On Socio-Technical Coordination and its Relation to Build Failure

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

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

ABSTRACT

Investigating social interactions in software development is becoming prominent in current research. The misalignment between the social and technical dimensions of software work has been linked to losses in developer productivity and defects. In a case study of coordination in the IBM Jazz™ project, we investigate the communication and technical dependencies between developers involved in software builds and relate their misalignment to the build failure. We found that historical project information about socio-technical coordination and software builds can be used in a model that predicts the quality of upcoming builds. We also identify a number of developer pairs that did not communicate about their dependencies and thus increased the likelihood of build failure. Upon this actionable knowledge developers and mangers can act to prevent build failure. If any one of these pairs is present in a builds social network, the build had at least an 74% chance to fail. This has several practical implications for the design of collaborative systems, such as the integration of recommendations about inter-personal relationships.

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

We hypothesize that with the ever growing size of software teams the lack of effective coordination is the main source of integration failures. With the ever growing complexity and sophistication of large software projects, error-free integrations are not only important but difficult to achieve. The development work that precedes integrations involves significant coordination of developers that work in teams and need to rely on the code of others and its stability. But often code is everything but stable, further contributing to developers' need to coordinate to keep up with code changes that impact their work. This problem is amplified in software builds where an entire team needs to integrate their work and on which the development of new features depends. Not only do failed builds destabilize the product [11] but they also demotivate software developers [18].

Despite their importance, keeping integrations builds error-free can be a very time consuming process. A lightweight approach that can determine whether the build contains failures before invoking the build process is thus very valuable to developers. This lightweight approach could determine a builds outcome in minutes rather than hours or days. Having a faster way to assess the quality of a build helps developers to continue working with newest builds while being aware of its quality. Previous research [15, 31] trained predictive models to assess the quality of software builds without the need of invoking large test suits. Although this research reaches a high degree of accuracy in their predictions, knowing that a build will fail does not necessarily help developers to actually prevent the build from failing. The goal of this research is to find a way to create actionable knowledge that developers can act upon to avoid integration failure.

In this paper we describe a case study of IBM's Jazz project where we leverage information about socio-technical developer coordination and software builds to identify pairs of developers that negatively influence the quality of the upcoming build. Using historical project information we construct socio-technical networks that capture information about developers with technical dependencies as well as their ongoing communication as conceptualization of their coordination. We found that using a support vector machine on a combination of this social and technical project data yields a powerful predictor of build failure. We then identify that there are certain pairs of developers that have a negative influence on the build outcome. On this knowledge developers and management can act upon to avoid future build failure.

Maintaining proper communication and awareness of work others perform is important in any kind of project. Specifically in software engineering many studies found that factors such as geographical and organizational distance have an impact on communication and even effect software quality [25]. In our study we uncover the existence of pairs of developers, that, if technically dependent in a build but not discussing their dependencies, have a negative influence on the success on builds. This actionable knowledge can be integrated in real-time recommender systems that indicate, based on project historical data, which developer pairs tend to be failure related. Developers and management can then devise strategies to prevent the failure before build time.

The paper is organized as follows: We start with formulating our research questions (Section 2) followed by putting our research into the perspective of related work (Section 3). Section 4 covers details about our case study of IBM's Jazz team to investigate socio-technical coordination in relation to builds. Then Sections 5 and 6 cover the actual analysis and their discussion. Before we concluding the paper in Section 9, we discuss some practical implications in Section 7 and the threats to validity (Section 8).

2. RESEARCH QUESTIONS

Communication is an important mechanism for coordination in software development [10,20], especially in maintaining the awareness of changes relevant to your work. Past research showed that a high coverage of technical dependencies by social relationships correlates with higher productivity of development teams (e.g. [7]). We hypothesize that a similar relation exists between this coverage and the success of software builds.

Thus socio-technical networks that capture information about the social and technical relationships between developers, should predict build failure:

RQ1 Can we use information about relations (technical and social) between developers to predict build outcome?

Although valuable, having a model that predicts build outcome is often not enough. We seek to generate actionable knowledge upon which developer can act to avoid a build from failing. Past research suggests that the absence of communication between developers that are technically dependent leads to problems, such as slow down in development [6].

We hypothesize that due to the high coordination needs the absence of this important communication also has a negative influence on build outcome. Communication problems can arise from many factors including organizational, social or technical reasons. Being able to pinpoint mismatches between technical dependencies and required communication that relate to build failure is even more important in a team's ability to devise strategies to avoid build failure. Thus, we investigate pairs of developers that share a technical dependency without talking with each other (referred to as *technical pairs*):

RQ2 Are there technical pairs that influence the build outcome?

Acknowledging that build failure can be the result of factors other than lack of developer communication, e.g. the number of changes or developers in a build [15], our analysis also studies the effect of technical pairs in the presence of such confounding variables.

3. RELATED WORK

Our study aims on integrating work investigating team collaboration and failure prediction to produce actionable knowledge upon which developer can act. Several studies bear relevance with respect to different dimensions of our work:

With respect to research on software builds: To the best of our knowledge the studies by Hassan et al. [15] and Wolf et al. [31] are the only studies that conducted research to predict build outcome. Hassan et al. [15] found that a combination of social metrics (e.g. number of authors) and technical metrics (e.g. number of code changes) derived from the source code repository yield to be best predictor. On the other hand Wolf et al. [31] solely used metrics that they derived from the social network created from discussions among developers and showed that communication structure has an influence on the build outcome.

With respect to team coordination: In order to manage changes and maintain quality, developers must coordinate. In software development, coordination is largely achieved through communicating with people who depend on the work that you do [20]. The software engineering literature is recognizing the role of communication as something that should be nurtured not eliminated and recent collaborative software development environments aim to support developers' social interactions along with artifact creation activities [26].

Ehrlich et al. [13] 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 coordination practices (e.g. [4, 16, 17, 19]). In contrast to studies of the general development process, Marczak studied social networks to identify best practices for requirements management processes [21].

Inspired by Conways Law [9], Cataldo et al. [6, 7] formulated a coefficient that measures the alignment of the social and technical networks defining the term of socio-technical congruence. They observed that higher socio-technical congruence leads to higher developer productivity [6, 7]. Others used this notion and coefficient to further investigate the effect of congruence (e.g. [30]). Prior to Cataldo et al. [6,7] proposal, Ducheneaut [12] investigated the evolution of social and technical relationships of open source project participants to see how those participants become a part of the community.

With respect to failure prediction: There have been a large number of studies looking into predicting failures. For example, Zimmermann et al. [32] used networks constructed from interdependent binaries to predict the failure probability of files. They used different metrics characterizing the relationship binaries have to each other and found that ego centric social network measures are powerful failure predictors. Previous research conducted at Microsoft used code complexity metrics, such as cyclomatic complexity or object oriented metrics, that are derived from source code. Nagappan et al. [24] found that no single source code metric was capable of being a good predictor over all studied Microsoft projects. Motivated by this study that suggested that predictions might be domain specific Schröter et al. [29] characterized domain of packages in the Eclipse project by their imports and found them to be a powerful predictor.

With respect to failures related to team coordination: More recent studies started to relate the social with the technical dimensions of software development to build predictive models. Pinzger et al. [27] successfully used social networks connecting developers via code artifacts to predict failures. Meneely et al. [22] used similar networks but excluded the code artifacts and connected the developers directly. Two studies at Microsoft looked into the geographical [2] and organizational [25] distance between people that worked on the same binary and the relation to the failure proneness of said binary. They found that the organizational distance is a very

		Successful	Failed	Total
	min	1	1	1
#work item	avg	16.68	26.52	19.63
	max	111	109	111
	min	1	1	1
#change set	avg	26.71	46.27	32.57
	max	227	194	227
	min	1	1	1
#Developers	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).

powerful predictor of failure proneness of binaries whereas the investigation of geographical distance has little to no effect. A recent study [3] combines the work of Pinzger et al. [27] and Zimmermann [32] by creating socio-technical networks that capture developer contributions and binary interdependencies. They found this combination to be a more powerful predictor that works for different software project and even prevails across multiple revisions of a project.

Despite the high predictive power of the state-of-the-art prediction models, most of them suffer from a profound shortcoming: the knowledge provided by these models is not always easily actionable. In this work we lie the foundation to the recommender system described by Schröter et al. [28]. This recommender system compares social networks derived from technical and social dependencies over successful and failed builds to recommend changing failure related pairs of developers. This way we generate knowledge that not only tells us when a build fails but immediately helps us to provide suggestions to prevent the build from failing.

4. SOCIO-TECHNICAL COORDINATION AND BUILDS IN JAZZ

Before we describe our approach to collect data and construct socio-technical networks in Jazz, we give a brief description of the Jazz™ team and project. Subsequent, we explain our approach of mining the project archives to construct socio-technical networks (Subsections 4.2 and 4.3).

4.1 Development and Builds in the Jazz Team

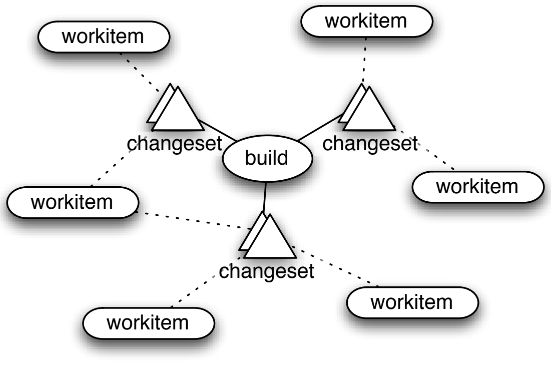


Figure 1: Linking work items to builds using change sets.

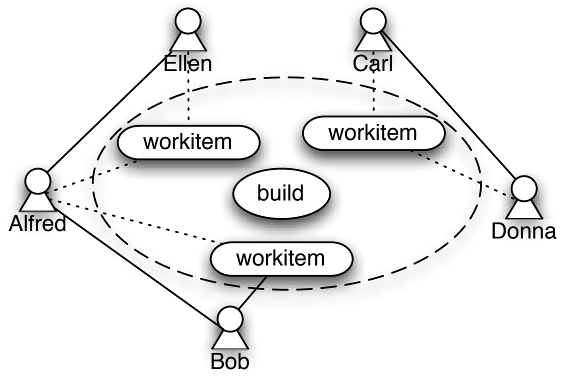


Figure 2: Social network connecting developers through discussions about work items in a build

The Jazz™ team is a large distributed team and uses the Jazz™ platform for development. The Jazz™ development involves distributed collaboration over 16 different sites located in the United States, Canada, and Europe. Seven sites are active in Jazz development and testing. There are 151 active contributors working in 47 teams at these locations, where developers can belong to multiple teams. Each team is responsible for developing a subsystem or component of Jazz™.The team size ranges from 1 to 20 and has an average of 5.7 members. The number of developers per geographical site ranges from 7 to 24 and is 14.8 in average.

The project uses the *Eclipse Way* development process [14]. It defines six-week iteration cycles, which are separated into planning, development and stabilization activities. A project management committee formulates the goals and features for each release at the beginning of the each iteration, and *work items* represent assignable and traceable tasks for each team. Furthermore, the Jazz™ team's development process demands that the developers coordinate using work item discussion.

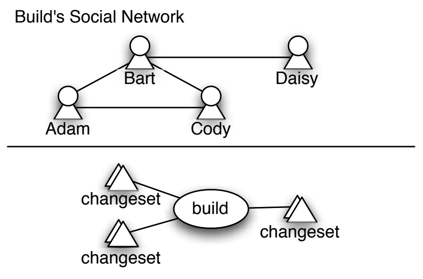
The coordination process within each iteration requires the integration of subsystems developed by individual Jazz™ teams in a major milestone build of the product. Each team owns a source code Stream for collaboration and concurrent implementation of the subsystem. A Stream is the Jazz™ equivalent to a branch of a source configuration management system, such as Subversion.

A continuous integration process takes place at team-level or project-level. In frequent intervals, each integration build (referred to a build henceforth) compiles, packages, and tests the source code of a stream. At the team-level, contributors commit code changes that are encapsulated in change sets from their own workspace to the Team Stream. The team integrations build the subsystem developed by the team. Once a team has a stable version within the Team Stream, the team publishes the change sets into the Jazz™ Project Integration Stream. At the project level, the automated Jazz™ integration builds the subsystems of all teams.

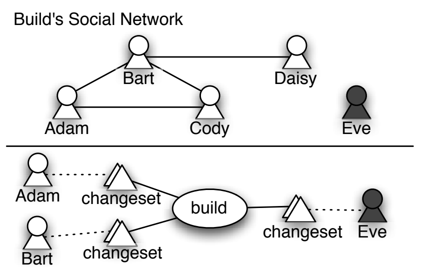
We mined the Jazz project repository between April and July 2008, and analyzed a total of 328 builds, out of which 99 were failed and 227 were successful. Table 1 contains summary statistics describing the Jazz™ repository. We report different aggregations (minimum, average and maximum) of number of work items, change sets, and developers over all builds.

4.2 Extracting Social Networks

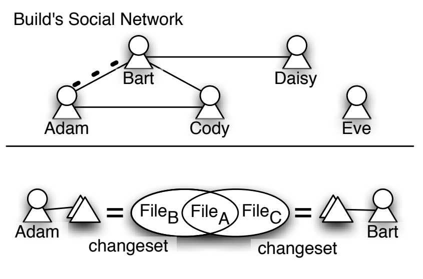
To model the communication between developers for a given build we construct *social networks*. We use the information contained in the Jazz™ work items for the construction. A *work item*



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



(b) Adding change set 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 Jazz$^{\text{TM}}$ 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 work items by posting comments in a discussion board style which we use as conceptualization of their coordination behavior. Note that the communication through board discussions is enforced by the team's development process.

We are interested in constructing a social network for each build in Jazz$^{\text{TM}}$. To create a social network for a given build we proceed in six steps:

1. Select the build of interest.
2. Extract change sets that are part of the build.
3. Extract work items linked to the retrieved change sets.
4. Extract developers commenting on a work item before the build's built time.
5. Connect all developers commenting on the same work item.

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

4.3 Constructing Socio-Technical Networks

Past research [23] has shown that dynamic dependencies have the strongest influence on the software quality. Therefore we use these dynamic dependencies 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.

1. Extract the change sets that are part the build (Figure 3(a)).
2. Determine change set owners and add those that are not already part of the social network (Figure 3(b)).
3. Add a technical edge between change set owners that did change 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*.

5. BUILD FAILURE PREDICTION

To answer our first research question, we build a predictive model that uses the constructed socio-technical networks as input to predict whether a build succeeds or fails. Since we are interested in the practical application of this model we diverge from the standard evaluation tactic and use a more practical relevant approach. After we presente the results of our prediction model we discuss their implications.

5.1 Model Evaluation

We train several prediction models, such as logistic regression, support vector machines, decision trees, and a bayesian classifiers, using features we extract from the constructed socio-technical networks. Each feature represents a pair of connected developers in the network and the type of the edge that they are connected with (i.e. social, technical or socio-technical).

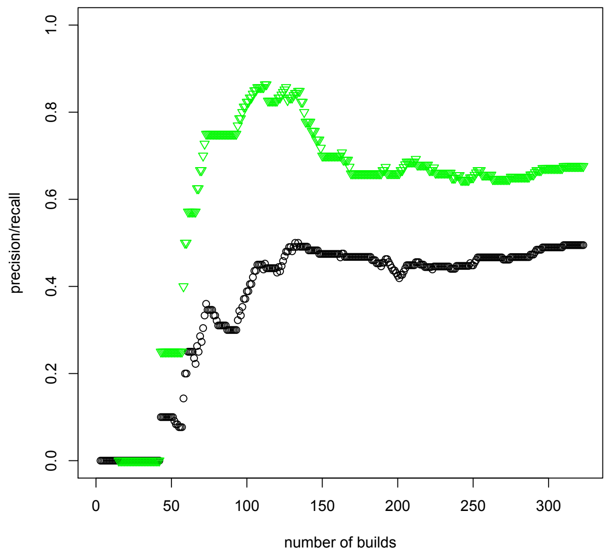
To accomplish a more practical evaluation, we order all our social networks by the time the build they were constructed from was tested. Having the networks ordered according to build time we evaluate our prediction model in the following five steps:

1. Get the first n networks and perform a principle component analysis on the extracted features.
2. Select the principal components that explain the most variance until 95% of the total variance of the training set can be explained.
3. Train the model using the principal components of the first n networks.
4. Test the model on network $n + 1$ after transforming it to the determined principal components.
5. Increase n by 1 and repeat until $n + 1$ is the size of complete data set.

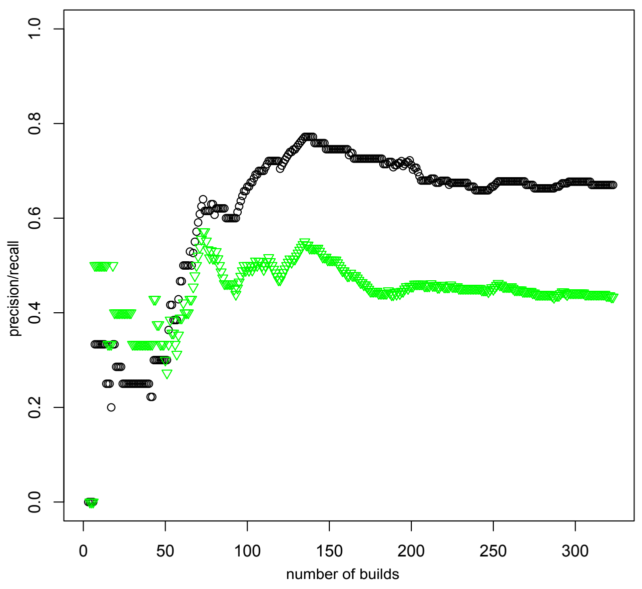
This evaluation technique is meant to simulate the actual usage of the prediction model. In software practice builds come in one by one. This means that whenever a build has been verified the model can be extended using the social network from the newest build for training. This method of evaluation is closer to the actual usage in the field than random splits or cross validation.

We used two coefficients to assess the models quality at any given time: recall (Eq. 1) and precision (Eq. 2). The recall of a model describes the percentage of how many failed builds where predicted correctly. This translates into the formula:

$$\text{recall} = \frac{\text{Correctly } \textit{as failed} \text{ Predicted Builds}}{\text{All Failed Builds}} \qquad (1)$$



(a) Evaluation results from the Support Vector Machine.



(b) Evaluation results from the Logistic Regression.

Figure 4: Plotting the precision (green downward pointing triangles) and recall (black hollow circles) of the support vector machine (left) and the logistic regression (right).

Precision on the other hand describes the percentage of how many of the *as failed* predicted builds are actually failed builds. Thus we can express precision using the following formula:

$$\text{precision} = \frac{\text{Correctly } \textit{as failed} \text{ Predicted Builds}}{\text{All } \textit{as failed} \text{ Predicted Builds}} \tag{2}$$

Both recall and precision lie in the interval from 0 to 1, with 1 being best and 0 being worst. We compute the recall and precision for each iteration by accumulating the prediction results of all previous prediction results. For example if we went through 20 iterations we create a contingency table from the predicted results for the last 20 builds. Each prediction can have one of four outcomes: (1) correctly predicted *as failed*, (2) correctly predicted as succeeded, (3) falsely predicted *as failed*, and (4) falsely predicted as succeeded. Adding those numbers till the most recent prediction enables us to compute precision and recall.

The current standard evaluation for failure prediction models in software engineering is to take the data set and generate a number of random splits (e.g. [25, 29, 32]). A random split partitions the data into a training and a testing data set, where the model is trained with the training data and then evaluated using the testing set. Other also used cross validation which creates n partitions and tests with each partition while training with the remaining $n - 1$ partitions (e.g. [31]).

Although random splits and cross validation allow for a random combination, they completely ignore the explicit order. This leads to the problem that random splits and cross validation allow features that might emerge later and should not have been available to be used for training. For example, if a developer joins the project after it started, she cannot have been present in any of the networks previous to her date of joining the project. This means that the model at first cannot use any information about her connections to others.

5.2 Results

Of the prediction models we evaluated, we present the results for the support vector machine and the logistical regression. The support vector machine produced the best results whereas the logistical regression serves as comparison to the support vector machine results as well as indicating the reliability of the regression analysis presented later in Section 6.

Figure 4 shows the recall and precision values for the suport vector machine and the logistical regression models. The green downward pointing triangles represent the precision of the model for each iteration, note that we started training each model with at least three data points. The black circles represent the recall of a model for each iteration.

Both Subfigures in Figure 4 can be divided into three sections. The first section is comprised of the first 70-80 iterations where, by a manual investigation, we observe that the support vector machine predicts almost everything to be successful in contrast to the unstable logistic regression.

The middle section is characterized by a peek efficiency between 100 and 150 iterations in both prediction models. Before that peak both models underperform, where the support vector machine suffers more than the logistic regression from a small data set.

In the last segment after 150-180 iteration the precision and recall values stabilize over both models. In contrast to the logistic regression the support vector machine obtains a higher precision and a lower recall with a slight upward trend. The support vector machine ended with a precision of .68 (median: .67) and a recall of .49 (median: .45) whereas the logistic regression obtained a precision and recall value of .43 (median: .45) and .67 (median: .67) respectively.

5.3 Discussion

Although the overall performance of the models is not yet practical due to low precision and recall, these results are interesting.

	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

On the one hand, we consider answering our first research question with yes: we can use developer pairs to predict build failure. We consider the model to be sufficient because we, as Wolf et al. [31], outperform a random guess which would have resulted in both recall and precision of less than .33, which is the percentage of failed builds. To our knowledge there have been only two studies that focused on predicting build outcomes: by Hassan et al. [15] and by Wolf et al. [31]. Our approach places itself between the results of both studies with our study being better than results obtained by Wolf et al. [31] but being worse than Hassans et al. [15] approach.

Despite of outperforming the random guess the precision of our models is of concern because it indicates the rate at which we falsely report a build to fail. With a median precision of .67, every third of the *as failed* build predicted by the support vector machine would be a false positive. Besides the low trust a developer can develop in the model, it also reports less than half of the failed builds as failed.

We provide some possible explanations here. In the first part of the Figures 4(a) and 4(b), we observed that all models perform poorly as long as they have less than 100 builds to train on. After investigating the first 100 builds we found that new developers are continually appearing in the development pairs. This means that prediction models need to make predictions without having enough knowledge to train on, thus resorting to predict the build to be the most likely outcome, to be OK.

A peak in the interval of 100-150 iterations occurred in both models for both recall and precision. Within this interval the team changed and new people joined the project and people where reallocated to work on new functionality which meant creating new dependencies and leaving old dependencies behind. This change in dependencies confused the model in a way that it did not perform as well. Both models stabilize over time with the support vector machine exhibiting a slight upward trend.

Since the goal of this research is to find a way to create actionable knowledge to avoid build failure, building a prediction model was the first step to show that developer pairs have an effect on the actual prediction. The results from the prediction models is first evidence that developer relationships have an influence on the build. Next we continue with perusing our second research question that examined the relationship between particular developer pairs (i.e. technical pairs) and build results. This will help us investigate how we might prevent builds from failing by changing the nature of developer relations in these pairs.

6. WHICH PAIRS INDUCE FAILURE?

In this section we answer our second research question, "Are there developer technical pairs that influence the build outcome?". We first explain our analysis approach followed by the results obtained and a short discussion of the results.

6.1 Analysis of Socio-Technical Gaps

The lack of communication between two developers that share a technical dependency is referred to in the literature as a socio-technical gap [30]. Because research suggests negative influence of such gaps, we are interested in analyzing pairs of developers that share a technical edge (implying coordination need) but no social edge (implying unmet coordination need) in socio-technical networks. We refer to these pairs of developers as *technical pairs* (there is a gap), 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 analyze the technical pairs in relation 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 224 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 86 failed builds. A Fischer Exact Value test yields significance at a confidence level of $\alpha = .05$ 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 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 normalized likelihood of a build to fail in the presence of the specific pair:

$$p_x = \frac{\text{pair}_{failed}/\text{total}_{failed}}{\text{pair}_{failed}/\text{total}_{failed} + \text{pair}_{success}/\text{total}_{successs}} \quad (3)$$

The coefficient is comprised of four counts: (1) pair_{failed}, the number of failed builds where the pair occurred; (2) total_{failed}, the number of failed builds; (3) $\text{pair}_{success}$, the number of successful builds where the pair occurred; (4) $\text{total}_{success}$, the number of successful builds. A value closer to one means that the developer pair is strongly related to build failure.

6.2 Results

We found a total of 2872 developer pairs in all the constructed socio-technical networks, out of which 961 were technical pairs. We choose to present the twenty that are most frequent across failed builds.

We rank the failure relating *technical* pairs (see Tables 3(a)) by the coefficient p_x. This coefficient indicates the strength of relationship between the developer pair and build failure. For instance, the developer pair (Adam, Bart), appears in 13 failed builds and in 3 successful builds. This means that pair_{failed} = 13 and $\text{pair}_{success}$ = 3 with total_{failed}= 99 and $\text{total}_{success}$= 227 result in p_x= 0.9016. Besides that we report the number of successful builds the pair was

(a) Twenty most frequent *technical pairs* that are failure-related.

Pair	#successful	#failed	p_x
(Cody, Daisy)	0	12	1
(Adam, Daisy)	1	14	0.9697
(Bart, Eve)	2	11	0.9265
(Adam, Bart)	3	13	0.9085
(Bart, Cody)	3	13	0.9085
(Adam, Eve)	4	16	0.9016
(Daisy, Ina)	3	12	0.9016
(Cody, Fred)	3	10	0.8843
(Bart, Herb)	3	10	0.8843
(Cody, Eve)	5	15	0.8730
(Adam, Jim)	4	11	0.8631
(Herb, Paul)	5	12	0.8462
(Cody, Fred)	5	11	0.8345
(Mike, Rob)	6	13	0.8324
(Adam, Fred)	6	13	0.8324
(Daisy, Fred)	8	13	0.7884
(Gill, Eve)	7	10	0.7661
(Daisy, Ina)	7	10	0.7661
(Fred, Ina)	8	10	0.7413
(Herb, Eve)	8	10	0.7413

(b) The twenty corresponding *socio-technical pairs*, which are not statistically related to failed builds.

Pair	#successful	#failed	p_x
(Cody, Daisy)	—	—	—
(Adam, Daisy)	—	—	—
(Bart, Eve)	1	4	0.9016
(Adam, Bart)	—	—	—
(Bart, Cody)	—	—	—
(Adam, Eve)	—	—	—
(Daisy, Ina)	—	—	—
(Cody, Fred)	1	0	0
(Bart, Herb)	1	2	0.8209
(Cody, Eve)	0	3	1
(Adam, Jim)	0	1	1
(Herb, Paul)	1	0	0
(Cody, Fred)	—	—	—
(Mike, Rob)	—	—	—
(Adam, Fred)	—	—	—
(Daisy, Fred)	—	—	—
(Gill, Eve)	—	—	—
(Daisy, Ina)	1	0	0
(Fred, Ina)	0	2	1
(Herb, Eve)	—	—	—

Table 4: The 20 most frequent statistically failure related technical pairs and the corresponding socio-technical pairs.

observed with (#successful) as well as the number of failed builds the pairs was observed with (#failed). The p_x values are all above 0.74, implying that the likelihood of failure is at least 74% in all builds in which these developers pairs are involved.

We then checked for the 120 pairs whether the corresponding *socio-technical* pairs are related to failure. Only 23 of the 120 technical pairs had an existing corresponding socio-technical pair of which none were statistically related to build failure. In Table 3(b) we show the socio-technical pairs that match the 20 technical pairs shown in Table 3(a) as well as the same information as in Table 3(a). If the corresponding socio-technical pair existed we computed the same statistics as for the technical pairs, but for those that existed we could not find statistical significance. Note that we use fictitious names for confidentiality reasons.

The failure-related technical pairs span 48 out of the total 99 failed builds in the project. Figure 5 shows their distribution across the 48 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 (15 out of 48 failed builds have 4 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.

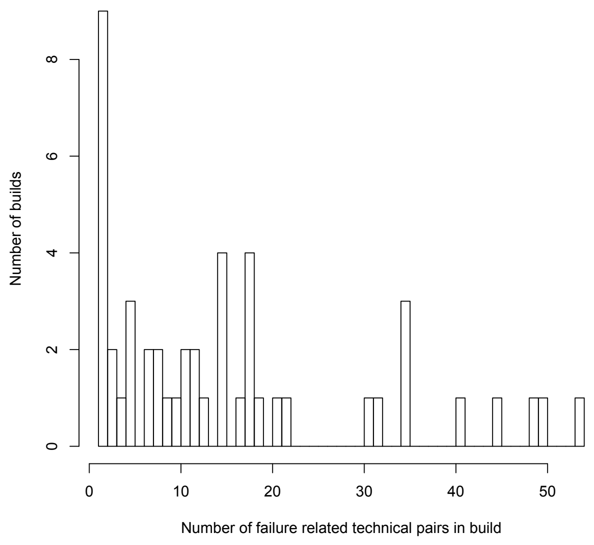


Figure 5: Histogram plotting how many builds have a certain number of failure-reated technical pairs.

6.3 Discussion

These results show that there is a strong relationship between certain technical developer pairs and increased likelihood of a build failure. Out of the total of 120 technical pairs that increase the likelihood of a build to fail, only 23 had an existing corresponding socio-technical pair. Of these, none were statistically related to build failure. This means that 97 pairs of developers that had a technical dependency did not communicate with each other and consequently increased the likelihood of a build failure. Our results not only corroborate past findings [6, 7] that socio-technical gaps have a negative effect in software development. More importantly, they indicate that the analysis presented in this paper is able to identify the specific socio-technical gaps, namely the actual developer pairs where the gaps occur and that increase the likelihood of build failure.

Although not a goal of this paper, we sought possible explanations for the socio-technical gaps in this project. A preliminary analysis of developers membership to teams shows that most of the technical pairs related to build failure consist of developer belonging to different teams. Naggappan et al. [25] found that using the organizational distance between people predicts failures.

Feature	Coefficient	p-value	
(Intercept)	7.897e+74	<2e-16	***
(Cody, Daisy)	-5.669e+75	<2e-16	***
(Adam, Daisy)	-9.846e+75	<2e-16	***
(Bart, Eve)	-1.258e+75	<2e-16	***
(Adam, Bart)	-1.605e+76	<2e-16	***
(Bart, Cody)	-3.419e+76	<2e-16	***
(Adam, Eve)	-2.610e+76	<2e-16	***
(Daisy, Ina)	-8.105e+74	<2e-16	***
(Cody, Fred)	-5.348e+76	<2e-16	***
(Bart, Herb)	-2.977e+76	<2e-16	***
(Cody, Eve)	-2.315e+76	<2e-16	***
(Adam, Jim)	-2.724e+76	<2e-16	***
(Herb, Paul)	-1.636e+76	<2e-16	***
(Cody, Fred)	-1.645e+74	<2e-16	***
(Mike, Rob)	-1.327e+75	<2e-16	***
(Adam, Fred)	-5.250e+76	<2e-16	***
(Daisy, Fred)	-2.455e+75	<2e-16	***
(Gill, Eve)	-7.162e+75	<2e-16	***
(Daisy, Ina)	-5.325e+74	<2e-16	***
(Fred, Ina)	-2.777e+75	<2e-16	***
(Herb, Eve)	-1.799e+75	<2e-16	***
#Change Sets per Build	6.480e+60	<2e-16	***
#Files changed per Build	-4.530e+60	<2e-16	***
#Developers contributed per Build	3.386e+61	<2e-16	***
#Work Items per Build	-3.690e+61	<2e-16	***

Table 5: Logistic regression only showing the technical pairs from Table 3(a), the intercept, and the confounding variables, the model reaches an AIC of 7006 with all shown features being significant at $\alpha = 0.001$ level (indicated by *).**

They reasoned that this is due to the lack of awareness what people separated by organizational distance work on. Although the Jazz team strongly emphasizes communication regardless of team boundaries, it still seems that organizational distance has an influence on its communication behavior.

Further, although the analysis of technical pairs in relation to build failures showed that they have a negative influence on the build result, it does not take into account possible confounding variables. The question is if there developer pairs still effect the build outcome in the presence of confounding variables, such as number of developers involved in a build, number of work items related to a build, number of change sets per build, and number of files changed. We tested this using a logistical regression model including both developer pairs and confounding variables.

Overall the logistic regression confirms the effect of the developer pairs even in the presence of confounding variables, such as number of files changed and numbers of developers (AIC of 7006). Table 5 shows an excerpt from the complete logistical regression which shows that the twenty pairs previously identified are still significant under the influence of the four confounding variables. Because we have 2872 developer pairs, space constraints prevent us from reporting the entire regression model. We chose to report on the coefficients for the twenty pairs that we reported in Table 5 as well as the coefficients for the four control features.

All features shown in Table 5 are significant at the $\alpha = .001$ level (indicated by the p-value and ***). We checked for all the other technical pairs that were reported significant using the approach described in the previous section and found that they are all significant in the regression model as well. Moreover, the four control variables of number of developers per build, number of work items per build, number of change sets per build, and number of files changed per build are also significant.

Since we model failed builds with 0 and successful builds with 1, a negative coefficient means that the feature increases the chances of a failure. All pairs reported in Table 5 and the technical pairs that we identified to be related to build failure have a negative coefficient.

In conclusion, we can answer our second research question with yes: there are certain technical pairs that are significantly related to build failure, and this happens even in the presence of possibly confounding variables.

7. PRACTICAL IMPLICATIONS

Our findings have several implications for the design of collaborative systems. By automating the analyses presented here 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.

Such a recommender system can use project historical data to calculate the likelihood that an upcoming build fails given a particular developer pair that worked on that build without communicating to each other. In the case of the pair (Adam, Bart) the system may recommend that these developers should communicate about their technical dependencies, as there would be a probability of 91% of failure of the next build should they not follow the recommendation. This probability of failure serves as mechanism to rank importance of a socio-technical gap and more importantly 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^{TM}. 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, as another example from our study, most developer pairs consisted of developers that were part of different teams. In such situations management may decide to investigate reasons for coordination problems that include factors such as geographical or functional distance in the project.

8. THREATS TO VALIDITY

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

8.1 Data

We performed all our analysis on one set of data, the Jazz^{TM} 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. The Jazz^{TM} team's development process demands that the developers coordinate using workitem discussion.

8.2 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, 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.

9. CONCLUSION

Our study investigated the relationship between pairs of developers that share a technical dependency but do not communicate and build 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.

Our case study of coordination in the Jazz project indicates that historical project information about socio-technical coordination and software builds can be used in a model that predicts the quality of upcoming builds. We also found that the influence of developer pairs with a socio-technical gap on the build failure was very high. This means that if any one of these pairs was present in a social network of a build, the build had at least an 74% chance to fail. Thus, we add to the literature on failure prediction with a method that provides actionable knowledge to avoid failures.

Besides the possible of integrating our findings into a real-time recommender system that indicates the specific developer pairs that should talk about their dependencies, our results add to the research in collaborative software engineering. Not only do we add to the evidence about the role of effective communication in the development of high quality software, but our findings indicate that measures of socio-technical congruence can serve as a mechanism to identify which inter-personal relationships are important to avoid breaking software integrations. We plan to extend this research in a number of directions:

More specific information in socio-technical networks. We plan to study socio-technical coordination at a finer level by refining all three edge types social, technical and socio-technical as well as the information about developers. 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. We also plan on also investigating the developers that are part of the social networks more closely by incorporating developer characteristics, such as experience, geographical location, role or team allocation.

Additional edges that indicate emergent coordination. 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 and of those developer pairs that communicated without sharing a technical dependency, and which indicate emerging communication in the project, and expertize seeking behaviors that are important to effective coordination.

Different types of 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.

10. ACKNOWLEDGEMENTS

This project is funded by an IBM Jazz Innovation 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. And special thanks goes to the Jazz^{TM} development team as well as the IBM research for they ongoing support.

11. 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, 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.

[3] C. Bird, N. Nagappan, H. Gall, B. Murphy, and P. Devanbu. Putting It All Together: Using Socio-technical Networks to Predict Failures. In *Proceedings of the 20th International Symposium on Software Reliability Engineering*, pages 109–119, Washington, DC, USA, 2009. IEEE Computer Society.

[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, J. D. Herbsleb, and K. M. Carley. Socio-Technical Congruence: A Framework for Assessing the Impact of Technical and Work Dependencies on Software Development Productivity. In *Proceedings of the Second International Symposium on Empirical Software Engineering and Measurement*, pages 2–11, New York, NY, USA, 2008. ACM.

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

[10] 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.

[11] M. A. Cusumano and R. W. Selby. How microsoft builds software. *Communication of the ACM*, 40(6):53–61, 1997.

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

[13] 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.

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

[15] A. E. Hassan and K. Zhang. Using Decision Trees to Predict the Certification Result of a Build. In *Proceedings of the 21st International Conference on Automated Software Engineering*, pages 189–198, Washington, DC, USA, 2006. IEEE Computer Society.

[16] 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.

[17] 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.

[18] J. Holck and N. Jørgensen. Continuous integration and quality assurance: A case study of two open source projects. *Australasian Journal of Information Systems*, pages 40–53, 2003/2004.

[19] 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.

[20] R. Kraut and L. Streeter. Coordination in software development. *Communications of the ACM*, 38(3):69–81, March 1995.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] K. Nakakoji, Y. Ye, and Y. Yamamoto. Supporting Expertise Communication in Developer-Centered Collaborative Software Development Environments. In A. Frinkelstein, J. Grundy, A. van der Hoek, I. Mistrik, and J. Whitehead, editors, *Collaborative Software Enginnering*, chapter 11. Springer-Verlag, 2010.

[27] M. Pinzger, N. Nagappan, and B. Murphy. Can Developer-Module Networks Predict Failures? In *Proceedings of the 16th International Symposium on Foundations of Software Engineering*, pages 2–12, New York, NY, USA, 2008. ACM.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.