same source code file in the build of interest.

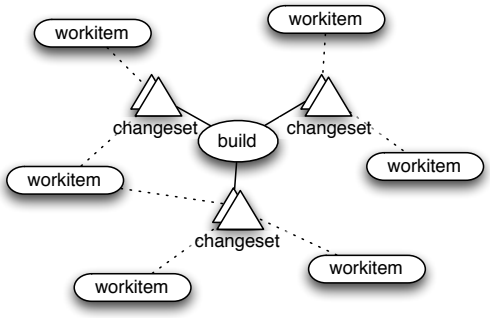


Figure 1: Linking workitems to builds using changesets.

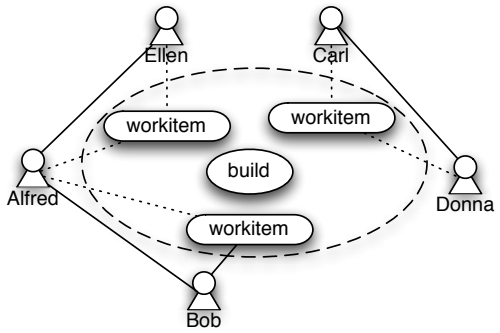


Figure 2: Social network connecting developers through workitem discussions linked to a build

1. Extract the changesets that are part the build (Figure 3(a)).
2. Determine changeset owners and add those that are not already part of the social network (Figure 3(b)).
3. Add a technical edge between changeset owners that changed the same file (see Figure 3(c)).

We thus call a network that contains both social and technical edges a *socio-technical network*. The developers in the socio-technical network that share both a technical and social edge are said to share a *socio-technical edge*.

4.2 Data Analysis

We perform two analyses on the networks we constructed, starting with comparing the socio-technical congruence index of networks of successful and failed builds, followed by a more detailed investigation of socio-technical gaps in relation to build outcome.

4.2.1 Build Outcome and STC Predictive power

To answer our first research question, we investigate the build outcome and the socio-technical congruence for all builds. Socio-technical congruence (STC) describes the match between the coordination needs in a development team as demanded by technical interdependencies and the ongoing coordination among developers.

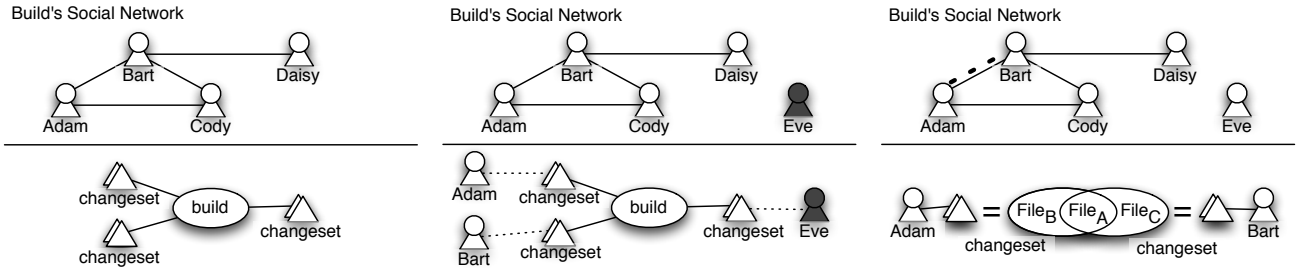
In calculating the STC index for each build's socio-technical network, we use the measure defined by Cataldo et al. [7], which is the ratio between the number of coordination needs that are met and the number of all coordination needs. The index ranges from 0 (no congruencet) to 1 (perfect congruence).

We conceptualize the coordination needs with technical edges and the ongoing coordination with social edges. Thus the socio-technical congruence index is determined by the number of socio-technical edges (number of met coordination needs) over the number of technical-edges (number of coordination needs). We compute this index for all socio-technical networks.

Since each build has an assigned build outcome (successful or failed), we split each build's socio-technical congruence index into two bins. To determine whether socio-technical congruence can be used to distinguish between successful and failed builds we perform a Wilcoxon signed rank test.

4.2.2 Analysis of Socio-Technical Gaps

Socio-technical gaps occur when coordination needs are not met by ongoing coordination. As such, we are interested in analyzing pairs of developers that share a technical edge (implying coordination need) but no social edge (implying unmet coordination need)



(a) Identify all changesets related to the social network's build.

(b) Adding changeset owners that are not part of the social network.

(c) Connect Adam and Bart with a technical edge via the co-changed File$_A$.

Figure 3: Creating a socio-technical network by adding technical dependencies to a build's social-network.

in socio-technical networks. We refer to these pairs of developers as *technical pairs*, and to those that do share a socio-technical edge (there is no gap) as *socio-technical pairs*.

To answer our second research question, we are interested in whether the technical pairs are related to build failure. Our analysis proceeds in four steps:

1. Identify all technical pairs from the socio-technical networks.
2. For each technical pair count occurrences in socio-technical networks of failed builds.
3. For each technical pair count occurrences in socio-technical networks of successful builds.
4. Determine if the pair is significantly related to success or failure.

For example, in Table 2 we illustrate the analysis of the technical pair (Adam, Bart). This pair appears in 3 successful builds and in 13 failed builds. Thus it does not appear in 171 successful builds, which is the total number of successful builds minus the number of successful builds the pair appeared in, and it is absent in 57 failed builds. A Fischer Exact Value test yields significance at a confidence level of α = .05 with a p-value of $4.273 \cdot 10^{-5}$.

Note that we adjust the p-values of the Fischer Exact Value test to account for multiple hypothesis testing using the Bonferroni adjustment. The adjustment is necessary because we deal with 961 technical pairs that need to be tested.

To enable us to discuss the findings as to whether closing socio-technical gaps are needed to avoid build failure, or which of these gaps are more important to close, we peform a two additional analyses. First we analyze whether the socio-technical pairs also appear to be build failure-related or not, by following the same steps as above for socio-technical pairs. Secondly, we prioritize the developer pairs using the coefficient p_x, which represents the normalizd likelihood of a build to fail in the presence of the specific pair:

$$p_x = \frac{\text{pair}_{failed}/\text{total}_{failed}}{\text{pair}_{failed}/\text{total}_{failed} + \text{pair}_{success}/\text{total}_{successs}}$$

The coefficient is comprised of four things: (1) pair$_{failed}$, the number of failed builds where the pair occurred; (2) total$_{failed}$, the number of failed builds; (3) pair$_{success}$, the number of successful builds where the pair occurred; (4) total$_{success}$, the number of successful builds. This coefficient is normalized with the number of failed and successful builds. A value closer to one means that the developer pair is strongly related to build failure. Additionally it describes a probability of failure likelihood that accounts for the imbalance in the data.

	successful	failed
(Adam, Bart)	3	13
¬ (Adam, Bart)	224	86
total	227	99

Table 2: Contingency table for technical pair (Adam, Bart) in relation to build success or failure

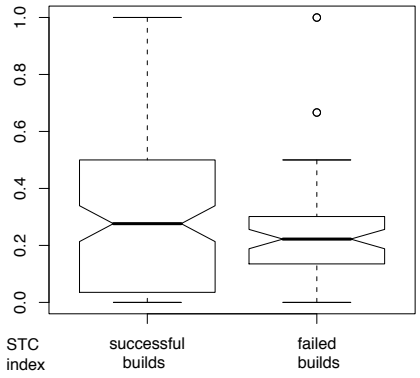


Figure 4: Boxplot of the STC index associated with successful (left) and failed (right) builds.

5. RESULTS

In this section we present our findings, starting with the investigation of the relation between socio-technical congruence and build success. Next we show the results we obtained from analyzing the socio-technical gaps.

5.1 STC and Build Outcome

We divide the socio-tecnical networks according to the respective build outcome, to see whether the socio-technical index [7] can be used to determine build success. After performing a Wilcoxon Signed Rank test with the hypothesis that the two populations are different at a confidence value of α = .05, we rejected the hypothesis (p = 0.2135). This implies that socio-technical congruence cannot differentiate between successful and failed builds and thus we cannot answer our first research question with yes.

The box plot in Figure 4 shows the two distributions of the networks according to build outcome. The medians of both categories (middle line) indicate a STC index of 0.27 and 0.22 for successful and failed builds respectively. The upper and lower boundaries

Technical Pair	#successful	#failed	p_x
Cody-Daisy	0	12	1.0000
Adam-Ina	0	8	1.0000
Adam-Kim	0	8	1.0000
Adam-Nina	0	6	1.0000
Fred-Gina	0	6	1.0000
Gina-Oliver	0	6	1.0000
Adam-Daisy	1	14	0.9720
Bart-Daisy	1	9	0.9572
Adam-Lisa	1	8	0.9521
Bart-Eve	2	11	0.9318
Adam-Bart	**3**	**13**	**0.9150**
Bart-Cody	3	13	0.9150
Adam-Eve	4	16	0.9086
Daisy-Ina	3	12	0.9086
Cody-Fred	3	10	0.8923
Bart-Herb	3	10	0.8923
Cody-Eve	5	15	0.8817
Adam-Jim	4	11	0.8723
Herb-Paul	5	12	0.8564
Mike-Rob	6	13	0.8434
Adam-Fred	6	13	0.8434

Table 3: Twenty-one *technical pairs* that are failure-related

Socio-technical Pair	#successful	#failed	p_x
Adam-Cody	0	9	1.0000
Lisa-Sarah	0	8	1.0000
Bart-Ina	0	7	1.0000
Tom-Xavier	0	6	1.0000
Vince-Xavier	0	6	1.0000
Bart-Will	3	13	0.9150
Zac-Eve	**3**	**10**	**0.8923**

Table 4: Seven *socio-technical pairs* that are failure-related

of a box represent the 75% and 25% percentile respectively and the wiskers denote the 90% and 10% percentiles. All data points outside the 10-90% interval are shown as outliers.

5.2 Socio-technical Gaps and Build Outcome

We found a total of 961 technical pairs. While only 21 pairs are significantly correlated with build failure (see Table 3), none are correlated with successful builds. Similarly, we investigated whether specific socio-technical pairs influence build outcome and found that 7 pairs significantly correlated with build failure (see Table 4) while none correlated with successful builds. Note that we use fictitious names for confidentiality reasons.

We rank the the 28 identified technical and socio-technical pairs (see Tables 3 and 4) by the coefficient p_x. This coefficient indicates the strength of relationship between the developer pair and build failure. In other words p_x represents the normalized likelihood that a build that has the respective pair will fail. Note that all p_x values are above 84%. Below we describe how to read Tables 3 and 4:

Technical Pairs. Table 3 lists all 21 technical pairs that are significant according to the Fischer Exact Value test and ranks them according to the coefficient p_x. For instance, the developer pair (Adam, Bart), appears in 13 failed builds and in 3 successful builds. This means that pair$_{failed}$ = 13 and pair$_{success}$ = 3 with total$_{failed}$= 70 and total$_{success}$= 174 result in p_x= 0.9150.

Socio-Technical Pairs. Table 4 lists all 7 socio-technical pairs are significantly related to build failure. For instance, the developer pair (Zac, Eve), appears in 10 failed builds and in 3 successful builds. This means that pair$_{failed}$ = 10 and pair$_{success}$=3 with total$_{failed}$ =70 and total$_{success}$=174 result in p_x = 0.8923.

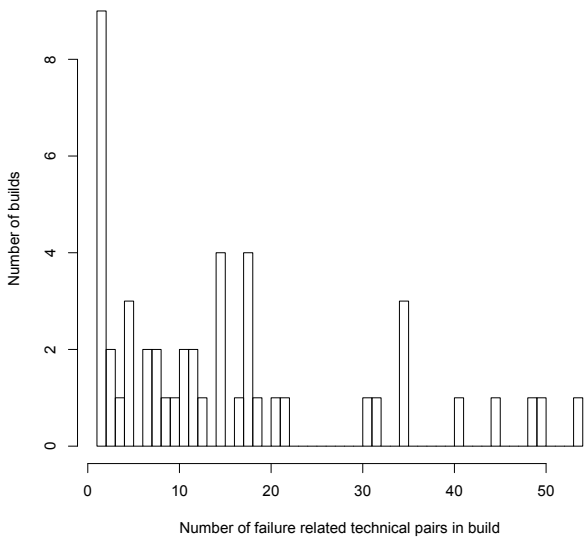


Figure 5: Histogram plotting how many builds have a certain number of failure-reated technical pairs.

The failure-related technical pairs span 55 out of the total 70 failed builds in the project. Figure 5 shows their distribution across the 55 failed builds. The histogram illustrates that there are few builds that have a large number of failure related builds, e.g. 4 with 18 or more pairs, but most builds only show a small number of pairs (22 out of 54 failed builds have 2 or less). This distribution of technical pairs indicate that the developer pairs we found did not concentrate in a small number of builds. In addition, it validates the assumption that it is worthwhile seeking insights about developer coordination in failed builds. Moreover, this enables us to explain why two thirds of the builds failed.

6. DISCUSSION

We first discuss the results that directly relate to our research questions and provide some explanations from our knowledge of the development process and developers in the Jazz™ project. Then we discuss some interesting insights about the failure related developer pairs that we found. These insights allow us to also outline a number of implications for the design of collaborative tools that can assist developers and managers in addressing the socio-technical gaps that matter to the upcoming gap.

6.1 STC and Build Outcome

Our analysis found that in Jazz, the socio-technical congruence index does not influence build outcome. Similarly, only a small number (21 out of 961, 2%) of technical pairs could be related to build failure. This indicates that those developer pairs that did not talk to each other although they shared a technical dependency were generally not harmful in the Jazz™ project. Below we give two

Developer	#Failure-related Pairs	Team	Role
Adam	9	TeamA	Contributor
Bart	5	TeamB	Contributor
Cody	4	TeamA	Contributor
Daisy	4	TeamC	Contributor
Eve	3	TeamB	Contributor
Fred	3	TeamB	Contributor
Gina	2	TeamC	Contributor
Herb	2	TeamA	TeamLead
Ina	2	TeamB	Contributor
Jim	1	TeamA	Contributor
Kim	1	TeamC	Contributor
Lisa	1	TeamD	Contributor
Mike	1	TeamE	Contributor
Nina	1	—	—
Oliver	1	TeamC	Contributor
Paul	1	TeamB	Contributor
Rob	1	TeamF	Contributor

Table 5: List of developers with number of failure related technical pairs they appear in as well as their team and role.

reasons we believe might be responsible for why our results are different from existing literature:

- *Development process.* Many processes focus on reducing the amount of unnecessary coordination between developers. For example, a process might demand the generation of component specifications. Developers working with the specified components may not explicitly coordinate their work but use these specifications. Thus, the process reduces socio-technical congruence by removing the need to talk about certain technical dependencies.

- *Developer experience.* Experienced developers are capable of assessing the impact of a technical dependency and thus the need to talk about it. If a developer knows how technical dependencies affect others work, by for example reading others code [5], then he can act without the need to talk to the respective other developer. This, in turn, lowers the socio-technical congruence because developers do not need to talk about as many technical dependencies.

The IBM Jazz™ development team mostly consists of experienced developers that were part of the Eclipse development or part of IBM Rational™. Establishing communication channels and communicating itself is easy in Jazz™, which is additionally supported by the Eclipse Way of development [17]. Similar to other mature development processes, the Eclipse Way produces specifications of software components that others can rely on, thus reducing the need to explicitly coordinate.

6.2 Investigating Failure-Related Pairs

Our analysis revealed 21 technical pairs that significantly correlate to builds failure. Although this number as compared to the overall number of technical pairs is low, their high p_x values (all above 84%) indicate a strong likelihood that their presence in a build will result in failure. Moreover, their distribution across all failed builds indicates that they were not located in only a few builds but actually spanned the majority of builds. But what makes those pairs so dangerous? We discuss below our investigation of four characteristics of these pairs:

Developer characteristics. We list the distinct developers that appear in the failure related pairs in Table 5. The table contains the number of failure related technical pairs the respective developer was part of. We see that Adam appears in 9 out of the 21 failure-related pairs. After performing a Fischer Exact Value test to determine if the developer is an influencing factor whether a pair is failure related or not we found that the developer has significant influence.

To perform this Fiqsher Exact Value test we counted in how many failure-related pairs and in how many success-related pairs the developer appears. Together with the number of total failure- and success-related pairs, we can use the Fischer Exact Value test to see if the developer makes it more likely that a pair is failure-related.

In addition we also tested whether the presence of a developer in a build already suffices to make it more likely that a build fails. The results of the corrected Fischer Exact Value tests all turned out to be significant at an $\alpha = .05$ level. This implies that there might be skill or habits that a developer exhibits, that may be harmful to the build outcome.

Developer Roles. Similar to the analysis of the individual developer we investigated individuals at a more abstract level, i.e. the role played in the project. Looking at Table 5 one would expect that the most role that is mostly associated to failure is of *contributor*. To ensure the validity of this claim we performed a Fischer Exact Value test to see if the role influences the likelihood that a pair is associated with build failure.

The Fischer Exact Value tests did not yield any significant results. The two roles we looked at, contributor and leader, are very common in softawre projects and especially the concept of role is very coarse. This means that in the Jazz™ project the explicit and assigned roles at this level of granularity do not make any difference with respect to build failure.

Membership to Teams. Typically developers work in teams rather than on their own, where each team is entrusted with developing a specific part of the software project. Table 5 also contains information on the teams to which developers belong to. We observe that TeamA spans more failure related technical pairs mostly due to the fact that Adam is part of 9 failure related technical pairs. But to be able to give a reasonable explanation why it seems that working with TeamA is more problematic, we need to investigate the teams in the future in more detail.

We also observe that most of the harmful pairs do not have developers from the same team (see Table 3 and Table 5). We found only three pairs where both developers are from the same team. While confirming previous findings that developers often communicate across teams ([15]), these findings also align with other research that suggests coordination across distance is problematic (e.g. [11, 15, 22]).

Developers Geographical location. We planned to investigate the influence of geographical location on the different pairs. But in the data we are analyzing, the pairs found in the Jazz™ project, geographical location and team coincide. Thus the analysis of location and team yield the same results.

Now that we have more insights into the developer pairs that are related to build failure, the next step would be to advise strategies to break those patterns.

6.3 Practical Implications

Our findings have several implications for the design of collaborative systems. We can incorporate the knowledge about developer pairs that tend to be failure related in a real-time recommender system. Not only do we provide the recommendations that matter to the upcoming build, we also provide incentives to motivate developers to talk about their technical dependencies.

Project historical data can be used to calculate the likelihood that the builds fails given a particular developer pair that worked on that build without talking to each other. In the case of the pair (Adam, Bart) the system may recommend that these developers should talk about their technical dependencies. Thus, we inform them that the next build will fail with a probability of 91% if they do not follow the recommendation. This probability does not only serve as mechanism to rank importance of a socio-technical gap but also as an incentive to act upon.

For management, such a recommender system can provide details about the individual developers in, and properties of, these potentially problematic developer pairs. Individual developers may be an explanation for the behavior of the pairs we found in Jazz™. This may indicate developers that are harder to work with or too busy to coordinate appropriately, prompting management to reorganize teams and workloads. This would minimize the likelihood of a build to fail, by removing the underlying cause of a pair to be failure related. Similarly, all but two developer pairs consist of developers that were part of different teams. Management may decide to investigate reasons for coordination problems that include factors such as geographical or functional distance in the project.

7. THREATS TO VALIDITY

During our study we identified two main threats. One threat covers issues that arise from the underlying data we used. The second threat deals with possible problems from the conceptualization of constructs in our study.

7.1 Data

We performed all our analysis on one set of data, the Jazz™ repository. This limits the generalizability of our findings, due to the fact that we only made the observations within one project. The project size and the project properties – incorporating open source practices such as open development and encouraging community involvement – make us believe that our findings still hold value.

Furthermore we only investigated three months of the project's lifetime. This might lead to smaller significance of our results. However, since the three months are directly before a major release of the project, this dataset contains the most viable data for our analysis. In those three months a lack of necessary coordination is the most harmful to the project.

Another threat that is inevitable in studies of software engineers is the possible lack of recorded communication. This and the possibility of people coordinating without communicating, such as reading each others source code [5], are mitigated in Jazz by its development process. In Jazz™ the development process demands that the developers coordinate using workitem discussion.

7.2 Conceptualization

Our conceptualization of the three edges we use to construct the socio-technical networks might introduce inaccuracy in our findings. First, social edges are extracted from workitem discussions. We assumed that every developer commenting on or subscribed to a work item reads all comments of that work item. This assumption might not always be correct. By manual inspection of a selected number of work items, however, we found that developers who commented on a work item are aware of the other comments, confirming our assumption.

Second, the technical edges are not problematic by themselves, but they are not complete, since there are more technical relationships between developers that can be examined. For example, two developers can be connected if one developer changes someone else's code. This however does not invalidate our findings, it just suggests that there is room for improvement and which we should address next.

Third, socio-technical edges on the other hand may suffer from the combination of social and technical edges. For example, it is not necessarily true that the discussion of two developers in a technical dependency is always about their technical dependency. In our study however, since the changes to source code files we use to extract technical dependencies are attached to workitem discussion, we are confident that they addressed the changes at least indirectly.

8. CONCLUSION AND CONSEQUENCES

Our study investigated the relationship between socio-technical congruence and integration failures. We were motivated by findings in the literature that suggested that high alignment between technical dependencies and actual coordination in a project has a positive effect on task performance.

We hypothesized that a similar relationship may be found in relation to a broader coordination outcome, i.e. integration outcome, because developers not coordinating about dependencies in their work might lead to errors remaining in the code that break the build.

We did not, however, find a strong relationship between socio-technical congruence or the socio-technical gaps and build outcome in the Jazz project. The influence of those few developer pairs with a socio-technical gap on the build failure was, however, very high. We found that if any one of these pairs was present in a social network of a build, the build had at least an 84% chance to fail. This suggests two things: (1) In general those developers who did not talk about their changes that affect others were not harmful to the integration outcome and that (2) There are a selected few developer pairs that should talk about their dependencies. The socio-technical congruence can serve as a mechanism to identify which inter-personal relationships are important to avoid breaking the upcoming build and we discussed how collaborative systems can incorporate such recommendations in real-time. As a next step we plan to extend this investigation in two directions:

Finer edges. We plan to refine all three edge types social, technical and socio-technical. For the social edges we plan on identifying who people are talking to and exactly about what. Technical edges can be refined by examining other source code relations, such as call graphs, or changes made to others' source code. To combine social and technical edges to socio-technical edges we plan to use content analysis techniques on communication to match it to the appropriate technical edge.

Different edges. In this study we focused on technical pairs and only briefly touched on socio-technical pairs. In the future we plan to extend this focus to include a detailed analysis of socio-technical pairs. It was interesting that we found that even socio-technical pairs were significantly related to failure, when one would expect that closing a socio-technical gap is the solution towards more effective coordination. Similarly, worthwhile investigating are the developer pairs that talked without sharing a technical dependency, and which indicate emerging communication in the project, and expertize

seeking behaviors that are important to effective coordination.

Finer failures. A build fails often because of a single or several failures scattered across several locations Hence we plan on investigating single bugs. This additionally enables us to investigate pre-release and post-relase failures separately.

In addition to the future research directions we plan on repeating this study on more data sets to ensure a better generalizability.

9. ACKNOWLEDGEMENTS

This project is funded by an IBM Jazz Inovation Award and a Univeristy fellowship of the University of Victoria. Thanks for continuous feedback go to the SEGAL Group members especially to Irwin Kwan, Sarbina Marczak, and Mattew Richards.

10. REFERENCES

[1] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group Formation in Large Social Networks: Membership, Growth, and Evolution. In *Proceedings of the 12th International Conference on Knowledge Discovery and Data Mining*, pages 44–54, New York, NY, USA, 2006. ACM.

[2] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan. Mining Email Social Networks. In *Proceedings of the Third International Workshop on Mining Software Repositories*, pages 137–143, New York, NY, USA, 2006. ACM.

[3] C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy. Does Distributed Development Affect Software Quality?: An Empirical Case Study of Windows Vista. *Communication of the ACM*, 52(8):85–93, 2009.

[4] C. Bird, D. Pattison, R. D'Souza, V. Filkov, and P. Devanbu. Latent Social Structure in Open Source Projects. In *Proceedings of the 16th International Symposium on Foundations of Software Engineering*, pages 24–35, New York, NY, USA, 2008. ACM.

[5] F. Bolici, J. Howison, and K. Crowston. Coordination without Discussion? Socio-Technical Congruence and Stigmergy in Free and Open Source Software Projects. In *Proceedings of the Second International Workshop on Socio-Technical Congruence*, 2009.

[6] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb. Software Dependencies, Work Dependencies, and Their Impact on Failures. *IEEE Transactions on Software Engineering*, 99(1), 2009.

[7] M. Cataldo, P. A. Wagstrom, J. D. Herbsleb, and K. M. Carley. Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools. In *Proceedings of the 20th Anniversary Conference on Computer Supported Cooperative Work*, pages 353–362, New York, NY, USA, 2006. ACM.

[8] K. S. K. Chung, L. Hossain, and J. Davis. Individual Performance in Knowledge Intensive Work through Social Networks. In *Proceedings of the 45th Conference on Computer Personnel Research*, pages 159–167, New York, NY, USA, 2007. ACM.

[9] B. Cole, D. Hakim, D. Hovemeyer, R. Lazarus, W. Pugh, and K. Stephens. Improving Your Software Using Static Analysis to Find Bugs. In *Companion to the 21st Symposium on Object-Oriented Programming Systems, Languages, and Applications*, pages 673–674, New York, NY, USA, 2006. ACM.

[10] M. E. Conway. How do Comittees Invent? *Datamation*, 14(4):28–31, 1968.

[11] B. Curtis, H. Krasner, and N. Iscoe. A Field Study of the Software Design Process for Large Systems. *Communincation of the ACM*, 31(11):1268–1287, 1988.

[12] D. Damian, L. Izquierdo, J. Singer, and I. Kwan. Awareness in the Wild: Why Communication Breakdowns Occur. In *Proceedings of the Second International Conference on Global Software Engineering*, pages 81–90, Washington, DC, USA, 2007. IEEE Computer Society.

[13] C. R. B. de Souza, D. Redmiles, L.-T. Cheng, D. Millen, and J. Patterson. Sometimes You Need to See Through Walls: A Field Study of Application Programming Interfaces. In *Proceedings of the 18th Conference on Computer Supported Cooperative Work*, pages 63–71, New York, NY, USA, 2004. ACM.

[14] N. Ducheneaut. Socialization in an Open Source Software Community: A Socio-Technical Analysis. *Computer Supported Cooperative Work*, 14(4):323–368, 2005.

[15] K. Ehrlich and K. Chang. Leveraging Expertise in Global Software Teams: Going Outside Boundaries. In *Proceedings of the First International Conference on Global Software Engineering*, pages 149–158, Washington, DC, USA, 2006. IEEE Computer Society.

[16] K. Ehrlich, M. Helander, G. Valetto, S. Davies, and C. Williams. An Analysis of Congruence Gaps and Their Effect on Distributed Software Development. In *Proceedings of the First International Workshop on Socio-Technical Congruence*, 2008.

[17] R. Frost. Jazz and the Eclipse Way of Collaboration. *IEEE Software*, 24(06):114–117, 2007.

[18] J. R. Galbraith. *Designing Complex Organizations*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1973.

[19] H. Gall, K. Hajek, and M. Jazayeri. Detection of Logical Coupling Based on Product Release History. In *Proceedings of the Fourteenth International Conference on Software Maintenance*, page 190, Washington, DC, USA, 1998. IEEE Computer Society.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, 1995.

[21] D. M. German, A. E. Hassan, and G. Robles. Change Impact Graphs: Determining the Impact of Prior Codechanges. *Information and Software Technology*, 51(10):1394–1408, 2009.

[22] R. E. Grinter, J. D. Herbsleb, and D. E. Perry. The Geography of Coordination: Dealing with Distance in R&D Work. In *Proceedings of the International Conference on Supporting Group Work*, pages 306–315, New York, NY, USA, 1999. ACM.

[23] J. Herbsleb, A. Mockus, and J. Roberts. Collaboration in Software Engineering Projects: A Theory of Coordination. In *Proceedings of 27th International Conference on Information Systems*, 2006.

[24] J. D. Herbsleb and R. E. Grinter. Splitting the Organization and Integrating the Code: Conway's Law Revisited. In *Proceedings of the 21st International Conference on Software Engineering*, pages 85–95, New York, NY, USA, 1999. ACM.

[25] D. Hinds and R. M. Lee. Social Network Structure as a Critical Success Condition for Virtual Communities. In *Proceedings of the 41st Annual Hawaii International*

Conference on System Sciences, page 323, Washington, DC, USA, 2008. IEEE Computer Society.

[26] P. Hinds and C. McGrath. Structures that Work: Social Structure, Work Structure and Coordination Ease in Geographically Distributed Teams. In *Proceedings of the 20th Anniversary Conference on Computer Supported Cooperative Work*, pages 343–352, New York, NY, USA, 2006. ACM.

[27] L. Hossain, A. Wu, and K. K. S. Chung. Actor Centrality Correlates to Project Based Coordination. In *Proceedings of the 20th Anniversary Conference on Computer Supported Cooperative Work*, pages 363–372, New York, NY, USA, 2006. ACM.

[28] S.-K. Huang and K.-m. Liu. Mining Version Histories to Verify the Learning Process of Legitimate Peripheral Participants. In *Proceedings of the Second International Workshop on Mining Software Repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[29] Luis, G. Barahona, and G. Robles. Applying Social Network Analysis to the Information in CVS Repositories. In *Proceedings of the First International Workshop on Mining Software Repositories*, 2004.

[30] S. Marczak, D. Damian, U. Stege, and A. Schröter. Information Brokers in Requirement-Dependency Social Networks. In *Proceedings of the 16th International Conference on Requirements Engineering*, volume 0, pages 53–62, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[31] A. Meneely, L. Williams, W. Snipes, and J. Osborne. Predicting Failures with Developer Networks and Social Network Analysis. In *Proceedings of the 16th International Symposium on Foundations of Software Engineering*, pages 13–23, New York, NY, USA, 2008. ACM.

[32] N. Nagappan and T. Ball. Use of Relative Code Churn Measures to Predict System Defect Density. In *Proceedings of the 27th International Conference on Software Engineering*, pages 284–292, New York, NY, USA, 2005. ACM.

[33] N. Nagappan, T. Ball, and A. Zeller. Mining Metrics to Predict Component Failures. In *Proceedings of the 28th International Conference on Software engineering*, pages 452–461, New York, NY, USA, 2006. ACM.

[34] N. Nagappan, B. Murphy, and V. Basili. The Influence of Organizational Structure on Software Quality: An Empirical Case Study. In *Proceedings of the 30th International Conference on Software Engineering*, pages 521–530, New York, NY, USA, 2008. ACM.

[35] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting Vulnerable Software Components. In *Proceedings of the 14th Conference on Computer and Communications Security*, pages 529–540, New York, NY, USA, 2007. ACM.

[36] 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*, pages 2–12, New York, NY, USA, 2008. ACM.

[37] A. Sarma, L. Maccherone, P. Wagstrom, and J. Herbsleb. Tesseract: Interactive Visual Exploration of Socio-Technical Relationships in Software Development. In *Proceedings of the 31st International Conference on Software Engineering*, pages 23–33, Washington, DC, USA, 2009. IEEE Computer Society.

[38] D. Schreck, V. Dallmeier, and T. Zimmermann. How Documentation Evolves over Time. In *IWPSE '07: Proceedings of Ninth International Workshop on Principles of Software Evolution*, pages 4–10, New York, NY, USA, 2007. ACM.

[39] A. Schröter, I. Kwan, L. D. Panjer, and D. Damian. Chat to Succeed. In *Proceedings of the First International Workshop on Recommendation Systems for Software Engineering*, pages 43–44, New York, NY, USA, 2008. ACM.

[40] A. Schröter, T. Zimmermann, and A. Zeller. Predicting Component Failures at Design Time. In *Proceedings of the Fifth International Symposium on Empirical Software Engineering*, pages 18–27, New York, NY, USA, 2006. ACM.

[41] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the 21st International Conference on Software Engineering*, pages 107–119, New York, NY, USA, 1999. ACM.

[42] G. Valetto, M. Helander, K. Ehrlich, S. Chulani, M. Wegman, and C. Williams. Using Software Repositories to Investigate Socio-technical Congruence in Development Projects. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 25, Washington, DC, USA, 2007. IEEE Computer Society.

[43] T. Wolf, A. Schröter, D. Damian, and T. Nguyen. Predicting Build Failures Using Social Network Analysis on Developer Communication. In *Proceedings of the 31st International Conference on Software Engineering*, pages 1–11, Washington, DC, USA, 2009. IEEE Computer Society.

[44] T. Wolf, A. Schröter, D. Damian, L. D. Panjer, and T. H. D. Nguyen. Mining Task-Based Social Networks to Explore Collaboration in Software Teams. *IEEE Software*, 26(1):58–66, 2009.

[45] J. Xu, Y. Gao, S. Christley, and G. Madey. A Topological Analysis of the Open Souce Software Development Community. In *Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, page 198.1, Washington, DC, USA, 2005. IEEE Computer Society.

[46] L. Yu and S. Ramaswamy. Mining CVS Repositories to Understand Open-Source Project Developer Roles. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 8, Washington, DC, USA, 2007. IEEE Computer Society.

[47] T. Zimmermann, V. Dallmeier, K. Halachev, and A. Zeller. eROSE: guiding programmers in eclipse. In *Proceedings of the 20th Conference on Object-Priented Programming, Systems, Languages, and Applications*, pages 186–187, New York, NY, USA, 2005. ACM.

[48] T. Zimmermann and N. Nagappan. Predicting Subsystem Failures using Dependency Graph Complexities. In *Proceedings of the 18th International Symposium on Software Reliability*, pages 227–236, Washington, DC, USA, 2007. IEEE Computer Society.

[49] T. Zimmermann and N. Nagappan. Predicting Defects Using Network Analysis on Dependency Graphs. In *Proceedings of the 30th International Conference on Software Engineering*, pages 531–540, New York, NY, USA, 2008. ACM.