~~Is NOT Talking About Your Impact on Others Harmful?~~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 the human aspect of~~ Investigating social interactions in software development is becoming prominent in current research. ~~Studies found that the~~ The misalignment between the social and technical dimensions of software work ~~leads~~ has been linked to losses in developer productivity and defects. ~~We go a step further and investigate if knowledge about this misalignment can be used to create a recommender system leveraging project historical data. Using data from~~ In a case study of coordination in the IBM Jazz™ project~~we investigate socio-technical networks in relation to~~ , we investigate the communication and technical dependencies between developers involved in software builds and relate their misalignment to the build failure. ~~Our findings reveal that overall~~ We found that historical project information about socio-technical ~~misalignment could not distinguish between failed and successful builds , and that~~ 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 ~~were related to~~ and thus increased the likelihood of build failure. Upon this actionable knowledge developers and mangers can act to prevent build failure. ~~We found that if~~ If any one of these pairs is present in a ~~social network of a build~~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

~~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.) 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 and thus affected software quality . Coordinating to handle technical dependencies in a project becomes ever more challenging as the level of interdependencies between tasks increases , especially in large and distributed projects~~ 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.

~~Complementary to studies that relate coordination to software defects (e.g.), we investigate the relationship between team coordination and the success of its code integration activity. But we want to take it one step further by automatically create recommendations from project history contained within software archives, that prevent such failure. In our previous work 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 and leverage the influence of developer communication by studying the misalignment between technical and social dependencies~~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 [**?**] but they also demotivate software developers [**?**].

Therefore, we conduct a finer grain investigation at the level of developer-developer dependency in these work groups. We seek to synthesize the relation between misalignment of developer socio-technical coordination and coordination failure into a recommender system. Thus, we pose the question "*Is NOT talking about your impact on others and awareness*" 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 generatereal-time recommendations that enhance coordination among team members before the integration fails and thus prevent a project slow downDespite 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 [**?**, **?**] 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.

We study technical dependencies and communication in In this paper we describe a case study of 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. This can be leveraged into a real-time recommender system in the Jazz environment that both development and management teams can use to avoid buildfailures.

In our methodology 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 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. 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.

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 thatif any one of these pairs is present in a social network of a build , the build had at least an 74chance to fail.

We use these findings to discuss the design of a recommender system that leverages information about the failure-related developer pairs to determine which dependencies, if not communicated about, are more critical to the upcoming build Maintaining proper communication 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 [**?**]. 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.

2. RESEARCH QUESTIONS

Recent research in 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 congruence , 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 . The idea behind socio-technical congruence (STC)is that if people are technically dependent (e.g., working on interdependent tasks) they should coordinate . 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 coordination in relation to builds. Then Sections 5 and 6 cover the actual analysis and their discussion. Before we concluding the paper in Section 10, we discuss some practical implications in Section 8 and the threats to validity (Section 9).

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:

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

2. RESEARCH QUESTIONS

Communication is an important mechanism for coordination in software development [**?**,**?**], 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.). They are responsible for a weak socio-technical congruence and have been said to be responsible for lowering productivity . 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. [?]). We hypothesize that a similar relation exists between this coverage and the success of software builds.

Do Thus socio-technical gaps lead to integration failure? networks that capture information about the social and technical relationships between developers, should predict build failure:

Having related socio-technical gaps to integration failure, we also seek explanations as to why they may be failure-related. For example,

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 different factorsthe absence of communication between developers that are technically dependent leads to problems, such as team distribution and problem domain , influence software qualityslow down in development [?].

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. 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?

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. Thuswe 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. 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 [?], our analysis also studies the effect of technical pairs in the presence of such confounding variables.

2.1 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.

2.0.1 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.

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:

The mining software repository community described different approaches to mine social networks from software repositories, like email lists *With respect to research on software builds:* To the best of our knowledge the studies by Hassan et al. [?] and Wolf et al. [?] are the only studies that conducted research to predict build outcome. Hassan et al. [?] found that a combination of social metrics (e.g.) . Gonzales-Barahona et al. used social networks to characterize entire projects, in contrast to Yu and Ramaswamy who investigated different roles developers take on in software projects. The Huang and Liu study used a similar granularity level to draw conclusions about the learning processes in projects.

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. 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 . Meneely et al. found similar evidence by extracting developer networks on file level by using code churn information. Several studies at Microsoft showed that different kinds of distance between people that work together on a binary determine the binaries failure proneness [?] 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 [?]. 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 [?].

Ehrlich et al. [?] investgiated how social networks can be used to leverage knowledge in distributed teams. Backstrom et al. [?] took a more general approach and investigated the evolution of large social networks and the information they hold. Chung et al. [?] 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. [?, ?, ?, ?]). In contrast to studies of the general development process, Marczak studied social networks to idenify identify best practices for requirements management processes [?].

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

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

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

Zimmermann et al. used technical networks to predict the failure probability of files. Similarly Pinzger et al. 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 .

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

3.0.1 *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 Inspired by Conways Law [**?**].

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

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 *With respect to failure prediction:* There have been a large number of studies looking into predicting failures. For example, Zimmermann et al. 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. 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 . We study the behavior of socio-technical networks in teams of developers invovled in a software build and draw recommendations to prevent build failures.

3.1 Pattern Mining in Software Engineering

Patterns in software engineering can be either project specific or more general. A pattern describes a reoccurring event [**?**] 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 people updating source codedocumentation 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. cyclomatic complexity or object oriented metrics, that are derived from source code. Nagappan et al. [**?**] 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. [**?**] extracted package usage information and found that using certain packages increases the chance of a file containing a failure. Neuhaus 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. extended on that approach and investigated used packages in relation to vulnerabilities in Mozilla Firefox. Zimmermann er [**?**] successfully used social networks connecting developers via code artifacts to predict failures. Meneely et al. developed eRose, an Eclipse plug-in that scans a source code repository for co-changed lines and makes recommendations for future changes. [**?**] used similar networks but excluded the code artifacts and connected the developers directly. Two studies at Microsoft looked into the geographical [**?**] and organizational [?] 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 powerful predictor of failure proneness of binaries whereas the investigation of geographical distance has little to no effect. A recent study [**?**] combines the work of Pinzger et al. [?] and Zimmermann [?] 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.

General Patterns. Programs such as FindBugs isolated anti patterns in source code, such as code smells, that have often been observed with failures. The book "Design Patterns" 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 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. [**?**]. 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. 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

		Successful	Failed	Total
#~~WorkItem~~ work item	min	1	1	1
	avg	16.68	26.52	19.63
	max	111	109	111
#~~ChangeSet~~ change set	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).

~~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.~~

3.1 ~~Data Collection~~

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

3.0.1 ~~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.~~

3.0.1 ~~Extracting Social Networks~~

~~We first construct *social networks* to capture the coordination behavior of developers invovled in a build. We~~

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

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 [**?**]. 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

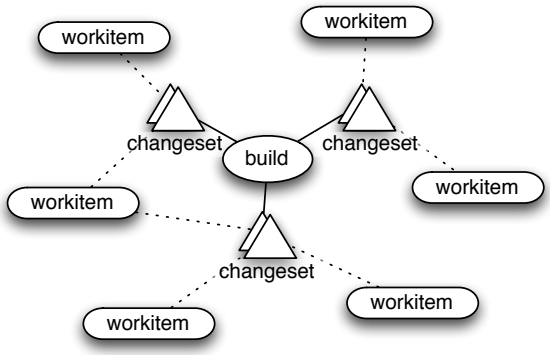
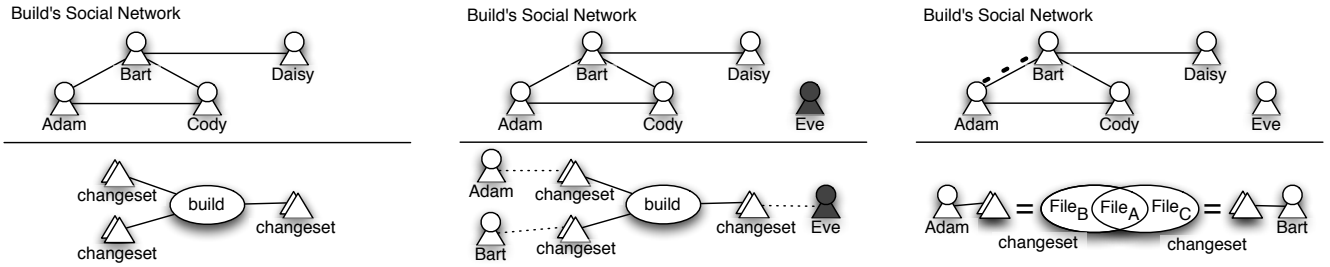


Figure 1: Linking ~~workitems~~ work items to builds using ~~changesets~~change sets.



(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.

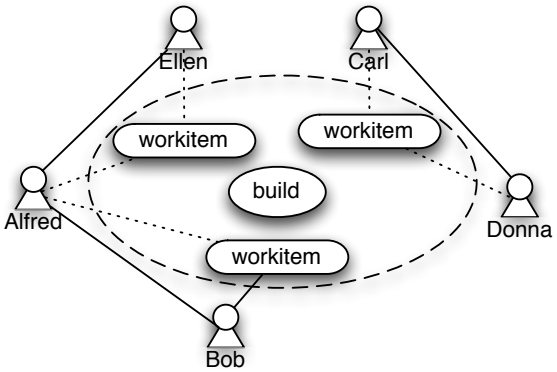


Figure 2: Social network connecting developers through ~~workitem~~ discussions ~~linked to~~ about work items in a build

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™ ~~workitems~~ work items for the construction. A ~~*workitem*~~*work item* 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~~ 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™. To create a social network for a given build we proceed in six steps:

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

These steps take us as illustrated in Figure ~~??~~ 1 from a build through a ~~changeset to a workitem~~change set to a work item. From the ~~workitem~~ work item we are able to see who contributed to the ~~workitem discussion (see~~ 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 ~~workitem~~work item. Note that all links we use to get from a build to a developer are explicitly contained in the Jazz™ repository.

4.2.1 ~~Extending to Socio-Technical Networks~~

~~To~~

4.3 Constructing Socio-Technical Networks

Past research [?] 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 ~~changesets~~ change sets that are part the build (Figure 3(a)).
2. Determine ~~changeset~~ change set 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~~ change set owners that did change the same file (see Figure 3(c)).

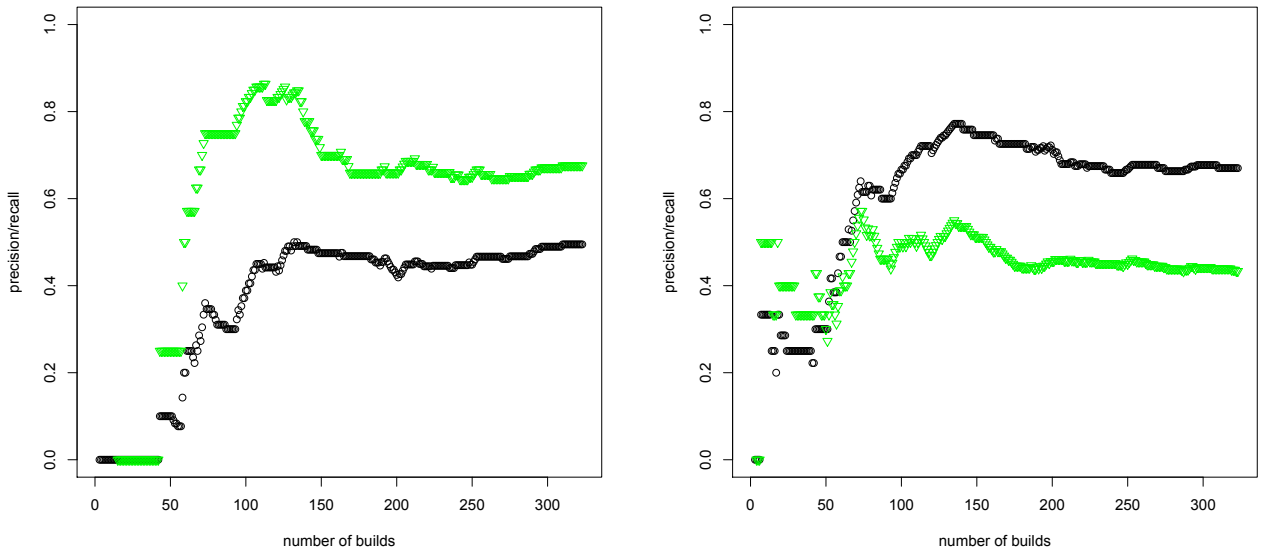
We thus call a network that contains both social and technical edges a *socio-technical network*. The developers in the socio-technical network that share both a technical and social edge are said to share a *socio-technical edge*.

4.4 ~~Data Analysis~~

~~We perform two analyses on the networks we constructed, starting with comparing the~~

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



(a) Evaluation results from the Support Vector Machine.

(b) Evaluation results from the Logistic Regression.

Figure 4: ~~Creating a socio-technical network by adding technical dependencies to a build's social-network~~**Plotting the precision (green downward pointing triangles) and recall (black hollow circles) of the support vector machine (left) and the logistic regression (right).**

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 ~~congruence index of networks of successful and failed builds , followed by a more detailed investigation of~~ 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~~gaps in relation to build outcome.~~).

5.1.1 ~~*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.~~ 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:

~~In calculating the STC index for each build's socio-technical network, we use the measure defined by Cataldo et al. , 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)~~

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:

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

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:

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

Both recall and precision lie in the interval from 0 to ~~1 (perfect congruence)~~ 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. [?, ?, ?]). 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. [?]).

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.

~~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.~~

5.3 Discussion

	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

Although the overall performance of the models is not yet practical due to low precision and recall, these results are interesting. 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. [?], 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. [?] and by Wolf et al. [?]. Our approach places itself between the results of both studies with our study being better than results obtained by Wolf et al. [?] but being worse than Hassans et al. [?] approach.

~~Since each build has an assigned buildoutcome (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~~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?".

(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.

We first explain our analysis approach followed by the results obtained and a short discussion of the results.

6.0.1 ~~Analysis of Socio-Technical Gaps~~

~~Socio-technical gaps occur when coordination needs are not met by ongoing coordination. As such~~

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 [?]. 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 ~~are interested in whether~~ analyze the technical pairs ~~are related~~ 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 ~~171~~ 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 ~~57~~ 86 failed builds. A Fischer Exact Value test yields significance at a confidence level of ~~$\alpha = .05$~~ $\alpha = .05$ 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 ~~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~~ normalized likelihood of a build to fail in the presence of the specific pair:

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

(4)

The coefficient is comprised of four ~~things~~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. ~~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 86total 22799Contingency table for technical pair (Adam, Bart) in relation to build success or failure~~

7. ~~RESULTS~~

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 *).**

In this section we present our findings, starting with the investigation of the relation between

6.1 Results

We found a total of 2872 developer pairs in all the constructed socio-technical congruence and build success. Next we show the results we obtained from analyzing the socio-technical gaps.

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

6.2 STC and Build Outcome

We divide the socio-tecnical networksaccording to the respective build outcome, to see whether the socio-technical index 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 $\alpha = .05$, 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 ?? 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 of a box represent the 75and 25percentile respectively and the wiskers denote the 90and 10percentiles. All data points outside the 10-90interval are shown as outliers.

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

6.2 Socio-technical Gaps and Build Outcome

We found a total of networks, out of which 961 were technical pairs. While only 120 pairs are significantly correlated with build failure (see Table 3(a)), none are correlated with successful builds. Similarly, we investigated whether corresponding socio-technical pairs influence build outcome and found that none significantly correlated with build failure (see Table 3(b)). Note that we use fictitious names for confidentiality reasons. We choose to present the twenty that are most frequent across failed builds.

Histogram plotting how many builds have a certain number of failure reated technical pairs.

We rank the the identified technical failure relating *technical* pairs (see Tables 3(a)for the twenty most frequent technical pairs that are related to build failure) 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 74. Below we describe how to read Tables 3(a) and 3(b):

Table 3(a) 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$_{failed}$= 99 and total$_{success}$= 227 result in p_x= 0.9085. 0.9016. Besides that we report the number of successful builds the pair was 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. Moreover, this enables us to explain why two thirds of the builds failed.

7. 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 failurerelated 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

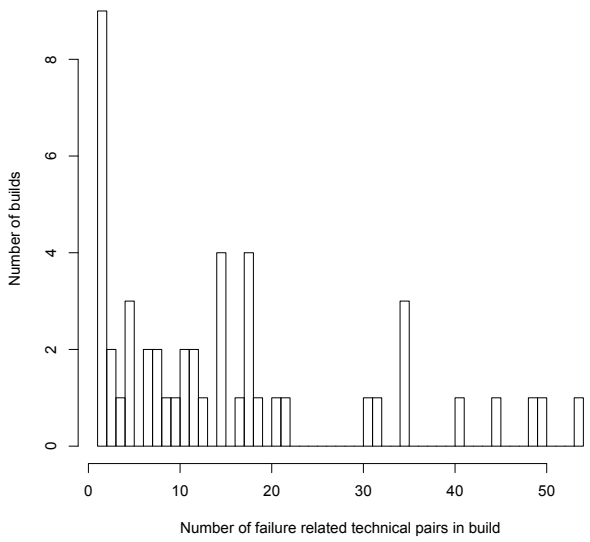


Figure 5: Histogram plotting how many builds have a certain number of failure-reated technical pairs.

~~that matter to the upcoming gap.~~

6.1 ~~STC and Build Outcome~~

~~Our analysis found that in Jazz,~~

6.1 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 ~~socio-technical congruence index does not influence build outcome. Similarly, only a small number (~~total of 120 ~~out of 961, 2) of technical pairs could be~~ 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 ~~indicates that those developer pairs that did not talk to each other although they shared~~ means that 97 pairs of developers that had a technical dependency ~~were generally not harmful in the Jazz™ project. Below we give two 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~~ did not communicate with each other and consequently increased the likelihood of a build failure. Our results not only corroborate past findings [?, ?] that 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 , then he can act without the need to talk to the respective other developer. This, in turn, lowers the~~ 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 ~~congruence because developers do not need to talk about as many technical dependencies.~~ gaps, namely the actual developer pairs where the gaps occur and that increase the likelihood of build failure.

~~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 . 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~~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. [?] found that using the organizational distance between people predicts failures. 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.

6.2 ~~Investigating Failure-Related Pairs~~

~~Our analysis revealed 120 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 74) indicate a strong likelihood that their presence~~ 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~~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 :~~ , 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.

~~We perform 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~~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. ~~To perform this Fischer Exact Value test we counted in how many failure-related pairsand in how many success-related pairs the developer appears. Together with the number of total failure~~

All features shown in Table 5 are significant at the $\alpha = .001$ level (indicated by the p-value 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~~***). 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.

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 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. This implies that there might be skill or habits that a developer exhibits, that may be harmful to the build outcome.

Similar to the analysis of the individual developer we investigated individuals at a more abstract level, i.e. the role played in the project. 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.

Typically developers work in teams rather than on their own, where each team is entrusted with developing a specific part of 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 software project. 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. presence of possibly confounding variables.

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

We planned to investigate the influence of geographical location on the different pairs . But in

7. REGRESSION ANALYSIS

Although the previous analysis showed that there are technical pairs that seem to have a negative influence on the build result we need to check how they influence each other. Additionally we need to control for confounding variables that might have an influence on the build outcome. Such variables are: 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 build a logistical regression model using the control variables.

Overall the logistic regression is confirming the developer pairs identified in Section 6. Even in the presence of confounding variables such as number of files changed and numbers of developers contributing to a build. Thus we can answer our third research question with yes the identified technical pairs are still significant in the presence of possibly confounding variables. In other words this means that 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 resultsdeveloper pairs add additional information besides common indices for software quality.

Table 5 shows an excerpt from the complete logistical regression. Because we have 2872 developer pairs, space constraints prevent us from reporting the whole 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. 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 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.

Now that we have more insights into the developer pairs that are Since we model ERROR builds with 0 and OK 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 , the next step would be to advise strategies to break those patterns. have a negative coefficient.

7.1 Towards the design of a real-time recommender system

As shown in Table 5 two of the four control features, number of change sets per build and number of developers contributing to a build increase the likelihood of a build succeeding. On the other hand number of files changed per build and number of work items per build increase the likelihood of a build failing.

Our findings

8. PRACTICAL IMPLICATIONS

Our findings have several implications for the design of collaborative systems. The analysis of the By automating the analyses presented here we can incorporate the knowledge about developer pairs that tend to be failure related can be automated and leveraged in a real-time recommender system. Not only does such a system do we provide the recommendations that matter to the upcoming build, but provides we also provide incentives to motivate developers to talk about their technical dependencies.

Project historical data can be used Such a recommender system can use project historical data to calculate the likelihood that the builds an upcoming build fails given a particular developer pair that worked on that build without talking communicating to each other. In the case of the pair (Adam, Bart) the system may recommend that these developers should talk communicate about their technical dependencies. Thus, we inform them that the next build will fail with , as there would be a probability of 91% if they do of failure of the next build should they not follow the recommendation. This probability does not only serve This probability of failure serves as mechanism to rank importance of a socio-technical gap but also 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™. 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 as another example from our study, most developer pairs consisted of developers that were part of different teams. Management In such situations management may decide to investigate reasons for coordination problems that include factors such as geographical or functional distance in the project.

9. 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~~ other threat deals with possible problems from the conceptualization of constructs in our study.

9.1 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 [**?**], are mitigated in Jazz by its development process. ~~In~~ The Jazz™ ~~the~~ team's development process demands that the developers coordinate using workitem discussion.

9.2 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.

10. CONCLUSION~~AND CONSEQUENCES~~

Our study investigated the relationship between ~~socio-technical congruence and integration~~ 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.

~~We did not, however, find a strong relationship between~~ Our case study of coordination in the Jazz project indicates that historical project information about socio-technical ~~congruence or the socio-technical gaps and build outcome in the Jazz project. The influence of those few~~ 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 ~~, however,~~ very high. ~~We found~~ 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. ~~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~~ 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~~. The~~ , 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 ~~the upcoming build and we discussed how collaborative systems can incorporate such recommendations in real-time. As a next step we~~ software integrations. We plan to extend this ~~investigation in two~~ research in a number of directions:

~~*Finer edges*~~*More specific information in socio-technical networks.* We plan to ~~refine~~ 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.

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

~~*Finer*~~ *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.

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

11. ACKNOWLEDGEMENTS

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

to the Jazz™ development team as well as the IBM research for they ongoing support.