Thesis Journal Version

Adrian Schröter, *Member, IEEE* and Daniela Damian, *Member, IEEE*

***Abstract*—Efficient coordination among software developers is one key aspect in producing high quality software on time and on budget. Many factors such as team distribution or the structure of the organization developing the software can increase the level of difficulty in coordination. However, the problem runs deeper, it is often unclear which developers should coordinate their work. We propose to leverage the concept of socio-technical congruence (which contrasts coordination needs with actual coordination) to improve the social interactions among developers by developing an approach and its implementation into a recommender system that identifies relevant coordinators. Our unit of analysis is the integration build whose outcome represents the quality of coordination. We developed and applied an approach in a number of case studies of the IBM Rational Team Concert development team. As each software product is just the latest integration build ensuring a failure free build is of utmost importance to industry. While developing an approach to improve coordination among software developers, we uncovered that unmet coordination needs as well as the communication structure in a team significantly influence build outcome.**

***Index Terms*—IEEEtran, journal, LaTeX, paper, template.**

I. Introduction

The software industry often visible through some of big companies such as Microsoft, Google, IBM, Dell, Apple, Oracle, and SAP represent several hundred billion US Dollars of profit a year. For example the software industry in USA in 2002 was producing according to the US Census a total revenue of 103.7 billion USD[1]. As many engineering companies those companies in the software industry strive to optimize their engineering processes to produce software of higher quality in less time.

Software engineering researchers all over the world have dedicated countless hours to improve the way software is developed. Several fields some not directly aimed at increasing productivity such as developing better programming languages [1], smarter compilers [2], and better educationial methods to teach algorithms and data structures [3] contribute indirectly. Other fields are more directly interested in productivity, among them are research in software processes [4], effort estimation [5], [6], and software failure prediction [7].

The vast body of knowledge accumulated to improve the software engineering process is strongly biased towards analyzing the technical side: supporting coding activities (e.g. [8], [9]) and analyzing source code to improve quality [10], [11]. Since producing source code is the main objective of software developer optimizing the coding aspect [8], [9] as well as analyzing the produced code for issues [12], [13] lies at hand.

Others have focused on the people that produce the code. Studying their behaviour around coding activities [14], how they communicate [15], [16], and how developer relations relate to productivity [16] and quality [17], [18]. As in the former case there is much merit in focusing on the developer in the end she implements the features a software consists of and she inevitably introduces errors to the code base.

Both avenues, studying the human aspect and studying the technical aspect, yielded many useful results. For example, on the human side, the organizational distance between developer is a good predictor of failure on file level [19], and on the technical side similar changes timely close are a good failure predictor [20].

Yet, to truly be able to optimize the software engineering process a more holistic view is needed that marries both the technical and social aspects. One such way to marry those two aspects that as Conway stated are influencing each other [21] is to use the concept of socio-technical congruence in software engineering first formalized by Cataldo et al [22]. They proposed to overlay networks constructed from social (who communicates with whom) and technical (whose code depends on whose source code) dependencies to get an overview of a projects social and technical interdependencies and derive insight through the miss-match between those two networks.

Socio-technical congruence forms a great basis to leverage several digitally recorded data treasures to generate useful and actionable information. Patterns of developer pairs showed that there are developers when not talking to each other yet sharing a technical dependency endangered the upcoming software build. Furthermore, we found in a student project that certain issues experienced during development can be traced back to code dependencies that could have been detected in real time.

To complement the research that studied the relationship between socio-technical congruence and performance, we focus on build outcome as a metric for software quality. Although build outcome is rarely considered when studying software quality, as it a course measure that often indicates multiple issues rather than a single specific one, studying build outcome is important as build success is fundamental in creating a product that can be shipped to a customer. Often a successful build indicates that not only all test cases deemed important passed, a successful build towards the end of the release cycle often is the only indicator of customer acceptance with respect to requested features and their stability. Hence, build success is of utmost importance to a business as it forms the very product the business hopes to sell.

II. Background

In this Section we provide an overview of related work: (1) the research on coordination in software development teams and (2) failure prediction using social networks.

[1]http://www.census.gov/prod/ec02/ec0251i06.pdf last visited May 10th, 2012

A. Coordination in Software Engineering Teams

Software is extremely complex because of the sheer number of dependencies [23]. Large software projects have a large number of components that interoperate with one another. The difficulty arises when changes must be made to the software, because a change in one component of the software often requires changes in dependent components [24]. Because a single person's knowledge of a system is specialized as well as limited, that person often is unable to make the appropriate modifications in dependent components when a component is changed.

Coordination is defined as "integrating or linking together different parts of an organization to accomplish a collective set of tasks" [25]. In order to manage changes and maintain quality, developers must coordinate, and in software development, coordination is largely achieved by communicating with people who depend on the work that you do [26].

A successful software build can be viewed as the outcome of good coordination because the build requires the correct compilation of multiple, dependent files of source code. A failed build, on the other hand, demotivates software developers [27], [28] and destabilizes the product [29]. While a failed build is not necessarily a disaster, it slows down work significantly while developers scramble to repair the issues. A build result thus serves as an indicator of the health of the software project up until that point in time.

Thus, a developer should coordinate closely with individuals whose technical dependencies affect his work in order to effectively build software. This brings forth the idea of aligning the technical structure and the social interactions [30], leading us to the foundation of socio-technical congruence.

Software developers spend much of their time communicating [31]. Because developers face problems when integrating different components from heterogeneous environments [32], developers engage in direct or indirect communication, either to coordinate their activities, or to acquire knowledge of a particular aspect of the software [33]. Herbsleb, et al examined the influence of coordination on integrating software modules through interviews [34], and found that processes, as well as the willingness to communicate directly, helped teams integrate software. De Souza et al [35] found that implicit communication is important to avoid collaboration breakdowns and delays. Ko et al [15] found that developers were identified as the main source of knowledge about code issues. Wolf et al [18] used properties of social networks to predict the outcome of integrating the software parts within teams. This prior work establishes the fact that developers communicate heavily about technical matters.

Coordinating software teams becomes more difficult as the distance between people increases [36]. Studies of Microsoft [19], [37] show that distance between people that work together on a program determine the program's failure proneness. Differences in time zones can affect the number of defects in software projects [38].

Although distance has been identified as a challenge, advances in collaborative development environments are enabling people to overcome challenges of distance. One study of early RTC development shows that the task completion time is not as strongly affected by distance as in previous studies [39]. Technology that empowers distributed collaboration include topic recommendations [40] and instant messaging [41]. Processes are adapting to the fast pace of software development: the Eclipse way [42] emphasizes placing milestones at fixed intervals and community involvement. These new processes lie the Eclipse way that focus on frequent milestones lends more importance to software builds warranting more support by research as we conduct.

B. Can communication predict build failure?

Social network analysis has an extensive body of knowledge of analysis and implications with respect to communication and knowledge management processes [43], [44]. Griffin and Hauser [45] investigated social networks in manufacturing teams. They found that a higher connectivity between engineering and marketing increases the likelihood of a successful product. Similarly, Reagans and Zuckerman [46] related higher perceived outcomes to denser communication networks in a study of research and development teams.

Communication structure in particular – the topology of a communication network – has been studied in relation to coordination (e.g. [47], [48]) and a number of common measures of communication structure include network density, centrality and structural holes [44], [49].

Density, as a measure of the extent to which all members in a team are connected to one another, reflects the ability to distribute knowledge [50]. Density has been studied, for example, in relation to coordination ease [48], coordination capability [47] and enhanced group identification [46].

Centrality measures indicate importance or prominence of actors in a social network. The most commonly used centrality measures include degree and betweenness centrality having different social implication. Centrality measures have been used to characterize and compare different communication networks constructed from email correspondence of W3C (WWW consortium) collaborating working groups developing new technical standards and architectures for the web [51]. Similarly, Hossain et al [47] explored the correlation between centrality in email-based communication networks and coordination, and found betweenness to be the best measure for coordination. Betweenness is a measure of the extent to which a team member is positioned on the shortest path in between other two members. People in between are considered to be "actors in the middle" and to have more "interpersonal influence" in the network(e.g. [47], [51], [52]).

The structural holes measures are concerned with the degree to which there are missing links in between nodes and with the notion of redundancy in networks [43]. At the node level, structural holes are gaps between nodes in a social network. At the network level, people on either side of the hole have access to different flows of information [53], indicating that there is a diversity of information flow in the network. Structural holes have been used to measure social capital in relation to the performance of academic collaborators (e.g. [54]).

Most prediction models in software engineering to date mainly leverage source code related data and focus on predicting failing software components or failure inducing changes

(e.g. [13], [52], [55], [56]). And only few studies, such as Hassan and Zhang [57], stepped away from predicting component failures and used statistical classifiers to predict integration outcome. We want to extend the body of knowledge surrounding prediction models using communication data or focusing on build outcome by investigating how to improve communication among software developers to prevent build failures.

III. Research Questions

The concept of socio-technical congruence shows potential to help make software development more efficient. Cataldo et al [22] demonstrated its relation to productivity, and we show among the ability to use socio-technical congruence to predict build outcome. The concept of socio-technical congruence lends itself to improve software development as it is based on social networks connecting developer on a coordination and technical level. Because of the concept being based on networks it is possible to manipulate the networks.

Any socio-technical network can be manipulated in two ways: (1) changing the technical dependencies among developer by refactoring or architectural changes to make them unnecessary and (2) by engaging developer in discussions about their recent work and therefore creating a coordination edge in the socio-technical network. As a first step we need to assess if the actual communication structure among software developers has an influence on build success to lay the basis for manipulating the actual coordination to increase build success. As a follow up step, we need to explore the relationship between socio-technical networks and build success. Especially we are interested in whether missing actual coordination in the face of a coordination needs is related to build failure.

We start with investigating the influence of communication among team members in the form of social networks on build success. Next, we investigate the amount of gaps (unfilled coordination needs) between developers as highlighted by socio-technical networks and the socio-technical networks in the form of socio-technical congruence can be brought into relation with build success. Therefore Section V and VI investigate the following two research questions respectively:

- **RQ 1.1:** Do Social Networks influence build success? (Section V)
- **RQ 1.2:** Does Socio-Technical Networks influence build success? (Section VI)

Having found a relationship between socio-technical networks, especially gaps between coordination and coordination needs with build success, while knowing that communication alone has an effect on build success, we formulate an approach to leverage socio-technical networks (Section VII). Thus we focus focus on evaluating this approach in two ways: (1) gathering general statistical evidence that parts of the network can be manipulated to increase build success and (2) exploring the acceptance of such recommendation based on those manipulations by developers. Hence, we are guided by the following two research questions:

- **RQ 2.1:** Can Socio-Technical Networks be manipulated to increase build success? (Section VII)
- **RQ 2.2:** Do developers accept recommendations based on software changes to increase build success? (Section IX)

In the following discussion (Section X) we will highlight how our findings from our research questions supporting the approach we detailed in Section VII.

IV. Constructs and Data Collection Methods

A. Constructs

From the definitions introduced previously we can derive the three central constructs we work with: (1) the social network connecting communicating and coordinating developers, (2) the technical network connecting developer that are dependent through code artifacts, and (3) the socio-technical network that combines the social and technical network in a meaningful way. These constructs are important for the three sections that are mining the repository provided by the Rational Team Concert development team (Sections V, VI, and VII).

1) Social Network: A social network is represented as a graph that consist of nodes connected by edges. In our approach, the nodes represent people and edges represent task-related communication between these people.

The approach is repository and tool independent and can be applied to any repositories that provide information about people, tasks, technical artifacts, and communication, this includes work, issue, or change management repositories, such as Bugzilla or IBM Rational Team Concert; or source code management systems, such as CVS or IBM Rational Team Concert; or even communication repositories such as email archives.

There are three critical elements that are necessary to construct task-based social networks for a collaboration scope and that need to be mined from software development repositories. *Project Members* can be developers, testers, project managers, requirements analysts, or clients. *Work Items* are units of work within the project that may create a need to collaborate and communicate such as a bug report or feature request. *Work Item Communication* is the information exchanged while completing a work item and is the unique information that allows us to build task-based social networks.

2) Technical Network: Building technical networks follows a very similar approach as we described for building social networks. In fact, the technical network is a social network whose main distinction from the social network described earlier lies in the way edges between nodes are created. We derive the name of technical network from the way we link developer with each other, namely if they are modifying related source code artifacts. As in the previous network construction, the construction of the technical network is based on three components. *Project Members* can be developers, testers, project managers, requirements analysts, or clients or in general anyone that modifies software artifacts through change-sets. *Change-Sets* consist of a number of artifacts that have been modified as well as the modifications themselves. *Software Artifact Relation* can be defined in several different ways. For example, in Figure 1c Alfred and Bob are related through a technical relationship because they modified the same file.

Constructing technical networks therefore follows three steps: (1) we gather all change-sets of interest, (2) identify the relations between artifacts, (3) infer from the change-sets and the relations between the source code artifacts the relation between the artifact owners.

3) Socio-Technical Network: Socio-Technical networks are a meaningful combination of both social and technical networks. Selecting this meaningful combination reflects itself in the selection of the work-items in the case of building the social network and selecting the change-sets and their relations in the case of the technical network. Hence constructing a socio-technical network requires the following four steps:

1) **Selecting the Focus** used for the socio-technical network represents the glue that binds the social and technical network into a socio-techncial network. This focus also referred to as filter in our earlier publication [58], determines the content of the networks.
2) **Constructing the Social Network** we build social networks from all work items that are relevant to the defined focus.
3) **Constructing the Technical Network** follows the description of constructing technical networks above with the focus determining the change-sets being used to determine and connect developers in the network.
4) **Combining Networks** overlays the networks by unifying the set of developers in both networks.

Figure 1 shows an example on how we in our studies of the IBM Rational Team Concert development team used to create socio-technical networks. In the first step (Figure 1a) we set the focus to be a software build which allows us via the change-sets that made it into the build to infer what work items are also represented in said build. Given the focus, the social network can be constructed using the work items that can be linked to the software build (Figure 1b). Similarly the construction of the technical network relies on the change-sets that went into a build. To actually infer edges between developer, we relying on co-changed files within a build as an indicator of work dependency (Figure 1c). Finally, the two networks are combined and yield the socio-technical network shown in Figure 1d.

B. Data Collection Methods

To conduct our research we drew upon multiple data sources. We employ repository mining techniques to identify larger trends in measurable activities. In contrast to gain a more in-depth understanding on how developers actually work and deal with interdependencies especially how they would react to certain recommendations and whether they are can be made useful we employ qualitative methods.

1) Repository Mining: Software development usually uses a number of tools to manage information electronically such as version archives and issue trackers. Additionally to storing source code and tasks/issues, those software repositories can also contain digital communication such as forum and email discussions.

We extract information from three different types of repositories: (1) version control, (2) task management, and (3) build engine systems. The version control supplies us with the knowledge on how developers are connected through their technical work. The task management supplies us with the information on who communicated with whom with respect to a work item. And lastly the build engine supplies us with the focus to construct socio-technical networks.

In order to derive socio-technical networks we need to link the different artifact types. Within IBM Rational Team Concert as illustrated in Figure 1a work items are linked to change-sets and change-sets are linked to builds, therefore, establishing the connections needed to construct socio-technical networks with a build as focus.

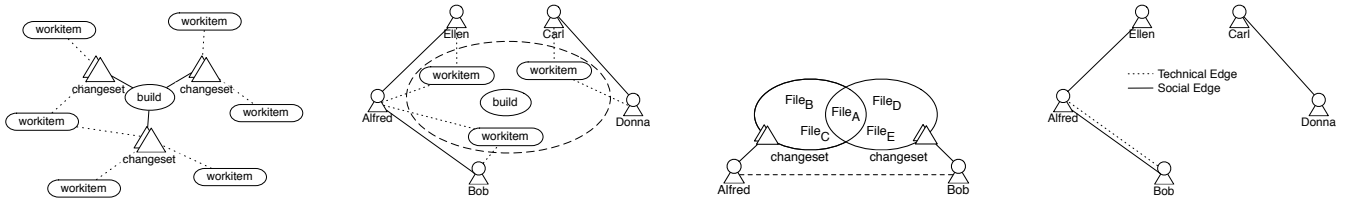
2) Surveys: To complement the insights we obtained from mining repositories we use surveys. Surveys are designed iteratively and piloted before deployment, and intended to collect input to enrich and clarify information obtained from the software repositories. With each survey we try to minimize the time each developer needs to spend completing them, which usually limits ourselves to focus on closed questions. We constrain ourselves in this way to minimize the distraction to each individual developer and thus increase the response rate.

Our surveys are deployed through web services to make the collections more convenient to each developer as they are spending most of their times working at a computer enabling them to fill out the survey at their earliest convenience. Keeping track of a paper version is more cumbersome as they might not easily be returned, especially considering that the development teams we are collaborating with are distributed across different continents.

3) Observations: The next richer and also to the developer more distracting method of data gathering are observations. Although not necessarily actively interrupting and distracting developer, the act of observing can distract developers and also change their behaviour. In order to minimize this type of distraction and to mitigate the observer bias, we employed a special form of observation study namely participant observation. In short we became both an observer and a participant. Being a participant observer has a multitude of advantages:

Reciprocity. By participating in the actual development we can provide value to the development team from the very beginning. This, in turn, motivates the developers to give us the time we need to conduct other parts of the study, like surveys and interviews. *Learning the Vocabulary.* Each development project has its own project vocabulary [59] in order to effectively and clearly communicate. Understanding this vocabulary as an outsider is not necessarily easy but very important to make sense of comments and answers supplied in interviews and surveys. *Understanding the Context.* For example, in one study our observation period coincided with the months prior to a major release. Because of this closeness of our observation period to a major release we observed mainly activities around integration testing with little coding activity aside from fixing major bugs.

Asking more Meaningful Questions. A better understanding of the project and how it affects the individual developer as well as understand the vocabulary helps with phrasing better questions.



(a) Inferring to the build focus relevant change-sets and work items.

(b) Constructing an social networks from work item communication.

(c) Linking developers in a technical networks via change-set overlaps.

(d) Combine social and technical networks into a socio-technical network.

Fig. 1: Constructing socio-technical networks from the repository provided by the IBM Rational Team Concert development team.

Besides gaining a better understanding of some easily missed or miss-understood intricacies, working together with the developer establishes a trust relationship [60]. This trust helps to mitigate observation biases that are introduced by just observing as well as makes developers more forthcoming during interviews and surveys [60].

4) Interviews: To further enhance our understanding on how developers view the situation, we employed interviews. Instead of following a structured interview approach, we opt for a semi-structured interview with a focus on war stories. War stories [61] ask the interviewee to share memorable stories from work life. The interviewer than can explore those war stories and help shape the focus of the discussion of those events. This type of interview comes with two major benefits over structured interviews that follow a set of questions:

Focus onto for the interviewee important events. Knowing the pain points as they are perceived by the projects participants allows us to focus on important issues. With prepared questions the focus of the interview might not uncover what is important to the interviewee and thus we might miss areas. *Better recall of events by interviewee.* Recall of events of importance is better than of arbitrary events [61]. This allows us to place more confidence into the reports and answers given by the interviewees.

By asking the interviewee to tell war stories of memorable events it can be more difficult to gain the insights into a particular area of interest if the war stories veer to far off the topic of interest. It is therefore necessary that the interviewer has a good understanding of the project and the project language to explore the stories for relevance to topics of interest making it thus more demanding on the side of the interviewer.

V. Communication and Failure

We open our investigation of how to modify the social relationships among software developers, represented by the communication between them, with searching for a relationship between communication and build success. This forms the basis and justification for an approach that we describe in a later Section to allow us to manipulate the social interactions among developers. Thus this Section explores our first research question:

- **RQ 1.1** Do Social Networks influence build success?

A connection between communication among developers and any sort of software quality including software builds makes intuitively sense. For example, any non trivial software project consists of several interdependent modules and with the growing size and number of modules the work of more than one software developer is required to finish the project within a certain time constraint often mandated by a customer. Now due to the interdependence of the software modules developers assigned either to the same or to interdependent modules need to coordinate their work. This coordination is in the most part accomplished through communication, which can take any form from a face-to-face discussion to electronically asynchronous messages such as email. Coupled with the fact that communication is inherently ambiguous and can often lead to misunderstandings, errors based on such misunderstandings may be introduces into the source code. Thus, we are confident that there exists a connection between developer communication and build success.

A. Measures

To address our research question we analyze data from a large software development project IBM Rational Team Concert.

1) Coordination outcome measure: In our study we conceptualize the coordination outcome by the Build Result, which is regarded as a coordination success indicator in Jazz and can be `ERROR`or `OK`. We analyze build results to examine the integration outcomes in relation to the communication necessary for the coordination of the build.

2) Communication network measures: To characterize the communication structure represented by the constructed social networks for each build (as described in Section IV), we compute a number of social network measures. The measures that we include in our analysis are: Density, Out-Degree/In-Degree/InOut-Degree Centrality, Betweeness and Structural holes [44]. Some of these measures characterize single nodes and their neighbours (ego networks), while others relate to complete networks. As we are interested in analysing the characteristics of complete communication networks associated to integration builds, we normalize and use appropriate formulas to measure the complete communication networks instead of measuring the individual nodes.

B. Analysis and Results

We combined communication structure measures into a predictive model that classifies a team's communication structure as leading to an `ERROR` or `OK` build. We explicitly exclude the

	Team Level Builds					Project Level Builds		
	B	C	F	P	W	nightly	weekly	beta
ERROR Recall	.55	.75	.62	.66	.74	.89	1	.92
ERROR Precision	.52	.50	.75	.76	.66	.73	.92	.92
OK Recall	.75	.62	.84	.80	.50	.50	.75	.67
OK Precision	.77	.83	.74	.71	.60	.75	1	.67

TABLE I: Recall and precision for failed (ERROR) and successful (OK) build results using the Bayesian classifier.

technical descriptive measures such as #Contributors, #Change Sets and the #Work Items from the model in order to focus on the effect of communication on build failure prediction. We validate the model for each set of team-level and project-level networks separately by training a Bayesian classifier [62] and using the leave one out cross validation method [62].

For example, to predict the build result N of team F's 55 build results, we train a Bayesian classifier with all other 54 build results and their communication related network measures. Then, we input the communication measures of Build N's related communication network into the classifier and predict the result of build N. We repeat the classification for all 55 builds of team F and sum up the number of correctly and wrongly classified results. The classification quality is assessed via recall and precision coefficients, which can be calculated for ERROR and OK build predictions.

We repeated the classification described above for each team and project-level integration. Note that the model prediction results only show how the models perform within a team and not across teams. Table I shows the recall and precision values for as to OK and ERROR leading classified communication networks for each of the five team-level and three project-level integrations. Since we are interested in the power of build failure prediction, the error related values from our model are of greater importance to us. The ERROR recall values (how many ERRORs were classified correctly) of team-level builds are between 55% and 75% and the recall values of the project-level builds are even higher with at least 89%. The ERROR precision values are equally high.

VI. Socio-Technical Congruence and Failure

Knowing that social networks have an effect on build success opens the next question as to how or more precisely which parts of the social network should be changed to increase the likelihood for a build to succeed. For this reason we turn to the concept of socio-technical congruence as it postulates that developers should communicate once their work intersects. Thus in this section we explore the effect of socio-technical networks on build success:

- **RQ 1.2:** Does Socio-Technical Networks influence build success?

Although socio-technical congruence has only been studied in connection with productivity intuitively there should be a connection to software quality such as build success. For example, imagine two developer modifying classes that share call and data dependencies and one developer making changes that violate certain assumptions the other developer relies on when using the modified code. This might introduce an error that could have been prevented if both developer would have discussed their work. Thus, we hypothesize that the concept of socio-technical congruence relates to software quality as well as productivity and might be used to point developers towards improvements in the social network by pointing out developers that should communicate.

Variable	Coef.	S.E.	p
Intercept	-0.5459	0.4663	0.2417
Congruence	**6.3410**	**1.6262**	****0.0001**
Authors	**-1.9759**	**0.5310**	****0.0002**
Files	**-1.0734**	**0.4561**	***0.0186**
Work items	-0.1456	0.2355	0.5363
Build type=I	**2.1533**	**1.0526**	***0.0408**
Build type=N	4.6833	200.7587	0.9814
Build date	-0.6560	0.6709	0.3282
Congruence * Build type=I	**-9.2151**	**2.5572**	****0.0003**
Congruence * Build type=N	-7.7308	91.8053	0.9329
Congruence * Build date	**-5.1266**	**1.9290**	****0.0079**
Authors · Build type=I	1.2688	0.7028	0.0710
Authors * Build type=N	105.4123	535.8792	0.8441
Authors * Build date	-0.6061	0.3616	0.0937
Authors * Files	0.7663	0.4289	0.0740
Files * Build type=I	1.0920	1.1838	0.3563
Files * Build type=N	-37.9274	199.2314	0.8490
Work items * Build date	**0.8040**	**0.3003**	****0.0074**
Build type=I * Build date	**2.6442**	**0.7678**	***0.0006**
Build type=N * Build date	84.7252	344.8129	0.8059
Model likelihood ratio	101.92		$R^2 = 0.581$
$^*p < 0.05$; $^{**}p < 0.01$		191 observations	

TABLE II: Logistic Regression models predicting build success probability with main and interaction effects

A. Calculating Congruence

In Section IV we described socio-technical networks and how we conceptualize them. If we reformulate this network into the terms originally used by Cataldo et al [22] the matrix representation of the technical dependencies among software developers turns into the coordination needs matrix CN and the social network in matrix representation is the actual coordination matrix AC. Thus we calculate the socio-technical congruence index by dividing the cardinality of AC without CN by the cardinality of CN.

B. Analysis Methods

Logistic regression is ideal to test the relationship between multiple variables and a binary outcome, which in our study is a build result being either "OK" or "Error". The presence of many data entities in this project means that we must consider confounding variables in addition to the socio-technical congruence when determining its effects on the probability of build success. Informally, logistic regression identifies the amount of "influence" that a variable has in the probability that a build will be successful. The two main variables we are interested in are as aforementioned the socio-techincal congruence index as well as the ratio between gaps and coordination needs, that is technical dependencies among developers that are not accompanies by a corresponding social dependency. To assess the fit of the logistic regression models, we use the Nagelkerke pseudo-R^2 and AIC.

C. Results

In the RTC repository, we analyzed 191 builds; of these builds, 60 were error builds, and 131 were OK builds. The congruence values are low on average with a mean value of 0.331 meaning that about one-third of the coordination needs are satisfied by actual coordination.

1) Effects of interactions involving congruence: The type $\times$ congruence interaction effect, the date $\times$ congruence interaction, and the type $\times$ date effect are each significant in our model (Table II). The congruence model (Table II) the effect of congruence on continuous builds is significant, and that increasing congruence also increases the probability that a continuous build will succeed.

2) Social and Technical Factors in RTC Affecting Build Success and Congruence: In light of our results, we examine not only the number of work items $\times$ date significant interaction, but different social and technical factors that may affect congruence and build success probability to find explanations for the interactions between socio-technical congruence and build success probability in RTC. Specifically, we examine the effect of build date on work items, coordination around fully-congruent builds and incongruent builds, and the effects of commenting behaviour on builds.

VII. The Approach

Motivated by research showing that socio-technical gaps represetend by low socio-technical congruence (presented in Section VI) increases the chance of a build to fail we formulate an approach that generates actionable knowledge to alleviate socio-technical gaps. Our approach takes into account the past history of a project analyzing socio-technical gaps with respect to their individual relation to build failure. These recommendations aim at initiating coordination between two developers that form a gap in the current build, thus increasing the chance of a build to succeed. This approach can support projects that use electronic repositories such as automated build engines, version control and task management.

For our approach we define a socio-technical gap as the relationship between two developers that share a technical dependency (implying coordination need) without any social interaction (implying unmet coordination need). A technical dependency can be inferred by two developers changing the same file, or a developer changing a method that another developer's code is calling. In contrast two developers that share a technical dependency and that also coordinate their work are referred to as being in a *socio-technical pair*. For example, two developers that discuss their work through email are said to coordinate; if they additionally share a technical dependency then they form a socio-technical pair. When analyzing the developer pairs in a project's set of builds, in our approach we recognize that each *technical* pair can have a *corresponding socio-technical* if the same two developers have a technical dependency matched by actual coordination in a different build.

Our approach analyzes the technical pairs in relation to build failure in the following steps:

1) Identify the set of all technical pairs T across all builds in the project.
2) From set T, select the set of *harmful* pairs H by identifying those technical pairs that are statistically related to build failure. To determine the statistical relationship we employ a Fisher exact value test that comparing the frequency of each technical pair's occurrence in failed vs successful builds. The p-values of the Fisher Exact Value test should be adjusted to account for multiple hypothesis testing.
3) To form our set of recommendations R we remove from H those pairs where the corresponding socio-technical pair is statistically related to failure (H_f) (which would indicate that even matching the technical dependency with actual coordination would not prevent the build from failing).
   Therefore $R = H - H_f$.
4) Having identified R, we further refine our recommendation by identifying two sets of technical pairs: R_1 contains the pairs that have a corresponding socio-technical pair that relates statistically to success and R_2 contains the pairs that either (1) have a corresponding socio-technical pair that does not relate to success or failure, or (2) do not have a corresponding socio-technical pair. Therefore, $R = R_1 \cup R_2$.
   In our recommendation the R_1 set has highest priority because it contains developer pairs that contributed to a successful build when matched by actual coordination.
5) Finally, for each set in our recommendation R (R_1 and R_2) we rank 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:

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

   where: pair_{failed} is the number of failed builds in which the pair occurred; total_{failed} is the number of failed builds; $\text{pair}_{success}$ is the number of successful builds in which the pair occurred, and $\text{total}_{success}$ is the number of successful builds. A value of p_x closer to one means that the developer pair is strongly related to build failure.

In summary, our approach analyzes the technical dependencies, actual coordination and build quality in all existing builds in the project, and recommends a ranked list of developer pairs that, if present in the current build, will increase the current build's chance of failure. This list is prioritized by the probability of this chance. This recommendation essentially represents the pairs of developers that should communicate in order to increase the chance of a build to succeed. In a busy manager's workday, the ranking of each developer pair is useful in prioritizing which socio-technical gaps should be closed first.

VIII. Applying the Approach

To evaluate our approach we analyzed collaboration and build data in the IBM Rational Team Concert™ (RTC) development team. An thus answer our third research question:

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

TABLE III: Twenty most frequent *technical pairs* that are failure-related.

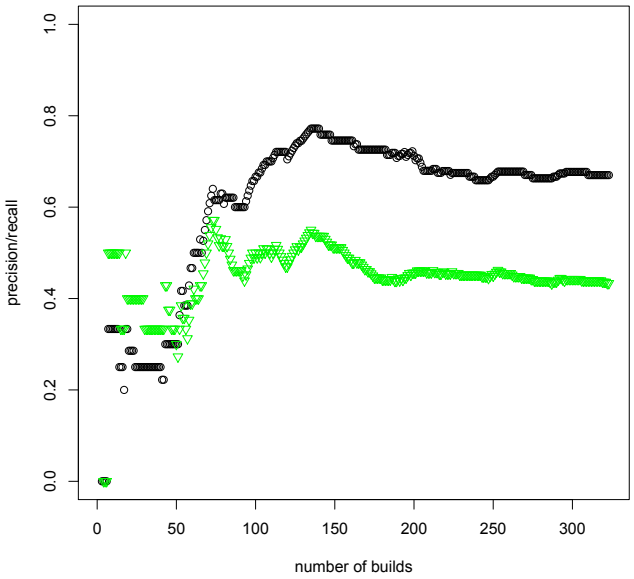


Fig. 2: Precision (green) and recall (black) of the logistic regression.

- **RQ 2.1:** Can Socio-Technical Networks be manipulated to increase build success?

RTC is a product that integrates source code management, agile planning, and issue management into a single server/client application that the team itself uses for development. RTC allows developers to link changes they made directly to work items, thus establishing within their repository traceable links between builds and work items through the changes they made. The repository spans three months in which the team started 326 builds, 99 of which failed and 227 that were successful. The team consists of more than 100 developers distributed across seven major sites in Europe, Asia, and North America. Due to the team's geographical distribution and agile development process the team's communication is largely found in the online repository in the form of comments related to work items.

In RTC we conceptualize a *technical pair* as two developers that modified the same file in a build. Similarly, two developers are in a *socio-technical pair* if they modified the same file in a build *and* commented on a work item that was linked to the build in which they changed the same file.

Preliminary Results. Using our approach we analyzed all existing builds in the project and were able to identify a ranked list of 120 developer pairs that should communicate in order to increase the likelihood of the upcoming build to succeed.

We used the approach as follows: From the total of 2872 developer pairs, we found 961 technical pairs (set T) (step 1). Of these 961 technical pairs we found 120 harmful pairs (set H) that were statistically related to build failure (step 2). See Table **??** for the top twenty pairs.

Next, for all pairs in set H, we examined their corresponding *socio-technical pairs* (step 3). Because this set (H_f) was empty in our data we did not have to remove any pairs from H. In step 4 we identified that none of the 120 technical pairs in R had an existing corresponding socio-technical pair related to build success nor failure. Thus R_1 was empty and $R = R_2$.

In step 5 we ranked the recommendations in R by the coefficient p_x as shown in Table **??**. 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 pair$_{success}$ = 3 with total$_{failed}$= 99 and total$_{success}$= 227 result in p_x= 0.9016.

To further validate our recommendations, we used the approach to generate a list of technical pairs for each of the 326 builds in our data. For any given build N we used the technical pairs as recommended by our approach from all the previous builds to build a logistical regression model predicting the outcome of build N. Applying this method to all builds yields Figure 2 which shows the recall (black) and precision (green) values for the a logistical regression model used for prediction. The recall and precision are cumulative at each point, including the prediction results obtained for the previous builds. The logistical regression ended with a precision and recall value of .43 (median: .45) and .67 (median: .67) respectively, thus outperforming a random guess based on the ratio between failed and successful builds. This further suggests that the recommendations that our approach identifies might prevent build failure.

In seeking explanations for the lack of communication in the RTC technical pairs found by our analysis, we were able to identify that most of these technical pairs consisted of developers belonging to different teams. This confirms Naggappan et al. [19]'s result that organizational distance predicts failures. Although the RTC team strongly emphasizes communication regardless of team boundaries, it still seems that organizational distance had an influence on its communication behaviour.

To further validate our recommendations, we used the approach to generate a list of technical pairs for each of the 326 builds in our data. For any given build N we used the technical pairs as recommended by our approach from all the previous builds to build a logistical regression model predicting the outcome of build N. Applying this method to

all builds yields Figure 2 which shows the recall (black) and precision (green) values for the a logistical regression model used for prediction. The recall and precision are cumulative at each point, including the prediction results obtained for the previous builds. The logistical regression ended with a precision and recall value of .43 (median: .45) and .67 (median: .67) respectively, thus outperforming a random guess based on the ratio between failed and successful builds. This further suggests that the recommendations that our approach identifies might prevent build failure.

In seeking explanations for the lack of communication in the RTC technical pairs found by our analysis, we were able to identify that most of these technical pairs consisted of developers belonging to different teams. This confirms Naggappan et al. [19]'s result that organizational distance predicts failures. Although the RTC team strongly emphasizes communication regardless of team boundaries, it still seems that organizational distance had an influence on its communication behaviour.

IX. Acceptability of Recommendations

In this section we present a study that explores to what extend developer would welcome recommendations generated through our approach. Thus the guiding question for this section is:

- **RQ 2.2:** Do developers accept recommendations based on software changes to increase build success? (Section IX)

By leveraging a combination of a participant observer study with interviews and surveys, we analysed the communication behaviour of a component team within the Rational Team Concert development team. We compiled a list of findings that detail on what triggered developers to talk to each other about change-sets – the level of our recommendations.

A. Methodology

In the three month period in which we joined the development team as an intern developer we kept a detailed logbook. From that log book we both inferred the foundation of the findings we present. Furthermore, the log book entries served as the basis for the questionnaire pilot that we run with the local team we were embedded with before replying the questionnaire with the Rational Team Concert development team at large. While the Rational Team Concert development team was responding to the questionnaire we interviewed the local team that served as pilot group for the questionnaire.

B. Findings

The four findings we present in this subsection are derived from the three month participant observation period as well as the 36 survey respondents as well as the 10 interviews we conducted with the local development team.

1) Development Process: One of the strongest findings from our study was the effect of the development mode that the team is in on its communication behaviour. Developers and managers alike give much importance to the development mode they are in, namely (1) normal iteration, and (2) endgame.

The normal iteration mode mainly consists of work that can be planned by the developer. This work tends to be new feature development, or modifications to existing features. Most of the planning for it has been laid out in advance; furthermore, each developer individually knows what features have been assigned to her, and can plan ahead to meet her obligations. In contrast, the endgame mode mainly consists of work that is coming uncontrollably in short intervals from others. As a result of integration and more intense testing, defects crop up more rapidly and need to be addressed more quickly. Beyond allocating time for endgame activities, there is very little in this development mode that can be planned in advance.

The RTC team switches between the two modes in each iteration. Of the six weeks of a regular iteration, four weeks are assigned to normal iteration mode and two weeks to endgame. However, as deadlines approach, there are occasional special iterations that consist mostly of endgame-like work. In this manner, the same detailed pattern of alternation between normal iteration and endgame within an iteration is reproduced at a higher level.

We identified the same pattern with respect to the amount of change-set-related interactions in the team. We found that the mode in which the team is currently in is an important factor determining whether developers will feel a need to communicate about change-sets. Being in endgame mode increases the need for developers to communicate about a change-set, whereas being in normal mode decreases the need to request information about it.

The first author of this paper experienced both modes during his participant-observer time with the RTC team. He joined the team in a release endgame phase, which was characterized by fixing bugs that were reported by testers, and he helped by fixing minor bugs as well as setting up servers for testing. When the team released the project, it switched gears, and in the decompression after the endgame he had the opportunity to develop a feature for the product. The differences in the patterns of interaction between the two modes were distinctly clear, for him and for his peers. During the endgame mode, developers were essentially "on demand," available to fix whatever bugs were discovered. Later, during the normal iteration mode, they experienced far more autonomy and control over their time, and began working on activities that were less demanding of an interaction back-and-forth, and especially less demanding of keeping track of the change-sets committed by their peers. In our interviews, developers almost unanimously pointed to the separation of the two types of modes, describing the differences in the type of communication in each, and identifying the autonomous *vs.* uncontrollable, inside *vs.* outside influence.

This is not to say that developers do not communicate while they are in normal iteration mode, but that the nature of their communication seems to be different. In normal iteration mode, developers communicate less about concrete change-sets, but they communicate more about high level ideas: for instance, they raise questions about how a feature fits into the existing architecture or general tactics to implement a feature, about how much of it actually needs to be implemented or can be reused from other libraries, and so forth. Generally,

however, the amount of communication decreases in normal iteration mode. As a developer commented in one of our interviews, *"during feature development iterations there are weeks I sometimes don't talk to my colleagues at all."*

In contrast, the endgame mode is characterized by bug reports and last minute feature requests—that is, by work coming into every developer's desk uncontrollably. change-sets become first class concerns during this mode because each change threatens the stability of the product: developers need to evaluate whether the change-set actually improves the product instead of making things worse. Therefore, they develop a habit of reviewing each other's change-sets to assess the risk they pose to the project's stability.

Our survey data provides significant support for this finding. There is a greater likelihood to request information during product-stabilizing phases, such as during the release endgame, and a lesser likelihood to request information when working towards an early milestone. The former was ranked as the most important of the process-related items in our survey; the latter was ranked as the least important. All the survey items that refer to a late stage in the iteration or in the project lifecycle were ranked highly, and all the items that refer to an early stage were ranked lowly. This finding resonates especially with the team we interviewed.

2) Perceived Knowledge of Change Author: Although a successful build is most important in the endgame phase developers are throughout the development interested in creating successful builds to verify the correctness of their work. Thus each change that is potentially interfering with the work of a developer is of interest.

We found that developers keep track of a certain set of developers that are characterized by the level of knowledge they are perceived to have. This means when a developers sees a relevant change from another developer she contacts the other developer if the knowledge of that developer is perceived to not be sufficient in the area the change was applied to.

3) Common Experience and Location: Similarly to the importance of the perceived knowledge of the change author the common experiences and location with the change author influence whether the affected developer starts to communicate with the change author. This, as reported in the previous reported finding, relates to the trust in the skills of change author.

Developers that worked through a problem together or are co-located and thus frequently talk to each other build up trust in each others skills. This trust can result in neglecting required coordination. In most cases this neglect is justified but developers recognize that this trust based approach can cause developer to miss important implications. Thus developers would welcome a way to minimize checking every change that affects them while still being notified of those that are important to check.

4) Risk Assessment: Because the upcoming build is essentially the latest version of the product senior developers and managers are, especially towards the end of a release cycle, interested in assessing the effect a change has on the stability of the product. This need currently translates into a process that while in the release endgame each change that is made to the product needs to be reviewed by two developers.

One major issue with the enforced review is to find an appropriate reviewer. Often developers turn to their collocated team members and walk them through the code change. Ideally developers would like to have a reviewer that is both knowledgeable in the area of the code change and has a good grasp of the potential impact on others. Our approach can provide developers that seek appropriate reviewers with developers that are affected by the change and have proven to be important to talk to in order to avoid a broken build.

X. Discussion

A. An Approach For Improving Social Interactions

We derived the approach presented in Section VII through two case studies that investigate the usefulness of social and socio-technical networks to predict build outcome (Sections V and VI). We conducted a study to see if the approach can generate relevant recommendations in Section V. The study we conducted in the subsequent Section IX further explores the usefulness of the information with respect to whether experts expect the level of recommendations to be of use as well as if these recommendations could be produced in real time and potentially prevent issues from arising.

In Section V we showed that the communication structure of a software development team influences build success, suggesting that there is value in manipulating this structure to improve the likelihood for a successful build. That evidence is further supported by our finding that gaps in the social network constructed from developer communication as suggested by technical dependencies among developers also affect build success (Section VI).

In those two studies we already applied the first four steps of the approach presented in Section VII. We defined the build as the scope together with the build outcome, success or failure, as the outcome metric. Using the scope we constructed social networks from the communication among developer that can be related to a build as described in Section II in both Section V and VI. Section VI used dependencies among change-set committed by developers that are relevant to a given build to construct a technical network to complement the social network forming a socio-technical network.

In Section VII we showed that we are able to produce recommendations from available repository data that affect build success. These recommendations take the form of highlighting two developers that have a technical dependency but did not communicate in the context of the build.

We decided to focus on generating recommendation enticing developer to communicate in order improve build success over recommendation that would suggest code changes changing the dependencies among developer for two reasons: (1) proper code changes are more difficult to suggest without a sufficient understanding of the program requiring more in-depth program analysis and (2) developer need to trust the recommendation, which is easier to achieve by limiting ourselves to suggesting people that are affected by a change.

In Section IX we explored the developer view with respect to recommendation systems and if and when recommendation

on a change-set level would be appropriate. The feedback we received generally welcomed such recommendations as long as they are not seen as irrelevant, thus, corroborating Murphy and Murphy-Hill's [63] point. Developers, in fact, discuss change-sets in general, but specifically towards the end of a release cycle, as each change becomes more important with respect to the stability of the overall project.

Overall, we gave evidence to the usefulness of our approach by selecting a specific scope and outcome metric, builds and build success respectively, and defined the construction of the social and technical networks in detail (see Section II). Furthermore, we showed that the approach can generate actionable insights that are acceptable to developers. In a final study involving students from two countries we found evidence that we are both able to generate recommendation early enough to be acted upon as well as demonstrated that such recommendation could actually prevent build failures.

B. Implementation Implications

Besides investigating if developer are open to recommendation that help them prevent build failures our findings from Section IX revealed three implications for the implementation of the approach into a recommendation system.

1) Adjust intrusiveness of the notifications: Our first finding strongly suggests that a developer's information needs can dramatically change between development modes. When in normal iteration mode, developers act upon planned work and can therefore anticipate the information they need, but in endgame mode, developers react to unplanned incoming work, such as bug reports or requests for code reviews.

Many tools, such as Codebook [64] and Ensemble [65] provide information and recommendations in a fixed way. Codebook enables developers to discover other developers whose code is related. In contrast, Ensemble provides a constant stream of potentially relevant events for each developer. In the Codebook case, this might lead to extra overhead in endgame mode when developers frequently need to search for information instead being automatically provided, whereas Ensemble might overload developers during the feature development mode by providing a constant stream of information.

To avoid overwhelming or to reduce overhead further for developers, recommendation systems should either automatically adjust to the development mode or feature customizable templates that can easily be switched.

2) Group Recommendations: Our second and third findings unveiled factors that trigger developers to seek information about a change-set that are not related to its code. Instead, developers pay close attention to the experience level as well as the quality of previously delivered work to determine whether to talk to the change-set owner.

Traditional recommender systems in software engineering focus on the source code to determine useful recommendations, e.g. Codebook [64] and Ensemble [65]. This might lead to providing developers with information about changes that are of little interest due to the trust placed in more experienced developer.

But because developers often look beyond source code and perform an additional step, namely considering the change-set owner's experience and recent work, information solely created from source code might miss interesting instances where novices to the code made inappropriate changes. Recommender systems might report issues that are of less importance due to the substantial experience of the change-set owner.

Because inferring characteristics of developers can be difficult and asking developers to provide the system with characterizations that change over time, we suggest that implementing a grouping by developers and teams of developer is a good strategy to minimize noise. This will enable developer to more easily dismiss recommendations pointing to coordinating with certain developers.

3) Provide Alternative Ranking: Although the implementation implication we inferred from our second and third finding already yield an alternative ranking that groups together recommendation and ranks those groups, it is still passed on our proposed measure for ranking. The fourth finding we reported on highlights the interest of developer in the risk that is posed by a change. Our approach to generate recommendation currently does not consider there change other than to indicate technical dependencies among developers.

To better support developers in their tasks we should provide alternative rankings that take into account other activities such as risk assessment or code-reviews. Thus an implementation of the approach should not only focus on providing the information that is relevant to the recommendation but also include metrics that developer relied on beforehand, such as number of files changed or which files have been changed.

Providing developer with metrics that they are familiar with serves two purposes: (1) By providing metrics that are already used in making decision we can entice the developers to use the recommendation system and (2) by providing known metrics we build trust into the recommendation system, which as highlighted by Murphy et al [63].

XI. Threats to Validity

In his section we detail the threats to validity of this thesis.

A. External Validity

In part of this work we draw on information from an observational study (Section IX) and studies relying on development repositories (Sections V, VI, and VII) that cover two development projects. Although this limits the generalizability of the findings presented as well as the validity of the inferred approach, we think that the approach still holds merit as the studies that lay the foundation for the validity of generating insights in real time are derived from and industrial project comprising more than one hundred developers at a large software corporation. This in-depth relationship created by working together with the IBM Rational Team Concert development team limits the amount of data available for the studies we presented but this in-depth relationship enables us to better interpret the collected data as well as gain a deeper understanding of the organization and their processes and how

they influence the data. In the case of the in class study, we aimed to minimize the conclusions we drew to only serve as a feasibility study to demonstrate that technical networks can be constructed in real time as well as give some evidence that potential recommendations can prevented build failures from occurring.

In our close relationship with the IBM Rational Team Concert team we had the chance to interview ten developers, which represent a fraction of the development team at large. These ten developers were all located at the same site. As a result of this, our interview data could be biased and unrepresentative of the RTC team at large. However, we are confident that this threat is minor, due to the mix of developers we interviewed, including novices, senior developers, and team members that had been part of the group since its beginning. Furthermore, the triangulation with our observations and survey responses increases our confidence in our findings.

B. Construct Validity

In this thesis we conceptualized social dependencies among developers using digitally recorded communication artifacts in the form of work item discussions as well as relied on technical dependencies inferred from developers changing the same source code file. Both constructs are used by the software engineering research community in several studies (e.g. [22]). Nevertheless, both the social and technical dependency characterizations come with the danger that they do not necessary measure social or technical dependencies of relevance or might as well miss existing dependencies. This leads to the threat that our inferences might be based on inconsistencies in the data such as meaningless communication among developers or file changes that are not technical in nature. For instance, due to storage problems the Jazz teams erased some build results. In the case of nightly builds we expected 90 builds (according to project duration) but found only 15. This might affect our results but we argue that due to our richness of data the general trend is still preserved. Given that we use data that was generate by highly disciplined professionals or by students that we monitored we are confident that the data available for analysis is of high quality.

C. Internal Validity

Sections VI and VII demonstrated that constructing the socio-technical networks is feasible and in Section VII we showed that there is a relationship between the network configuration and build success that can be used to generate recommendations. One issue that we will need to address in future work is showing a definite link between the insights presented in Section VII and the actual build failures and to what extend the recommendations actually can prevent build failures from happening. To mitigate this threat we showed some initial evidence of tracing a failed build back to its original failure source and showed that the failure could have been prevented with the socio-technical information available at the point in time when the error was introduced into the code base.

Another threat to the approach, which is related to the previously mentioned lack of tracing the basis of the recommendations back to actual build failures, is that we did not test it in the field to see how the recommendation affect the development process. We presented in Section IX a study that explored if the recommendations are made at an appropriate level of granularity as well as feedback to the usefulness of such recommendations. Furthermore, the study conducted in a class room setting also suggests that there is value in generating such recommendations.

The surveys we deployed in our qualitative studies (Section IX) survey asked developers to answer closed questions with a pre-defined list of answers which might introduce a bias. This bias poses a threat to our findings due to the possibility that we were missing important items. We mitigated it by developing the survey iteratively by piloting and discussing it with one of the development teams to identify the most important items, and by relying on our other two sources of data to triangulate our findings.

XII. Conclusions and Future Work

We illustrated an approach to leverage the concept of social-technical congruence to generate actionable knowledge. The work we presented lends itself to several venues of future work, such as building and testing the recommendation system with several software development teams to study its impact. A more interesting avenue to pursue is to explore what software architecture can support what kind of communication and organizational structure. So far, the research around socio-technical congruence is pointing into the direction of changing how software developers coordinate their work, but we propose to return to the original observation Conway made in that the software architecture will change to accommodate the communication structures in an organization. Therefore, analyzing software architectures with respect to the project properties, such as distribution of the development team or the organizational hierarchy, might yield valuable insight in guiding design decisions of the software product that not only take into account properties to increase the feature richness or maintainability of the software product but is optimal with respect to properties of the organization and the development team in order to increase productivity and quality.

References

[1] M. Hall, Ed., *PLDI '2011: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2011, 548110.

[2] J. Knoop, Ed., *CC'11/ETAPS'11: Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software*. Berlin, Heidelberg: Springer-Verlag, 2011.

[3] T. Cortina and E. Walker, Eds., *SIGCSE 2011: Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, 2011, 457110.

[4] D. Raffo, Ed., *ICSSP '11: Proceedings of the 2011 International Conference on Software and Systems Process*. New York, NY, USA: ACM, 2011.

[5] K. Molkken and M. Jrgensen, "A Review of Surveys on Software Effort Estimation," in *Proceedings of the 2003 International Symposium on Empirical Software Engineering*, ser. ISESE 2003. Washington, DC, USA: IEEE Computer Society, 2003, pp. 223–230. [Online]. Available: http://dl.acm.org/citation.cfm?id=942801.943636

[6] B. Boehm, C. Abts, and S. Chulani, "Software Development Cost Estimation Approaches: A Survey," *Annals of Software Engineering*, vol. 10, pp. 177–205, 2000, 10.1023/A:1018991717352. [Online]. Available: http://dx.doi.org/10.1023/A:1018991717352

[7] T. Menzies, Ed., *Promise 2011: Proceedings of the 7th International Conference on Predictive Models in Software Engineering*. New York, NY, USA: ACM, 2011.

[8] S. Bassil and R. K. Keller, "Software Visualization Tools: Survey and Analysis," in *Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on*, 2001, pp. 7 –17.

[9] T. Mens and T. Tourwe, "A Survey of Software Refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126 – 139, February 2004.

[10] T. Zimmermann, V. Dallmeier, K. Halachev, and A. Zeller, "eROSE: Guiding Programmers in Eclipse," in *Proceedings of the 20th Conference on Object-Priented Programming, Systems, Languages, and Applications*. New York, NY, USA: ACM, 2005, pp. 186–187.

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

[12] 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*. New York, NY, USA: ACM, 2005, pp. 284–292.

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

[14] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining Mental Models: A Study of Developer Work Habits," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE 2006. New York, NY, USA: ACM, 2006, pp. 492–501. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134355

[15] A. J. Ko, R. DeLine, and G. Venolia, "Information Needs in Collocated Software Development Teams," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 344–353. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2007.45

[16] A. Gopal, T. Mukhopadhyay, and M. S. Krishnan, "The Role of Software Processes and Communication in Offshore Software Development," *Communication of the ACM*, vol. 45, no. 4, pp. 193–200, Apr. 2002. [Online]. Available: http://doi.acm.org/10.1145/505248.506008

[17] R. Abreu and R. Premraj, "How Developer Communication Frequency Relates to Bug Introducing Changes," in *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, ser. IWPSE-Evol 2009. New York, NY, USA: ACM, 2009, pp. 153–158. [Online]. Available: http://doi.acm.org/10.1145/1595808.1595835

[18] 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*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–11.

[19] 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*. New York, NY, USA: ACM, 2008, pp. 521–530.

[20] S. Kim, T. Zimmermann, J. Whitehead, and A. Zeller, "Predicting faults from cached history," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE 2007. Washington, DC, USA: IEEE Computer Society, 2007, pp. 489–498. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2007.66

[21] M. E. Conway, "How do Comittees Invent?" *Datamation*, vol. 14, no. 4, pp. 28–31, 1968.

[22] 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*. New York, NY, USA: ACM, 2006, pp. 353–362.

[23] S. Sawyer, "Software Development Teams," *Communication of the ACM*, vol. 47, no. 12, pp. 95–99, 2004.

[24] C. R. B. de Souza and D. F. Redmiles, "An Empirical Study of Software Developers' Management of Dependecies and Changes," in *International Conference on Software Engineering, Leipzig, Germany*, May 2008, pp. 241–250.

[25] A. H. V. D. Ven, A. L. Delbecq, and J. Richard Koenig, "Determinants of Coordination Modes within Organizations," *American Sociological Review*, vol. 41, no. 2, pp. 322–338, 1976.

[26] R. E. Kraut and L. A. Streeter, "Coordination in Software Development," *Communications of the ACM*, vol. 38, no. 3, pp. 69–81, March 1995.

[27] J. Holck and N. Jørgensen, "Continuous Integration and Quality Assurance: A Case Study of Two Open Source Projects," *Australasian Journal of Information Systems*, pp. 40–53, 2003/2004.

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

[29] M. A. Cusumano and R. W. Selby, "How Microsoft builds software," *Communication of the ACM*, vol. 40, no. 6, pp. 53–61, 1997.

[30] J. D. Herbsleb, "Global Software Engineering: The Future of Sociotechnical Coordination," in *Future of Software Engineering in conjunction with ICSE 2007, Minneapolis, USA*, 2007, pp. 188–198.

[31] D. E. Perry, N. A. Staudenmayer, and L. G. Votta, "People, Organizations, and Process Improvement," *IEEE Software*, vol. 11, no. 4, pp. 36–45, 1994.

[32] D. Redmiles, A. van der Hoek, B. Al-Ani, S. Quirk, A. Sarma, S. Filho, C. de Souza, and E. Trainer, "Continuous Coordination: A New Paradigm to Support Globally Distributed Software Development Projects," *Wirtschaftsinformatik*, vol. 49, pp. 28–38, 2007.

[33] K. Nakakoji, Y. Ye, and Y. Yamamoto, "Supporting Expertise Communication in Developer-Centered Collaborative Software Development Environments," in *Collaborative Software Enginnering*, A. Frinkelstein, J. Grundy, A. van der Hoek, I. Mistrik, and J. Whitehead, Eds. Springer-Verlag, 2010, ch. 11, pp. 219–236.

[34] J. D. Herbsleb and R. E. Grinter, "Architectures, Coordination, and Distance: Conway's Law and Beyond," *IEEE Software*, vol. 16, no. 5, pp. 63–70, 1999.

[35] C. R. B. de Souza and D. Redmiles, "The Awareness Network: To Whom Should I Display My Actions? And, Whose Actions Should I Monitor?" in *European Conference on Computer Supported Cooperative Work, Limerick, Ireland*, September 2007, pp. 325–340.

[36] J. D. Herbsleb, A. Mockus, T. A. Finholt, and R. E. Grinter, "An Empirical Study of Global Software Development: Distance and Speed," in *Proceedings of the 23rd International Conference on Software Engineering*, ser. ICSE '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 81–90. [Online]. Available: http://portal.acm.org/citation.cfm?id=381473.381481

[37] C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy, "Does Distributed Development Affect Software Quality? An Empirical Case Study of Windows Vista," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE 2009. Washington, DC, USA: IEEE Computer Society, 2009, pp. 78–88. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070510

[38] M. Cataldo and S. Nambiar, "Quality in Global Software Development Projects: A Closer Look at the Role of Distribution," in *International Conference on Global Software Engineering, Limerick, Ireland*, July 2009, pp. 163–172.

[39] T. Nguyen, T. Wolf, and D. Damian, "Global Software Development and Delay: Does Distance Still Matter?" in *Proceedings of the 3rd International Conference on Global Software Engineering (ICGSE 2008)*. IEEE Computer Society, August 2008, pp. 45–54.

[40] S. Carter, J. Mankoff, and P. Goddi, "Building Connections among Loosely Coupled Groups: Hebb's Rule at Work," *Computer Supported Cooperative Work*, vol. 13, no. 3-4, pp. 305–327, 2004.

[41] T. Niinimäki and C. Lassenius, "Experiences of Instant Messaging in Global Software Development Projects: A Multiple Case Study," *International Conference on Global Software Engineering, Bangalore, India*, pp. 55–64, July 2008.

[42] R. Frost, "Jazz and the Eclipse Way of Collaboration," *IEEE Software*, vol. 24, pp. 114–117, November 2007. [Online]. Available: http://portal.acm.org/citation.cfm?id=1308450.1309088

[43] R. Burt, *Structural Holes: The Social Structure of Competition*. Harvard University Press, August 1995.

[44] L. C. Freeman, "Centrality in Social Networks: Conceptual Clarification," *Social Networks*, vol. 1, no. 3, pp. 215–239, 1979.

[45] A. Griffin and J. R. Hauser, "Patterns of Communication Among Marketing, Engineering and Manufactoring–A Comparison Between Two New Product Teams," *Management Science*, vol. 38, no. 3, pp. 360–373, 1992.

[46] R. Reagans and E. W. Zuckerman, "Networks, Diversity, and Productivity: The Social Capital of Corporate R&D Teams," *Organization Science*, vol. 12, no. 4, pp. 502–517, 2001.

[47] 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*. New York, NY, USA: ACM, 2006, pp. 363–372.
[48] 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*. New York, NY, USA: ACM, 2006, pp. 343–352.
[49] S. Wasserman and K. Faust, *Social Network Analysis: Methods and Applications*. Cambridge University Press, November 1994.
[50] D. L. Rulke and J. Galaskiewicz, "Distribution of Knowledge, Group Network Structure, and Group Performance," *Management Science*, vol. 46, no. 5, pp. 612–625, 2000.
[51] P. A. Gloor, R. Laubacher, S. B. C. Dynes, and Y. Zhao, "Visualization of Communication Patterns in Collaborative Innovation Networks - Analysis of Some W3C Working Groups," in *Proceedings of the 12th International Conference on Information and knowledge management*, 2003, pp. 56–60.
[52] T. Zimmermann and N. Nagappan, "Predicting Defects Using Network Analysis on Dependency Graphs," in *Proceedings of the 30th International Conference on Software Engineering*. New York, NY, USA: ACM, 2008, pp. 531–540.
[53] A. Hargadon and R. I. Sutton, "Technology Brokering and Innovation in a Product Development Firm," *Administrative Science Quarterly*, vol. 42, no. 4, pp. 716–749, 1997.
[54] C. N. Gonzalez-Brambila and F. Veloso, "Social Capital in Academic Engineers," in *Portland International Center for Management of Engineering and Technology*, 5-9 Aug. 2007, pp. 2565–2572.
[55] R. M. Bell, "Predicting the Location and Number of Faults in Large Software Systems," *IEEE Transactions on Softwware Engineering*, vol. 31, no. 4, pp. 340–355, 2005.
[56] S. Kim, J. Whitehead, and Y. Zhang, "Classifying Software Changes: Clean or Buggy?" *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181–196, 2008. [Online]. Available: http://dx.doi.org/10.1109/TSE.2007.70773
[57] 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*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 189–198.
[58] T. Wolf, A. Schröter, D. Damian, L. D. Panjer, and T. H. D. Nguyen, "Mining Task-Based Social Networks to Explore Collaboration in Software Teams," *IEEE Software*, vol. 26, no. 1, pp. 58–66, 2009.
[59] A. Espinosa, S. A. Slaughter, R. E. Kraut, and J. D. Herbsleb, "Team knowledge and coordination in geographically distributed software development," *Journal of Management Information Systems*, vol. 24, no. 1, pp. 135–169, 2007.
[60] T. C. Lethbridge, S. E. Sim, and J. Singer, "Studying Software Engineers: Data Collection Techniques for Software Field Studies," *Empirical Software Engineering*, vol. 10, pp. 311–341, 2005.
[61] W. G. Lutters and C. Seaman, "The Value of War Stories in Debunking the Myths of Documentation in Software Maintenance," *Information and Software Technology*, vol. 49, no. 6, pp. 576–587, 2007.
[62] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 3rd ed. Springer, July 2003.
[63] G. C. Murphy and E. Murphy-Hill, "What is Trust in a Recommender for Software Development?" in *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, ser. RSSE 2010. New York, NY, USA: ACM, 2010, pp. 57–58. [Online]. Available: http://doi.acm.org/10.1145/1808920.1808934
[64] A. Begel, Y. P. Khoo, and T. Zimmermann, "Codebook: Discovering and Exploiting Relationships in Software Repositories," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE 2010. New York, NY, USA: ACM, May 2010, pp. 125–134. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806821
[65] P. F. Xiang, A. T. T. Ying, P. Cheng, Y. B. Dang, K. Ehrlich, M. E. Helander, P. M. Matchen, A. Empere, P. L. Tarr, C. Williams, and S. X. Yang, "Ensemble: A Recommendation Tool for Promoting Communication in Software Teams," in *Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering*, ser. RSSE '08. New York, NY, USA: ACM, 2008, pp. 2:1–2:1. [Online]. Available: http://doi.acm.org/10.1145/1454247.1454259

PLACE PHOTO HERE

Adrian Schröter Biography text here.

PLACE PHOTO HERE

Daniela Damian Biography text here.