

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.

In this thesis 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 well as a large student project at the University of Victoria, Canada, and Aalto University, Finland.

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 USD¹. 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 educational 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. Therefore the two guiding research questions we address in this thesis to investigate whether socio-technical congruence can be used to generate actionable knowledge that can increase build success are:

RQ 1: Does Socio-Technical Congruence influence build success?

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

RQ 2: Can Socio-Technical Networks be leverage to generate recommendations to improve build success?

We are using a mixed methods approach to explore these two research questions. For **RQ 1** we employ data mining techniques by studying the artifacts such as task discussions and source code changes of a large industrial software project. The second research question (**RQ 2**) requires both quantitative and qualitative analysis methods. To find statistically relevant recommendations we employ data mining techniques, but to explore the usefulness and acceptance of such recommendations we make use of questionnaires, interviews, and observational studies.

II. PROBLEM STATEMENT

Socio-technical congruence as defined by Catalto et al [22], describes a measure that outlines how much the technical dependencies in the product are matched by social interactions among developers affected by these technical dependencies. This directly follows from Conway's observations [21] that the communication structure of any given organization dictate the underlying technical dependencies. In software engineering that roughly translates into the idea that the communication flow within software teams need to match the module dependencies described by the software architecture.

This idea shows great promise when applying it to software repositories such as versioning archives and issue trackers or other recorded communication. Cataldo et al [22], [23] as well as other researchers [24], [25] found that the better the satisfaction of the technical dependencies with social interaction is, the higher productivity and to some extent software quality [26]–[28] becomes. The ability to extract useful socio-technical measures from archives in an automated fashion enables the application to any software project that captures development data electronically.

However, we see three major issues with the concept of socio-technical congruence as it is currently used:

- The socio-technical congruence measure itself does not give much indication with respect to how to improve the over all situation other than to suggest people to talk to each other in case they share a technical dependency.
- The idea of achieving high congruence is based on the notion that it is important to communicate along all technical dependencies, which is not necessarily true.
- The analysis of socio-technical congruence can only be done post-mortem, which although valuable in a retrospective does not help in improving productivity or quality in an ongoing project.

The issue of imbalance between technical and social relationships between developers is related to the problem of not knowing how to improve the socio-technical congruence other than by pointing out the technical relationships between developers that did not communicate with each other. Given enough resources and time every technical dependency can be satisfied but this might run the risk of decreasing the productivity by introducing to many interruptions.

Over-communication of technical dependencies might arise from the underlying assumption that every technical dependency warrants the dependent developers to communicate

with each other. We are not solely referring to the ability of developers to read environment traces [29] but also to the fact that some changes are either not meant to be communicated or that the system architecture was designed to accommodate certain changes (think of optimizations) that should not affect other developers.

To fully leverage the concept of socio-technical congruence it is important to act on it. The current concept is only shown to relate to performance and quality post-mortem. To truly unlock the potential of the socio-technical congruence concept it needs to be extended such that it can make on demand recommendations to improve congruence.

III. BACKGROUND

In this chapter we provide an overview of five areas that are relevant to the research conducted with this thesis: (1) the research on software builds, (2) the research on coordination in software development teams (3) the research around the concept of socio-technical congruence, (4) failure prediction using social networks, and (5) recommender systems in software engineering.

A. Build Outcome

Although software builds are important to delivering a software product as the final product is just the latest acceptable build, research in software builds focuses mainly on tools and processes that support the build process. Software products supporting builds often intend to speed up the build process and the execution of all test cases to obtain an assessment of the quality of the build [32]. Similarly, processes that focus on supporting software builds are predominantly dealing with issues of obtaining all required code changes from the different development teams and integrating this code into a final build as fast as possible without introducing additional issues.

The issue that shifts into focus once the actual process of creating the build is thoroughly optimized is to gain an idea of whether a build will fail or succeed before the build process is started. If a project reaches a certain size, meaning the test suite grows considerably in size, the build process can take several hours just to run the whole test suite. To determine whether developers need to stay in order to apply quick fixes such that the product can be shipped or handed over to a team starting their work in a different time zone becomes important.

Following we review literature with respect to coordination and integration with builds representing a form of integration (Section III-A1). We compliment that review with research conducted the effect of social networks on software development.

1) *Communication, Coordination and Integration*: The relationship between communication, coordination and project outcome has been studied for a long time in the area of computer-supported cooperative work. More recently the domain of software and distributed software development showed increased interest as well.

Communication plays an important role in work groups with high coordination needs and the quality of communication has been found as determinant of project success [33], [34]. The

dynamic nature of work dependencies in software development makes collaboration highly volatile [35], consequently affecting a team's ability to effectively communicate and coordinate. Additional difficulties emerge in distributed teams, where team membership and work dependencies become even more invisible [36]. Moreover, team communication patterns are significantly affected by distance [37]. Maintaining awareness [38] becomes even more difficult when developers work in geographically remote environments; communication structures that include key contact people at each site are effective coordination strategies when maintaining personal cross-site relationships is challenging [37].

With respect to the role of effective coordination in project success, early studies indicate the issues that software development teams face in large projects [33]. A study by Herbsleb et al [39] showed that Conway's law is also applicable for the coordination within development teams, supporting the influence of coordination on software projects. Kraut et al [34] showed that software projects are greatly influenced by the quality of coordination of development teams. More recently a theory of coordination has been proposed and accounts for the influence of coordination on different project metrics such as rework and defects [40].

The importance of communication in successful coordination is also well documented and makes the study of communication structures important. For example, Fussell et al [41] found that communication amount and tactics were linked to the ability of effectively coordinate in work groups. In software development, others showed that communication problems lead to problems during the activity of subsystem integration [42], [43]. Coordination conceptualized via communication has also been studied more generally in relation to project success: factors such as "harmony" [44], communication structure [45], and communication frequency [46] were related to project success.

The difficulty in studying failed integration in relation to communication lies in capturing and quantifying information about communication in teams that have a well-defined coordination goal but dynamic patterns of interaction. In our work we use the Jazz project data, which captures communication of project participants. This enables us to study the structure of the communication networks emerged around code integrations, both at individual teams of the project and within the entire project.

2) *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 [47], [48]. Griffin and Hauser [46] 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 [49] 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. [37], [50]) and a number of common measures of communication structure include network density,

centrality and structural holes [48], [51].

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 [52]. Density has been studied, for example, in relation to coordination ease [37], coordination capability [50] and enhanced group identification [49].

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 [53]. Similarly, Hossain et al [50] 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. [50], [53], [54]).

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 [47]. 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 [55], 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. [56]).

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], [54], [57], [58]). And only few studies, such as Hassan and Zhang [59], stepped away from predicting component failures and used statistical classifiers to predict integration outcome. In this thesis 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.

B. Coordination in Software Engineering Teams

In the previous Section III-A1 we highlighted the connection between coordination and interaction. In this section we extend this review by discussing work about coordination in software teams, as it is important to understand the coordination in teams to be able to manipulate it to influence build outcome.

1) *The Need for Coordination:* Software is extremely complex because of the sheer number of dependencies [60]. 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 [61]. 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” [62]. 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 [34].

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 [36], [63] and destabilizes the product [64]. 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 [65], leading us to the foundation of socio-technical congruence.

2) *Coordination in Software Teams*: Research in software-engineering coordination has examined interactions among software developers [66], [67], how they acquire knowledge [68], [69], and how they cope with issues including geographical separation [70], [71]. The ability to coordinate has been shown as an influential factor in customer satisfaction [34] and improves the capability to produce quality work [72].

Software developers spend much of their time communicating [73]. Because developers face problems when integrating different components from heterogeneous environments [74], developers engage in direct or indirect communication, either to coordinate their activities, or to acquire knowledge of a particular aspect of the software [69]. Herbsleb, et al examined the influence of coordination on integrating software modules through interviews [75], and found that processes, as well as the willingness to communicate directly, helped teams integrate software. De Souza et al [76] 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 [77]. Studies of Microsoft [19], [78] 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 [79].

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 [80]. Technology that empowers distributed collaboration include

topic recommendations [66] and instant messaging [81]. Processes are adapting to the fast pace of software development: the Eclipse way [82] 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 it in this thesis.

C. Socio-Technical Congruence

As mentioned earlier this thesis explores to what extent we can leverage the concept of socio-technical congruence. Before we discuss the work that conducted with respect to using the concept of socio-technical congruence to analyze software development teams and their performance, we explain the socio-technical congruence concept.

1) *Socio-Technical Congruence Definitions*: The literature exploring and using the concept of socio-technical congruence often relies on two interconnected definitions of socio-technical congruence. Originally defined by Cataldo et al [22] socio-technical congruence was a single metric describing how much of the work dependencies between developers are covered by the communication between those developers. But the interest in socio-technical congruence took a broader view and instead of focusing on the metric the focus shifted to the underlying construct conceptualizing the different connections among developers. Following we discuss the two commonly used approaches to infer socio-technical dependencies among developers, starting with the traditional definition initially presented by Cataldo et al [22] followed by a more network centric definition.

a) *Task Assignment and Dependency*: Cataldo et al [22] defined technical dependencies among developers as the matrix multiplication of the matrix defining the assignment of a developer to a task with the matrix defining the dependencies among tasks multiplied with the inverse of the matrix defining the assignment of a developer to a task. Thus two matrices need to be inferred from a data set: (1) task assignment matrix describing which developer is assigned to what task and (2) the task dependency matrix describing which tasks share dependencies.

Task Assignment Matrix. The task assignment matrix dimension is the number of developers times the number of tasks. Each entry in the matrix denotes whether a given developer is assigned to a given task, note that this notation allows for more than one developer to be assigned to a task as well as one developer being assigned to multiple tasks. This information is inferred from task management systems such as BugZilla² or Jira³ that show who is assigned to work on a given task.

Task Dependency Matrix. The task dependency matrix dimension is the number of tasks times the number of tasks with each row and column representing all tasks. Each entry in the task dependency matrix indicated whether two tasks have a dependency, note that non-zero entries refer to the existence of a dependency but not its strength. The task dependency matrix is populated by identifying the code written to finish a

²<http://www.bugzilla.org>

³<http://www.atlassian.com/software/jira>

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 3 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 3 & 0 & 1 & 2 \end{pmatrix}$$

Fig. 1: Calculating technical dependencies among developer using the task assignment and task dependency matrix.

task and infer dependencies among the various code changes implementing different tasks. For example, Cataldo et al [22] defined two tasks to be dependent if the associated changes modify the same file.

The final calculation of the technical dependency among developer follows the formula presented below:

$$\text{Task Assignment} \times \text{Task Dependency} \times \text{Task Assignment}^T = \text{Coordination Needs} \quad (1)$$

Figure 1 shows an example on how to derive the technical dependencies among developers given a task assignment and task dependency matrix. Following the formula presented in Equation 1, we multiply the task assignment matrix with the task dependency matrix with the transposed task assignment matrix to obtain a matrix of dimension of number of developers by number of developer with each entry in the matrix greater than zero denoting a technical dependency between two developers. This resulting matrix is also referred to as the coordination needs matrix.

The technical dependency matrix obtained through the matrix multiplication described needs to be contrasted with the actual coordination that happened in the project. For this purpose Cataldo et al [22] proposed to create a matrix recording whether two developers coordinate their work. Note that communication is often [18], [22], [24]–[26], [83] used as a proxy for coordination allowing to rely on recorded communications found in email archives or task discussions in issue management systems. The congruence metric itself is the ratio between developers that have both a technical dependency and did coordinate over the number of developers that have a technical dependency.

The actual coordination matrix depicts a social network with developer being nodes and coordination instances edges. Similarly the coordination needs matrix depicts a social network connecting developers when they share a technical dependency. Thus another method to approach socio-technical congruence instead through the explicit definition of the task assignment and the task dependency matrix is to take a more social networks analysis point of view and construct the two types of social networks directly as we discuss in the next section.

b) Social and Technical Networks: Since the task dependency matrix as we saw earlier depends on the changes made to the software and their dependencies through the code it is often easier to directly construct the coordination needs matrix or for that matter the social network connecting developer via technical dependencies from the changes made to the system. This is possible since changes to a software system are usually recorded in a source code repository and each change belongs to a developer. Thus research [22], [24]–[26], [83] working with the socio-technical congruence concept with a social network view contrast social and technical networks.

Technical Networks. In Cataldo et al’s [22] formulation of technical dependencies they infer them by multiplying the task assignment and task dependency matrix. Since the task dependency matrix is inferred from the overlap in code modifications, say both tasks are accomplished by modifying the same source code files, the technical dependencies among developers can be directly inferred from a software repository. This more direct approach enables the construction of technical networks, connecting developers through the dependencies of the changes they made to a software project, without the need of accessing a task management system.

Social Networks. The social network representation of the ongoing communication is exactly the same as the actual coordination matrix as described by Cataldo et al’s [22] as the matrix is in effect a way to represent a network (also known as adjacency matrix).

The technical difficulties in this approach is to match the social and technical networks as the usernames used for code repositories and task management can be different, this is especially an issue with open source development as they are less governed by processes demanding naming conventions of account names [13].

2) Socio-Technical Congruence and Performance: Social-technical congruence as originally observed by Conway [21] states that any product developed by an organization will inevitably mirror the organization’s communication structure. From this starting point Cataldo et al [22] as well as other researchers [24], [25], [83] investigated whether the lack of this reflection relates to changes in productivity by investigating the overlap of communication among developers and their technical dependencies. The communication among developers represents the organizational communication structure whereas the technical dependencies between the work each developer represents the products organization. If the communication structure completely contains the work dependencies among developers, then developers accomplish their tasks faster for reasons that are mainly due to knowledge seeking and sharing [84]. For example, a developer can better accomplish her task if she is talking directly to co-workers that need to modify related code to avoid failures or because someone can help her understand the impact the code she is about to modify better.

The main performance criteria research investigated to measure the effect of socio-technical congruence is task completion time. For this purpose Cataldo et al [22] measures the congruence on a task basis and test for the correlation between congruence the metric with the time it took to resolve the task. Overall Cataldo et al [22] found that there is a statistically significant relation between the amount of congruence and a tasks resolution time, which was confirmed by other studies [24], [25].

D. Networks and Failure

Because we are investigating how to improve communication among software developers following their technical dependencies among each other we give an overview over work that involves work on source code that directly or indirectly indicates technical dependencies.

1) *Artifact Networks*: Using dependencies within a product one can construct a network of software artifacts that are connected via those dependencies. Artifacts that have direct dependencies in the case of source code referred to as code peers. One interesting property of code peers is that in case a code peer exhibits a defect it increases the likelihood that the code artifact whose peer contains a defect have a higher likelihood to contain a defect by itself [85].

From the notion of a code peer and its influence on other peers can the idea of analyzing these network with respect to an artifact and its surrounding artifacts be derived. In a first study Zimmermann et al [54] analyzed call dependencies of a single artifact and found measures characterizing those dependencies to be a good predictor for software defects.

In a follow up study Zimmermann et al [86] extended the influence of an artifacts peer by not solely focusing on an artifacts dependencies to its peers but taking into account the dependencies among an artifacts peers. This enables the application of network measures and social-network measures to characterize this ego network constructed around a software artifacts. As it turn out, the predictive power of such a network is stronger than only considering dependencies between an artifact and its peers [86].

2) *Technical Networks*: To go from artifact network to technical network developers can be included in the already existing artifact network and thus be represented as a kind of artifact [87]. These two mode networks can be used for the same analysis that Zimmermann et al [54], [86] performed by focusing on the software artifacts to predict the failure likelihood of each. Meneely et al [88] uses networks that consist only of developers that within a given release modified the same file. Social network measures extracted from these networks are able to predict whether a file contains a failure.

E. Recommendations in Software Engineering

In the software engineering community knowledge extracted from software repositories are usually brought to developers in the form of recommender systems. Since the goal of this thesis is to create an approach forming the basis for a recommender system, we present recommender systems using the socio-technical congruence concept. Several recommender systems derived from the implication of socio-technical congruence described by Conway's Law [21] provide additional awareness to improve coordination among software development especially in a distributed setting where coordination is most difficult [89]. In the following we describe five such awareness systems. We are aware that this list is not exhaustive. Nonetheless, we think this list presents a reasonable overview of awareness systems proposed by software engineering researchers.

Ariadne [90] provides awareness to developers by showing call dependencies between code a developer is working on and the code that she is potentially affecting. This allows a developer to see which other developer she might need to coordinate her work with to not negatively impact that developer's code.

Palantir [91] complements the dependencies among developers by providing the reverse awareness showing a developer

what source code she is currently accessing in their workspace is affected by code changes submitted by co-workers. For example, Palantir indicates which source code files have been changed in the mean time by other developers that are present in the developer's current work space and thus might hint at possible merge conflicts.

Tesseract [92] extends the concept of showing code dependencies among developers by fostering awareness through visualizing task and developer centric socio-technical networks, thus extending the networks underlying Ariadne and Palantir by a social component. A task centric socio-technical network is build from all developers and source code changes that are related through code dependencies or task discussions. These task centric socio-technical networks are complemented by developer centric networks, that show for a specific developer what social, technical, or socio-technical relationships she has with her colleagues.

Ariadne, Plantir, and Tesseract suffer from the issue that they cannot provide real time feedback on changes in technical networks, as they solely rely changes committed to the source code repository. *Proxiscientia* [93] address this issue by implementing an approach proposed by Blincoe et al [30] to instrument IDE's used by software developers and gather code edit events as recorded by tools such as Mylyn [94]. This enables a developer to be forewarned of changes that are made to related code as for example Palantir relies on.

Ensemble [95] provides a constant stream of events consisting of modifications to artifacts that are related to the stream owner. If developer Adam posts a comment on a task owned by developer Eve, then Eve's stream would contain an event showing that Adam commented on her task. Similarly, the stream of a developer also contains information about relevant code modifications that overlap or potentially interact with code she previously modified.

Overall all these recommender systems provide awareness of who might be worth to interact with. None of those systems are aiming at a concrete goal to accomplish other than achieving awareness. We think that a focus is needed, such as on awareness with respect to dependencies that are relevant for build success. Without such a focus the information that a developer needs to survey can quickly take up to much precious development time and may lead a developer to abandon those systems as they are taking up more time than they save.

F. Next Steps/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 other things in this thesis 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 devel-

oper 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. Since many products are not developed from scratch and because architectural changes once development has been going on for a number of months are costly and time consuming [96], we aim at generating recommendations to change the actual coordination to improve the socio-technical network where it matters. Therefore, 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 in the second part of this thesis with investigating the influence of communication among team members in the form of social networks on build success. Next, we investigate if gaps (unfilled coordination needs) between developers as highlighted by socio-technical networks and the socio-technical networks themselves can be brought into relation with build success. Therefore Chapter V and VI investigate the following two research questions respectively:

- **RQ 1.1:** Do Social Networks influence build success? (Chapter V)
- **RQ 1.2:** Does Socio-Technical Networks influence build success? (Chapter 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 (Chapter VII). The third and final part of this thesis focuses on evaluating this approach in three ways: (1) gathering general statistical evidence that parts of the network can be manipulated to increase build success, (2) exploring the acceptance of such recommendation based on those manipulations by developers, and (3) a proof of concept that the recommendation could prevent failures. Hence, the first three chapters of the third part of this thesis are guided by the following three research questions:

- **RQ 2.1:** Can Socio-Technical Networks be manipulated to increase build success? (Chapter VIII)
- **RQ 2.2:** Do developers accept recommendations based on software changes to increase build success? (Chapter IX)
- **RQ 2.3:** Can recommendations actually prevent build failures? (Chapter ??)

In the following discussion (Chapter X) we will highlight how our findings from these three research questions support the approach we detailed in Chapter VII.

IV. METHODOLOGY AND CONSTRUCTS

Before we dive into our actual studies of the effect of socio-technical congruence and its use to form recommendations, we present the overall roadmap for this thesis (Section IV-A) and some common definitions (Section IV-B) and constructs

(Section IV-C) that we use for the remainder of the thesis. Furthermore, we also will discuss the general approach to the data collection methods employed (Section IV-D).

A. Methodology Roadmap

In this section we discuss the methods we apply to answer the research questions presented in Chapter III. Figure ?? depicts the relationship between the research questions and the contribution of this thesis, an approach to improve social interactions among developers by characterizing the quality of interactions by the build outcome of the related build. Research questions 1.1 and 1.2 discussed in Chapters V and VI motivate the approach. Research questions 2.1 and 2.3 (Chapters VIII and ??) explore whether socio-technical networks can be used to form recommendations that can prevent build failures, whereas research question 2.2 inquires in Chapter IX whether such recommendations are acceptable by developers.

1) Research Question 1.1:

- **RQ 1.1:** Do Social Networks influence build success? (Chapter V)

The thesis goal is to design an approach that is able to improve the social interactions in the form of communication among software developers. As a first step we need to establish if the communication among software developers has an influence on the build success.

We have access to the development repositories used by the IBM Rational Team Concert development team such as their source code management system, communication repositories in the form of work item discussions, and their build results. All these artifacts are linked together in a way that we can trace from the build result which changes went into the build and which work items a change is meant to implement.

Using this information we can construct social networks from all the work items that are related to builds. These networks are then described using social network metric and form the input for machine learning algorithms to predict whether a build based on these metrics is more likely to fail or succeed.

Via this machine learning approach we want to establish a connection between a build's social network and its outcome. If we are able to predict the build outcome more accurately than by simply guessing, using the likelihood for a build failure, we demonstrated that there is a statistical relationship between build outcome and social networks. Thus, this result forms the first evidence that manipulating the social network might yield a positive effect on build success.

2) Research Question 1.2:

- **RQ 1.2:** Does Socio-Technical Networks influence build success? (Chapter VI)

Knowing that a social network can influence the success of the corresponding build leads us to the question of how networks should be manipulated to improve the likelihood for a build to succeed. Therefore, we explore the relationship between socio-technical networks generally and gaps within that networks, with a gap being formed by two developers that share a technical dependencies but did not communicate about work related to the build of interest, and build success.

Similarly to the previous research question, we base the analysis on the same data set, as it allows us to directly infer technical relationships among developers related to a software build from the changes submitted to the source code management tool. We used these changes previously to infer the work items developers used to communicate among each other about the build.

Since the socio-technical networks has two semantically different edges connecting two developer within a network (technical dependencies and communication among developers) we refrain from using social network metrics as they assume only one mode of connection among nodes within a network. Instead we investigate the relationship of the socio-technical congruence index and build success as well as focusing on the influence of gaps in the network on build success.

Via statistical analysis methods such as regression analysis we want to establish a relationship between the existence of gaps within the socio-technical network and build success. By addressing this research question we obtain another piece of evidence that let us formulate an approach to recommend actions to increase build success that are specifically alleviating gaps within the socio-technical network by recommending developer to communicate.

3) *Research Question 2.1:*

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

The previous two research questions enable us to formulate an approach to generate recommendations that are meant to foster communication among developers in order to increase build success. This leads to the next step, namely, to explore whether this approach can generate recommendations that bear a statistical relationship to build success.

With the same data source we used to explore the previous two research questions, we try to relate individual reoccurring gaps in socio-technical networks to build failure. Knowing those gaps, or developers that frequently share a technical dependency without communicating with respect to a build that failed we check if adding a social dependency would change the likelihood of a build to fail. We expect to find a number of gaps that when mitigated increase the likelihood of build success.

4) *Research Question 2.2:*

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

Before exploring if the recommendation hold actual value with preventing build failures, we explore if developers would welcome recommendations with respect to changes. Therefore, we joined the development effort of one of the Rational Team Concert development teams as participant observer to get an insight into how the actual developer communicate during their day to day work.

We complement these observations with followup interviews to gain a better understanding of the team dynamics and their discussion topics, since as a project newcomer our work is limited to more basic tasks in contrast to higher level decision making. To extend our reach beyond the local team

we deployed a questionnaire with the product team at large to gain a better understanding whether the recommendations we would supply can easily be integrated in their typical discussions with fellow developers.

This study should give us a better understanding of whether developers are discussing individual changes, thus justifying the appropriateness of the level of recommendations. Furthermore, we expect to uncover general suggestions on when and how to supply such recommendations as developers might not always be interested in individual changes even if they pose a threat to build success.

5) *Research Question 2.3:*

- **RQ 2.3:** Can recommendations actually prevent build failures? (Chapter ??)

We conclude our evaluation of our proposed approach with a proof of concept in a more controlled environment to ascertain whether the recommendation we can generate actually help in preventing builds from failing. As root cause analysis of failures can be very tedious and due to the various reasons a build can fail that are not necessarily due to software changes we depart from studying the development of Rational Team Concert and observe students extending an open source project during a course taught at the University of Victoria, Canada, and Aalto University, Finland.

This different setting allows us to ask the study participants to use tools we developed to collect more fine grained data on their development efforts, such as when they edited which parts of the source code. Additionally, we have access to the actual source code repositories such as source code management, issue tracking, email, and text chat sessions. Furthermore, we ask the students to answer regular questionnaires and keep a development diary that are meant to collect information on issues that the students encounter during the course and are especially related to their development effort.

Analyzing the collected data from the questionnaires and development diaries is done manually, by reading and annotating the different responses and entries to uncover build issues that were important to the students. Knowing the build issues the students encountered, we select the most severe in terms on the number of reports and trace them to the actually root cause if possible by identifying the code changes that causes them (if applicable) and continuing to analyze the offending changes and identifying corresponding communication.

We expect to find one representative example of a failed build that the students encounter during the course that can be resolved using recommendations based on information that is available before the students actually incorporate their modifications in to the source code repository of the project.

B. Definitions

We start with defining three constructs that we are heavily relying upon: (1) Work Item being a complete unit of work, (2) Change-set being the technical work submitted by a developer, and (3) builds referring to a testable product.

1) *Work Item:* A *Work Item* is a unit of work that can be assigned to a single developer. A unit of work can span anything from being a bug-report, reported by either the end

user or by a developer, to a feature implementation. *Work Items* can be hierarchically organized to show the work break-down from high-level requests to manageable pieces. One project team member is responsible for a work item to be completed, the sub work items that it is broken-down to do not necessarily need to be assigned to the owner of the parent work item.

For instance, in the case of the IBM Rational Team Concert development team creates story items to describe larger functionality from the user point of view and assigns them depending on the complexity and implication of the story. The owner of the story then either breaks down the story into multiple stories or tasks which are again assigned to team members according to their complexity and implications. Once the work item level is sufficiently low the developer assigned to it can make the necessary modifications to the project to accomplish the work detailed in the work item.

2) *Change-Set*: A *Change-Set* is a set of source code changes applied to a number of source code files, with a file being the artifact that a developer would change to add to, modify, or delete from the current product. The developer that applied those changes to the product bundles them into one or multiple change-sets. For example, in the Eclipse project⁴ the developers use CVS⁵ as their version control system to manage changes to the Eclipse IDE. A developer will check out her current version of the repository and started editing, creating and deleting files in order to fulfill a work item she is currently working on. Once the developer decides that she accomplished the work to finish her current work item, she commits her changes. Those changes that consists of file creations, deletions, and modifications taken together are referred to as a *Change-Set*.

3) *Build*: The goal of each software development team is to deliver a finished or improved product at some point in time. This finished product is often referred to as the final *Build*. A *Build* can generally be referred to as any instance of the product that can be run to some extend. To create a build a team gathers all the changes implementing work items that are required for the new build and compiles and packages the product. The amount of work items and their respective changes included in a build will gradually increase over time because more work will be finished as the project progresses.

In the case of the IBM Rational Team Concert development team, they create builds on a frequent basis to test the product as a whole to look for integration issues. The team also subscribes to the philosophy to use their own products themselves and thus try to bring each build to a level that it can be used for development. This intense use enables the team to spot issues that are still within the product and enables them to asses the severity of those found issues.

C. Constructs

From the definitions introduced previously we can derive the three central constructs we work with in this thesis: (1) the social network connecting communicating and coordinating developers, (2) the technical network connecting developer

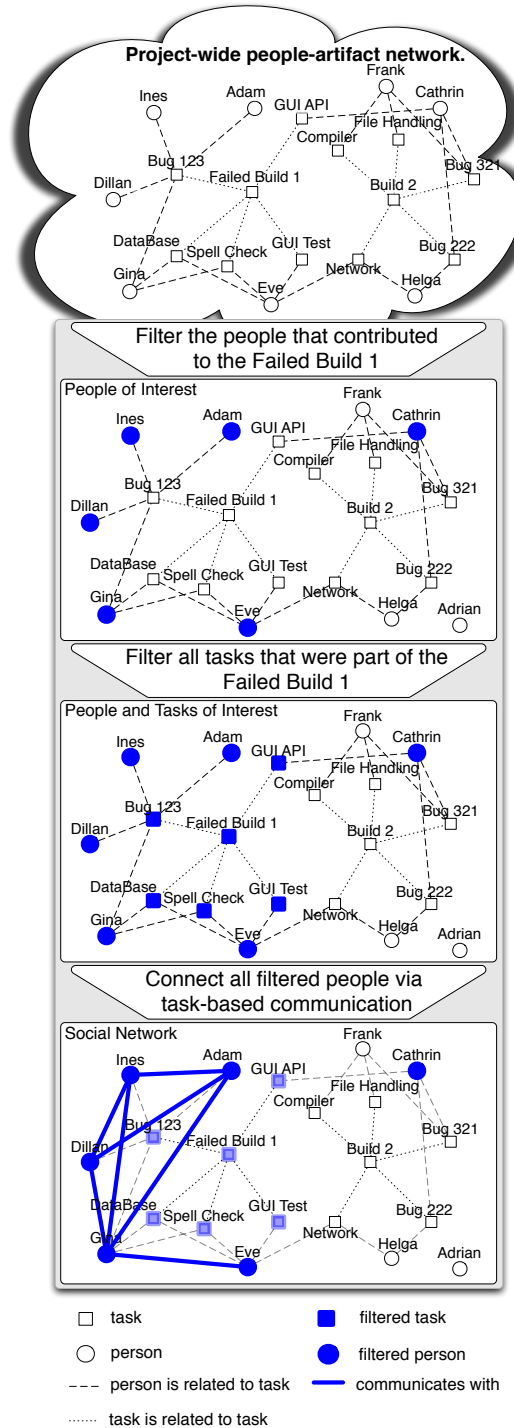


Fig. 2: Social network construction examples in our approach

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 chapters that are mining the repository provided by the Rational Team Concert development team (Chapters V, VI, and VIII).

1) *Social Network*: To illustrate our approach to construct social networks we go through the example of a failed build illustrated in Figure 2. A social network is represented as

⁴<http://www.eclipse.org>

⁵<http://www.nongnu.org/cvs>

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.

We construct and analyze social networks within a collaboration scope of interest. With a collaboration scope defining the people and interactions of interest. In this example, around Failed Build 1, the collaboration scope is the communication of the contributors to the failed build. Other examples include the collaboration of people working on a critical task, in a particular geographical location, or in a functional team such as testing.

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** are people who work on the software project. These project members can be developers, testers, project managers, requirements analysts, or clients. Project members, such as Cathrin and Eve, become nodes in the social network.
- **Work Items** are units of work (as defined earlier) within the project that may create a need to collaborate and communicate. Examples for a work item include resolving Bug 123 or implementing the GUI API. More generally, implementing feature requests and requirements can also be considered collaborative tasks.
- **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. In our example, dashed black lines represent task-related communication such as a comment on Bug 123, or an email or chat message about GUI API. Task-related communication is used to create the edges between developers in the social networks.

The data underlying the social network used throughout this thesis are based on work items and their associated discussions. In IBM Rational Team Concert each work item has an attached discussion thread where developers can discuss the work item or simply note down their thoughts while working on the work item. This means, we would create a link between two developer if they comment together on the same work item to indicate that they are part of the same discussion. Note that this sections draws heavily from our work done in collaboration with Timo Wolf, Daniela Damian, Lucas Panjer, and Thanh Nguyen [97].

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

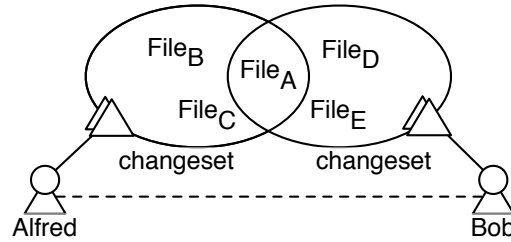


Fig. 3: Creating a technical network by connecting developers that changed the same file.

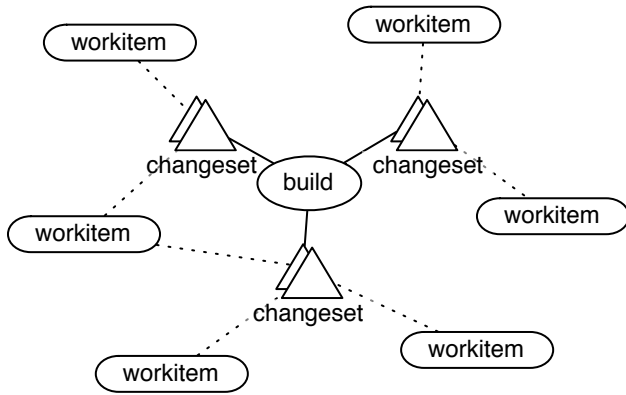
link developer with each other, namely if they are modifying related source code artifacts. As in the previous network construction, the construction of the technical network is based on three components:

- **Project Members** are people who work on the software project. These project members can be developers, testers, project managers, requirements analysts, or clients or in general anyone that modifies software artifacts through change-sets. In Figure 3 project members, such as Alfred and Bob, become nodes in the technical network because they modified the same file.
- **Change-Sets** are changes made to software artifacts by individual users. A set consists of a number of artifacts that have been modified as well as the modifications themselves. For example Alfred as shown in Figure 3 modified File_A and File_B.
- **Software Artifact Relation** describe the relation between developers in the technical networks. These relationship can be defined in several different ways. For example, in Figure 3 Alfred and Bob are related through a technical relationship because they modified the same file. Note, that we are mostly interested in relationships between artifacts that are effected by changes.

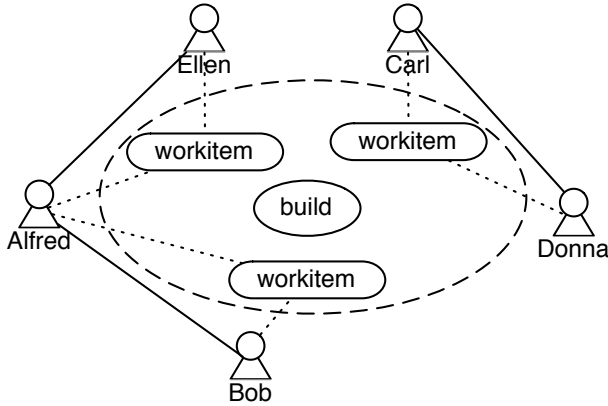
Constructing technical networks therefore follows three steps: (1) we gather all change-sets of interest, (2) identify the relations between artifacts, (3) infer from the change-sets and the relations between the source code artifacts the relation between the artifact owners. For example, after we selected the set of change-sets of interest we define the change-sets themselves as the source code artifact and identify the owners of those artifacts. Then we infer the relationship between those source code artifacts by relating all change-sets that changes the same source code file. And as Figure 3 shows this means in the case for Alfred and Bob that they are connected because both own a change-set that modifies the same file.

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:

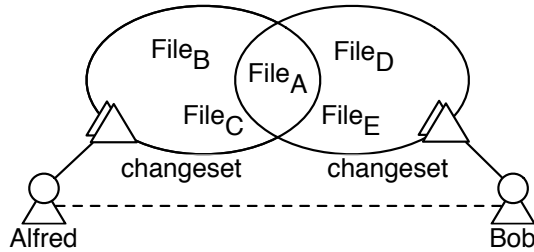
- 1) **Selecting the Focus** used for the socio-technical network represents the glue that binds the social and tech-



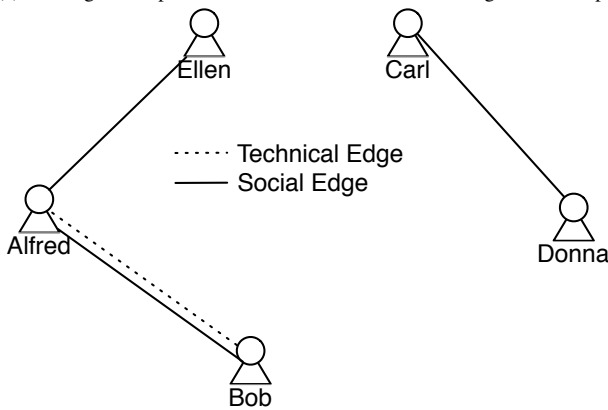
(a) Inferring to the build focus relevant change-sets and work items.



(b) Constructing a 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. 4: Constructing socio-technical networks from the repository provided by the IBM Rational Team Concert development team.

nical network into a socio-technical network. This focus also referred to as filter in our earlier publication [97], determines the content of the networks. In this thesis we use for the focus of such networks are software builds to construct networks that are describing the coordination among developer for a given software build.

- 2) **Constructing the Social Network** follows the description above with the focus determining the work items that are selected to generate the nodes from the work item participants and the edges from the communication among the participants through a work item.
- 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** is the final step that overlays the networks by unifying the set of developers in both networks. Thus a pair of developers can be directly connected with each other through two edges, one representing the edge from the technical network and the other the edge from the social network.

Figure 4 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 4a) 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 4b). 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 4c). Finally, the two networks are combined and yield the socio-technical network shown in Figure 4d.

D. Data Collection Methods

To conduct the research for this thesis 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 or logs of IRC⁶ or instant messenger chats.

Repositories can grow to considerable sizes depending on the projects live span and intensity. Therefore, it is often infeasible to manually review the history of a project and it is necessary to employ data-mining techniques to be able to analyze trends. Although this approach is limited in gaining

⁶http://en.wikipedia.org/wiki/Internet_Relay_Chat last visited June 8th, 2012

a deeper understanding of the intricacies of a development project, it nevertheless is invaluable to place in-depth results in the bigger picture of the project. Furthermore, data mining approaches are one way to easily give back value to the development team without burdening any individual developer with diverting time to other non-automatic data collection instruments. This is an important point as one goal of this thesis is to explore the ability of the concept of socio-technical congruence to generate actionable recommendations. In case a developer needs to personally provide a large amount of information manually the overhead generated by a system might outweigh the benefit of recommendations and therefore make the system useless.

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 4a 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. Similarly these links can be inferred as proposed by Cubranic et al [98] from repositories that are missing the capabilities of creating formalized links. We used repository mining techniques in Chapter V, VI, and VIII to explore the rich repositories provided by the IBM Rational Team Concert team.

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 offering prepared answers. 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.

We both deployed surveys working with the Rational Team Concert development team (Chapter IX) and when working with students working on a large course project at the University of Victoria, Canada, and Aalto University, Finland (Chapter ??).

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 [70] in order to effectively and clearly communicate. Understanding this vocabulary as an outside is not necessarily easy but very important to make sense of comments and answers supplied in interviews and surveys and of course to better interpret observations.
- **Understanding the Context.** For example, in one study our observation period coincided with the months prior to a major release and during which the team focused on extensive testing rather than new feature development. 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. Although it is easy to ascertain when the next major release of the product is, the effect this has on the developer besides a change in the process is harder to gauge. Being part of the development effort allowed us to better understand how developers react to the change in process and better understand their struggles.
- **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. Better in the sense of both more meaningful to the developer and in the sense of understandability because they can be phrased using the project vocabulary.

Besides gaining a better understanding of some easily missed or miss-understood intricacies, working together with the developer establishes a trust relationship [99]. This trust helps to mitigate observation biases that are introduced by just observing as well as makes developers more forthcoming during interviews and surveys [99]. It is difficult to separate the different data collection methods that involve human interaction, therefore we think combining these data collection methods in the right order can greatly enhance the quality of the collected data. We were able to join the Rational Team Concert development team as a participant observer mainly due to the convincing argument that we take the place of an intern, thus, contributing to their development effort (Chapter IX)

4) *Interviews*: To further enhance our understanding on how developers view the situation and further make sense of survey responses as well as results from mining repositories and our observations, 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 [100] ask the interviewee to share memorable stories from work life. The interviewer then 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.** Our goal with this thesis is to better support software development. 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 [100]. This allows us to place more confidence into the reports and answers given by the interviewees. Whereas in structured interviews the interview itself is guided by a set of prepared questions and goals that do not necessarily address events that the interviewee can easily recall.

The main drawback of using war stories over prepared interview questions in a structured interview framework lies with the loss of focus of the interviews. 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.

We tried to minimize the interruption to project members as much as possible therefore we limited the time we require for each interview as much as possible. Most interviews are aimed at taking 30 minutes with a 30 minute overflow in case a participant is willing to continue the interview. Furthermore, we gave the work of the interviewee priority over the interview and assured the interviewees that they could stop the interview anytime they felt that their work needs attention. This was especially valuable with a professional development team such as the IBM Rational Team Concert development team as we joined their development effort when they were nearing a major milestone (Chapter IX).

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 chapter to allow us to manipulate the social interactions among developers. Thus this chapter explores our first research question:

RQ 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 introduced into the source code. Thus, we are confident that there exists a connection between developer communication and build success.

In this chapter, we start with describing the methodology that is relevant to exploring our research question (Section V-A). Then, we present our analysis and results we obtained in Section V-B followed by a discussion of the results in Section VI-D. We conclude this chapter with offering an answer to our research question and leading into the subsequent Chapter VI (Section VI-E).

A. Methodology

To address our research question we analyze data from a large software development project IBM Rational Team Concert which we described in created detail in Chapter ??.

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, WARNING or OK. We analyze build results to examine the integration outcomes in relation to the communication necessary for the coordination of the build.

Conceptually, the WARNING and OK build results are treated similar by the Jazz team, as they require no further attention or reaction from the developers. In contrast, ERROR build results indicate serious problems such as compile errors or test failures and require further coordination, communication and development effort. We thus treated all WARNINGS as OKs to clearly separate between failed and successful builds in our conceptualization of coordination outcome.

2) *Communication network measures:* To characterize the communication structure represented by the constructed social networks for each build (as described in Chapter IV), we compute a number of social network measures. The measures that we include in our analysis are: Density, Centrality and Structural holes. 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.

a) *Density:* Density is calculated as the percentage of the existing connections to all possible connections in the network. A fully connected network has a density of 1, while a network without any connections has the density of 0. For example, the density in the directed network in Figure 5 is $12/42 = 0.28$.

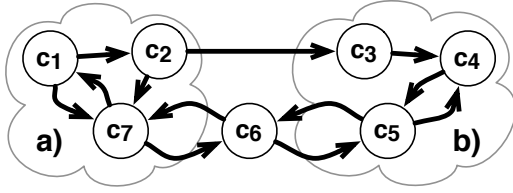


Fig. 5: Example of a directed network to illustrate our social analysis measures.

b) *Centrality measures:* We use the centrality measures *group degree centralization* and *group betweenness centralization* for complete networks, which are based on the ego network measures degree centrality and betweenness. The degree centrality measures for the ego networks are:

- The *Out-Degree* of a node c is the number of its outgoing connections $C_{oD}(c)$. E.g. $C_{oD}(c_1) = 2$ in Figure 5.
- The *In-Degree* of a node c is the number of its incoming connections $C_{iD}(c)$. For example $C_{iD}(c_1) = 1$ in Figure 5.
- The *InOut-Degree* of a node c is the sum of its In-Degree and Out-Degree $C_{ioD}(c)$. E.g. $C_{ioD}(c_1) = 3$ in Figure 5.

To compute the *Group Degree Centralization* index for the complete network we use formula (2) from Freeman [48], in which g is the number of nodes in a network, and $C_D(c_i)$ is any of the degree centrality measures of a node c_i as described above. $C_D(c^*)$ is the largest node degree index for the set of contributors in the network. The formula is also used by [37], [53].

$$C_D = \frac{\sum_{i=1}^g [C_D(c^*) - C_D(c_i)]}{(g-1)^2} \quad (2)$$

To calculate the *Group Betweenness Centralization* index for a whole network, we need to compute the betweenness centrality probability index for each actor of the network. The probability index assumes that a “communication” takes the shortest path from a contributor c_j to contributor c_k and if the network has more shortest paths, all of them have the same probability to be chosen. If g_{jk} is the number of shortest paths linking two contributors, $1/g_{jk}$ is the probability of using one of the shortest paths for communication. Let $g_{jk}(c_i)$ be the number of shortest paths linking two contributors that contain the contributors c_i . Freeman [48] estimates the probability that contributor c_i is between c_j and c_k by $g_{jk}(c_i)/g_{jk}$. The betweenness index for c_i is the sum of all probabilities over all pairs of actors excluding the i th contributor. Formula (3) shows the normalized betweenness index for directed networks.

$$C_B(c_i) = \frac{\sum_{j < k} g_{jk}(c_i)/g_{jk}}{(g-1)(g-2)} \quad (3)$$

To compute a betweenness index for the complete network instead of a single node, we used Freeman’s formula for *Group Betweenness Centralization*. The formula is shown in equation (4), in which $C_B(c^*)$ is the largest betweenness index of all actors in the network.

$$C_B = \frac{\sum_{i=1}^g [C_B(c^*) - C_B(c_i)]}{(g-1)} \quad (4)$$

c) *Structural holes:* We use the following structural hole measures:

- The *Effective Size* of a node c_i is the number of its neighbours minus the average degree of those in c_i ’s ego network, not counting their connections to c_i . The effective size of node c_1 in Figure 5a is $2 - 1 = 1$. Note, that only direct neighbours of c_1 are considered and the directed connections are replaced with undirected. The effective size of node c_4 in Figure 5b is $2 - 0 = 2$.
- The *Efficiency* normalizes the effective size of a node c_i by dividing the its effective size with the number of its neighbours. The efficiency of node c_1 in Figure 5a is $(2 - 1)/2 = 0.5$. The efficiency of node c_4 in Figure 5b is $(2 - 0)/2 = 1$.
- *Constraint* is a summary measure that relates the connections of a node c_i to the connections of c_i ’s neighbours. If c_i ’s neighbours and potential communication partners all have one another as potential communication partners, c_i is highly constrained. If c_i ’s neighbours do not have other alternatives in the neighborhood, they cannot constrain c_i ’s behavior.

To calculate network measures of the introduced ego network measures on structural holes, we compute the sum of the measures for each node of a network. As the measures are based on network connections, we normalize the sum by computing the fraction of the sum and the number of possible network connections.

3) *Data collection:* We mined the Jazz development repository for build and communication information. A query plug-in was implemented to extract all development and communication artifacts involved in each build from the Jazz server. These build-related artifacts included build results, teams, change sets, work items, contributors, and comments. We imported the resulting data into a relational database management system to handle the data more efficiently.

We extracted a total of 1288 build results, 13020 change sets, 25713 work items and 71019 comments. Out of a total of 47 Jazz teams, 24 had integration builds. The build results we extracted were created during the time range from November 5, 2007 to February 26, 2008.

Next, we had to make a decision which builds and associated communication to analyze. Our selection criteria was that we analyze a number of build results that is large enough for statistical tests and include both OK and ERROR builds. Some teams used the building process for testing purposes only and created just a view build results, while others had either only OK or only ERROR build results. Predicting build results for a team that only produced ERROR builds in the past, will most likely yield an ERROR, since no communication information representing successful builds is available. Thus, we considered teams that had more than 30 build results and at least 10 failed and 10 successful builds. Five teams satisfied these constraints and were considered in our analysis. In addition, we included the nightly, weekly, and one beta

	B	Team Level Builds				Project Level Builds		
		C	F	P	W	nightly	weekly	beta
# Builds	60	48	55	59	55	15	15	16
# ERRORS	20	16	24	29	31	9	11	13
# OKs	40	32	31	30	24	6	4	3
# Contributors:								
Min	3	9	6	5	13	43	37	55
Median	6	16.5	18	15	20	55	57	69.5
Mean	12.68	18.02	20.15	17.98	22.87	57.93	52.27	67.81
Max	58	31	64	61	52	75	75	79
# Directed Connections:								
Min	0	1	2	0	11	81	56	144
Median	13	39.5	95	36	74	236	149	280
Mean	51.58	53.4	87.78	63	88.35	253.1	171.9	285.8
Max	361	139	355	401	300	434	496	446
# Change Sets:								
Min	1	15	8	32	83	80	62	82
Median	10	38	35	46	111	117	115	178.5
Mean	10.83	44.38	42.65	47.25	115.3	129	114.2	166.8
Max	33	101	91	75	156	199	173	196
# Work Items:								
Min	0	2	1	1	10	11	5	31
Median	6.5	12	20	12	18	67	51	98
Mean	16.43	15.56	23.07	19.34	29.49	72.13	56.87	96.81
Max	131	50	100	107	119	132	202	170

TABLE I: Descriptive build statistics.

integration build, although they did not satisfy our constraints, because they integrate all subsystems of the entire project.

B. Analysis and Results

Table I shows descriptive statistics of the considered builds and related communication networks of the five teams (B, C, F, P and W in the first 5 columns) and the nightly, weekly, and beta project-level integrations. For example, team B created 60 builds from which 20 turned out to be ERRORS and 40 OK. The communication networks of this team had between 3 and 58 contributors (51.58 directed connections in average) and spanned 0 to 131 work items. The builds involved in average 10.83 change sets.

1) Individual communication measures and build results:

To examine whether any individual measure of communication structure can predict integration failure or success, we analyze the builds from each team and project-level integration in part in relation to the communication structure measures as follows: For each team we categorize the builds into two groups. One group contains the ERROR builds and the other the OK builds. For each build and associated communication network we compute the network measures described in Section V-A and compare them across the two groups of builds (ERROR and OK).

The communication measures used in the analysis were: Density, Centrality (in-degree, out-degree, inout-degree, and betweenness), Structural Holes (efficiency, effective size, and constraint), and number of directed connections. We used the Mann-Whitney test [101] to test if any of the measures differentiate between the groups of ERROR and OK related communication networks. We used the α -level of .05 and applied the Bonferroni correction to mitigate the threat of

		prediction	
		OK	ERROR
actual	OK	26	5
	ERROR	9	15

TABLE II: Classification results for team F.

multiple hypothesis testing. None of the tests yielded statistical significance, which indicates that no individual communication structure measures significantly differentiate between ERROR and OK builds.

We also tested for the possible effect of the technical measures shown in Table I: #Contributors, #Change Sets and the #Work Items on the build result. Also, none of the tests yielded statistical significance to differentiate between ERROR and OK builds.

2) *Predictive power of measures of communication structures:* 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 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 [102] and using the leave one out cross validation method [102].

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.

Table II shows the classification result for team F. The upper left cell represents the number of correctly classified communication networks as related to OK builds (26 vs. 31 actual), and the lower right cell shows the number of correctly classified networks as leading to ERROR builds (15 vs. 24 actual). The other two cells show the number of wrongly classified communication networks.

The classification quality is assessed via recall and precision coefficients, which can be calculated for ERROR and OK build predictions. We explain the coefficients for prediction of ERROR builds.

- **Recall** is the percentage of correctly classified networks as leading to ERROR divided by the number of ERROR related networks. In Table II the lower right cell shows the number of correct classified networks that are leading to ERRORS, which is divided by the sum of the values in the lower row, which represents the total number of actual ERRORS. This yields for Table II a recall of $15/(9+15) = .62$. In other words, 62% of the actual to ERROR leading networks are correctly classified.
- **Precision** is the percentage of as to ERROR leading classified networks that turned out to be actually ERRORS. In Table II, it is the number of correctly classified ERRORS

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

divided by the sum of the right column, which represents the number of as ERROR classified builds. In Table II the precision is $15/(5 + 15) = .75$. In practical terms, 75% of the ERROR predictions are actual ERRORS.

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 III 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.

C. Discussion

In our analysis we examined the relationship between integration builds and measures of the related communication structure. We found that none of the single communication structure measures (density, centrality or structure hole measures) significantly differentiated between failed and successful builds at the team-level and project-level. Therefore none of these individual communication structure measures could be used to predict integration build results.

In addition to the communication related measures, we also examined whether the technical measures we computed when constructing the communication networks – the number of change sets, contributors, and work items – have an impact on the integration build result, as they are an indication for the size and complexity of the development tasks to be coordinated. According to Nagappan and Ball [12], one might expect that increased size and complexity of code changes relates to more build failures. But in our study these single measures did not significantly differentiate between successful and failed build results. However, additional technical measures that were used by Nagappan might be good predictors in Jazz as well.

The second contribution of this work is the predictive model that uses measures of communication structures to predict build results. Interestingly, the combination of communication structure measures was a good predictor of failure even when the single measurements were not. Our model's precision in predicting failed builds, which relates to the confidence one can have in the predicted result, ranges from 50% to 76% for

any of the five team-level integration builds, and is above 73% for the project-level integration builds.

We found that, for all prediction models, the recall and precision values are better than guessing. A guess is deciding on the probability of an ERROR or an OK build if the build fails or succeeds. The probability is the number of ERRORS or OKs divided by the number of all builds. For example, if we know that the ERROR probability is 50% and we guess the result of the next build we would achieve a recall and precision of 50%. In our case, our model reached an ERROR recall of 62% for team F, where as a guess would have yield only $24/55 = .44 = 44\%$ (see Table I).

D. Conclusions

We conclude this chapter by bringing it back to the initial research question we set out to answer:

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

The result we presented in Section V-B show that our predictions, though not highly accurate, outperform random guesses. Therefore, we conclude that with recall of 55% to 75% and precision of 50% to 76%, depending on the development team, that communication indeed influences build success.

This finding opens the research avenue of investigating whether the manipulation of communication among software developers can yield positive results with respect to build success. This leads us to the next research question that we want to search for places within the social networks that we should manipulate to stimulate build success. For that purpose we turn in the next chapter to the concept of socio-technical congruence that might help us highlighting those developers that should have communicated.

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

Box 1 Examples of actual coordination

- Communication—A communicates with B [22], [23], [25], [103].
- Location—A is in the same location as B [22], [25].
- Team structure—A is in the same team as B [22].

towards improvements in the social network by pointing out developers that should communicate.

In this chapter, we start with discussing how we calculate socio-technical congruence (Section VI-A). Subsequently, we briefly discuss the methodology that is relevant to exploring our research question (Section VI-B). Then, we present the analysis and results we obtained in Section VI-C followed by a discussion of the results in Section VI-D. We conclude this chapter with offering an answer to our research question and leading into the subsequent chapter (Section VI-E).

A. Calculating Congruence

In Chapter IV we described socio-technical networks and how we conceptualize them in this thesis. 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 as follows:

$$\text{congruence} = \frac{\text{Diff}(CN, AC)}{|CN|}$$

The main difference to the original formula lies solely in our more direct approach of deriving the coordination needs matrix instead of deriving them from task relationships, that are themselves derived from source code dependencies as we used to directly relate software developers with each other.

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-technical congruence index as well as the ratio between gaps and coordination needs, that is technical dependencies among developers that are not accompanied by a corresponding social dependency.

We show the relationship between a variable and the build success probability by plotting the y-axis as the probability. We use probability because we feel that it is more intuitive than odds ratios or logistic functions. If there is a relationship between a variable and the probability of build success, then we should see that as the variable’s value increases, the

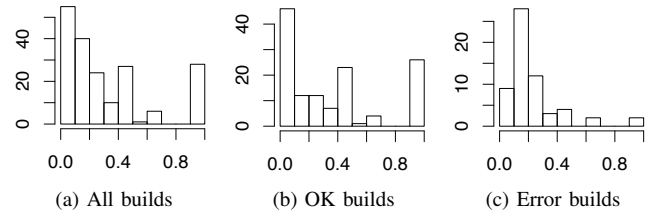


Fig. 6: Distribution of Congruence Values

	Min	Median	Max	Mean
Authors	2	17	44	18.62
Files	5	131	3101	342.3
change-sets	4	34	226	54.2
Work items	4	34	182	48.3
Build date range (days)	0	345	361	319.2
Congruence	0	0.21	1	0.331

TABLE IV: Summary statistics

probability also increases. In the probability figures, the solid line is the expected value, and the dashed lines indicate the 95% confidence intervals.

We run a logistic regression models for congruence. We include the following variables: number of files per build, number of authors contributing to the build, number of files in the build, number of work items per build, the congruence, the build type, and the date of the build. We centre and scale each numeric variable. Because we were concerned about possible interactions affecting our results, we included first-order interaction effects and used backward stepwise elimination to remove variables to keep AIC (Akaike’s Information Criterion) low.

C. Results

In the RTC repository, we analyzed 191 builds; of these builds, 60 were error builds, and 131 were OK builds. Table IV displays summary statistics per build. Figure 6 displays histograms for congruence. The histograms compare the frequencies for each type of congruence for all builds, the OK builds, and the error builds only.

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.

We calculated pairwise correlations between the variables congruence, number of authors, number of files, number of change-sets, number of work items, and build date using a Spearman Rank Correlation Coefficient [104] (Table V). To avoid multicollinearity problems in our data, we choose to remove change-sets from our logistic regression analysis because, due to the enforced processes in RTC, we know that there is exactly one author per change-set, and thus there is at least as many change-sets as authors per build.

To assess the fit of the logistic regression models, we use the Nagelkerke pseudo- R^2 and AIC. R^2 shows the proportion of variability explained by the model, and AIC is a measure of how well the model fits the data. Ideally, R^2 is high and AIC is low. Our current model contains 19 variables and has

	2.	3.	4.	5.	6.
1. Congruence	-0.27	-0.22	-0.33	0.08	0.19
2. Authors	–	0.41	0.76	0.10	-0.30
3. Files		–	0.37	0.08	-0.20
4. change-sets			–	0.02	-0.38
5. Work items				–	0.04
6. Build date					–

TABLE V: Pairwise Correlation of Variables per Build

Model	Variables	AIC	R^2
Every interaction	27	188.6	0.595
Main effects only	7	213.2	0.269
Our model	19	175.8	0.581

TABLE VI: Model comparison

an R^2 of 0.581. We present our model in Table VI to a model containing every first-order interaction effect with 27 variables and a model that contains the 7 main effects only (in Table VIII). We found that 19 variables is optimal and that removing further variables lowered the R^2 value while raising the AIC.

1) *Effects of Congruence on Build Result:* The result of logistic regression indicates that the following effects are significant: The congruence \times build type effect, the congruence \times build date interaction effect, the number of work items \times build date interaction effect, and the build date \times build type effect. In addition, the number of authors and the number of files are significant main effects, although their coefficients are lower than the interaction effects involving congruence.

In the next section we discuss the main effects and interactions effects that involve congruence affecting build probability. We discuss the effects of the non-congruence effects, including the authors, files, work items \times date interaction effect and the date \times nightly build effect in Section VI-C3.

a) *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 VII). We plot in Figure 7 the effects of congruence vs. probability of build success at the 10% date quantile (2008-01-25), at the 25% date quantile (2008-05-14), the 50% date quantile (2008-06-07), and the latest build (2008-06-26).

The congruence model (Table VII) the effect of congruence on continuous builds is significant, and that increasing congruence also increases the probability that a continuous build will succeed. For integration builds (Figures 7, in black), an increase in congruence decreases build success, with the exception of the 2008-01-25 build (Figure 7a). In in our 2008-01-25 build, we see that low congruence leads to low build probability, but high congruence has high build probability. As the project ages, this trend reverses and congruence is clearly inversely related with build success probability (Figure 7d). The effect of congruence is totally opposite for continuous builds and integration builds. Based on Figure 7d, increasing congruence significantly improves the continuous build success rate. However, increasing congruence significantly

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	$R^2 = 0.581$	
191 observations			
Build type is set to continuous			
Nagelkerke is used as the pseudo- R^2 measure			

$*p < 0.05$; $**p < 0.01$

* $p < 0.05$; ** $p < 0.01$

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

Variable	Coef.	S.E.	<i>p</i>
Intercept	0.5265	0.3040	0.0833
Congruence	0.9371	0.6807	0.1686
Authors	-0.5702	0.2003	**0.0044
Files	-0.6398	0.2477	**0.0098
Work items	-0.1755	0.1713	0.3055
Build type=I	0.1693	0.4269	0.6917
Build type=N	0.2133	0.7791	0.7842
Build date	-0.1331	0.1821	0.4649
Model likelihood ratio	40.59	$R^2 = 0.269$	
191 observations			
Build type is set to continuous			
Nagelkerke is used as the pseudo- R^2 measure			

p* < 0.05; *p* < 0.01

* $p < 0.05$; ** $p < 0.01$

TABLE VIII: Logistic Regression models predicting build success probability with main effects only

decreases the integration build success rate.

2) *Effect of Gap Ratio on Build Result:* We build a logistic regression model based on the model in Table VII using the gap ratio measurement (percentage of unmet coordination needs). In the interest of saving space, we report only the odds ratio. We retain every significant interaction from our previous congruence logistic regression in Table VII.

The effect of gap ratio on build result is significant (Table IX). This indicates that increasing the gaps ratio significantly increases the odds that an OK build will occur, which is the opposite of what we hypothesized (Figure 9). This means that if the gap is large, the build success probability increases.

3) *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 found in Section VI-C1, but different social and

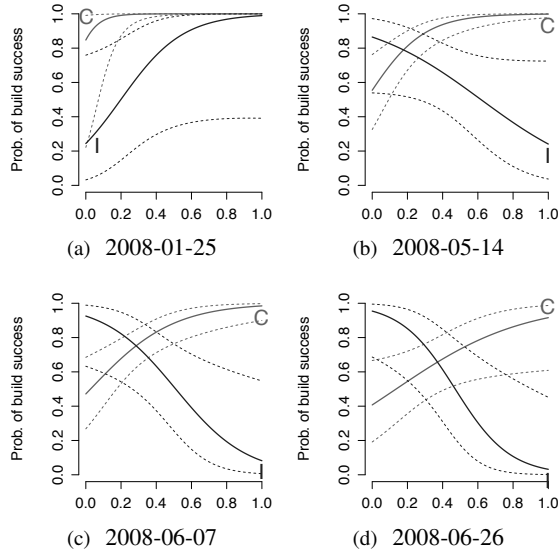


Fig. 7: Estimated probability of build success for *congruence* and *continuous builds C* or *integration builds I* over time, adjusted to authors ≈ -0.156 (17 authors), files ≈ -0.352 (131 files), work items ≈ -0.399 (34 work items)

	Model
Intercept	1.32
Authors	0.60
Files	0.63
Work items	0.85
Build type=I	1.31
Gap ratio	8.71
Build date	0.59
Authors * Build date	0.74
Work items * Build date	1.83
Build type=I * Build date	2.52

TABLE IX: Odds Ratio for Gap Ratio Models

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.

a) Other Effects on Build Success:

- **Authors** As the number of authors involved in a build increases, the probability that the build succeeds decreases. The build probability is significantly lowered after more than 15 authors are involved in the build (Figure 10a). When over 30 authors are involved in the build, the estimated build success probability falls under 10%.
- **Files** As the number of files involved in a build increases, the probability that the build will succeed decreases (Figure 10b).
- **Build Date and Work items** The work items \times date interaction is significant. Early in the project, as the number of work items increases, the probability of build success decreases (Figure 11a). As the project ages, this

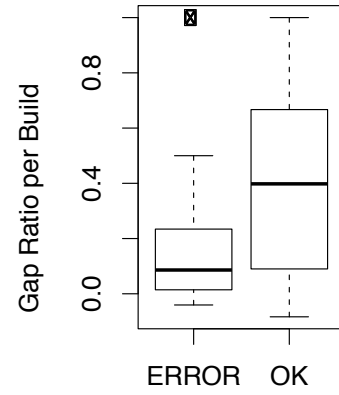


Fig. 8: Gap Ratio per Build

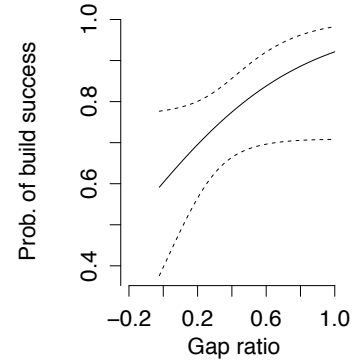


Fig. 9: Effect of gap ratio on build success probability.

trend reverses and as the number of work items increases, the probability of build success increases as well (Figure 11b). According to the coefficients in Table VII, this effect on build success probability is not as strong as the authors main effect or the files main effect.

b) Examining Extreme Congruence Values: We are interested in the differences between high-congruence builds and low-congruence builds. We further this investigation by looking at builds that have extreme values of congruence: zero, where absolutely no coordination needs are satisfied with communication, and one, where every coordination need is satisfied with communication. We chose to investigate the extreme cases to see if there were differences in the way people coordinated in fully-congruent builds, and in incongruent builds. Table X shows the number of OK and error builds that occurred when congruence was equal to one, and equal to zero.

To determine if the presence of commenting affected the builds, we examined the number of comments on work item-change-set pairs in builds with extreme congruence values. Our results are shown in Table XI. Build success probabilities improve with respect to builds that have no comments, though work items with no comments are in the minority.

Of note is the high number of comments on work items that have zero congruence. This indicates that individuals who have no technical relationship to the work item are commenting on the work item.

We manually inspected the work items with extreme

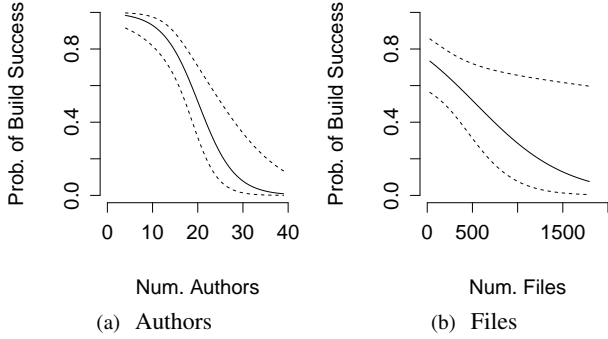


Fig. 10: Estimated probability of build success for *authors* and *files*, congruence. Adjusted to work items ≈ -0.399 (34), authors ≈ -0.156 (17), files ≈ -0.352 (131), congruence ≈ 0.1446 , type = cont, date=2008-06-26

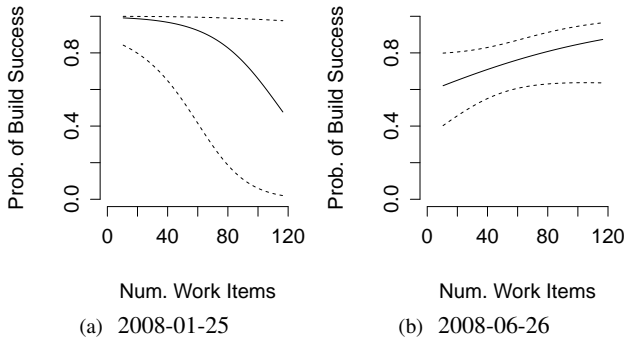


Fig. 11: Estimated probability of build success for *work items* and *date*, congruence. Adjusted to authors ≈ -0.156 (17), files ≈ -0.352 (131), congruence ≈ 0.1446 , type = cont

amounts of congruence, reading the comments for any differences in the content discussed. Unfortunately, there were no obvious qualities between comments made in a build with a congruence of zero, and comments made in a build with a congruence of one. In both builds, individuals discussed technical implementation details, provided updates to colleagues, or requested assistance from colleagues. We are unable to discover root causes of failure without a deeper examination of the technical changes and more knowledge of the RTC context.

D. Discussion

The concepts illustrated in Conway's Law, as well as previous empirical work on socio-technical congruence lead us to expect that team members must coordinate according to coordination needs suggested by technical dependencies in order to build software effectively. In this case study, we applied socio-technical congruence to study coordination and its relationship to build success probability in RTC.

Overall, we found that the average congruence across builds was very low—only 20–30% of the coordination needs in the project were fulfilled with actual coordination. Even in the cases where there is zero congruence, the build result was an OK build in over 90% of the observed cases.

Congruence		
	1	0
OK	26	30
ERR	2	2

TABLE X: Number of Builds with Congruence Values 0 and 1

		Num. of Pairs		Success rate	
Congruence		1	0	1	0
No comments	OK	42	143	49%	69%
	Error	43	64		
Comments	OK	610	445	68%	69%
	Error	290	199		
Total	OK	652	588	66%	69%
	Error	333	263		

TABLE XI: Number of work items-change-set pairs with comments and build success probabilities for congruence 0 and 1

We found that there was an interaction effect involving congruence and build type on build success probabilities (Section VI-C1a). For continuous builds, increasing congruence improves the chance of build success in continuous builds and can actually decrease build success probability in integration builds (Figures 7). Congruence significantly reduce integration build success probability.

The gap ratio is a representation of whether enough coordination occurred to fulfill multiple coordination needs. If two developers have multiple dependencies on each other, one would expect them to coordinate more often as well. We hypothesized that a small gap ratio would increase the probability of successful builds and that a large gap ratio would decrease the probability of a successful build. Instead, we found that as the mean gap increases, the build success probability also increases (Figure 9).

Below we discuss the reasons for these observed results based on our knowledge of RTC.

1) *Strong Awareness Helps Coordination*: The overall congruence for the majority of builds is low: over 75% of builds have a congruence of less than 0.25 (Section VI-C1). Despite low congruence, the RTC team is able to successfully build its software in many situations.

If socio-technical congruence is a measure of coordination quality in software, and builds rely on coordination quality to be successful, then there must be reasons why builds can succeed even when the congruence is zero. Because RTC is a highly-distributed project, the product under development uses a modular design [105] and thus is affected less by dependencies. Second, team members in RTC do not conduct all of their coordination through *explicit communication* even though work item inspection and discussion with developers indicate that the RTC corporate culture focuses on the work item as their base for communication. Rather, they use the *shared workspace* that incorporates cues from the environment and from peers in order to address technical issues. Both of these effects may contribute to congruence being lower than expected.

a) *RTC supports Explicit Communication:* The RTC team members use the RTC environment extensively to communicate with each other. We were informed that RTC team members rarely use private email, and our inspection of the mailing list reveals that its primary purpose is for announcements such as server outages rather than for discussing technical work.

This leaves the RTC work item comment system and instant messaging as methods for communication, as well as the phone and internal face-to-face meetings. We learned that while face-to-face interaction is efficient for solving local issues, it does not benefit remote teams, and the RTC team as a whole encourages every team member to record face-to-face discussions as comments for the purpose of archiving and sharing information.

However, explicit coordination has a cost. There is evidence that involving too many authors in the same build also reduces the build success (Figure 10a). The overhead required to coordinate many people may interfere with the ability of the team to build the project successfully, suggesting that there is a limit before a developer is overloaded with information.

b) *RTC is a Shared Workspace:* The RTC client software helps a developer acquire and maintain *environmental awareness* of what is going on in the project by providing access to a shared workspace. Much of the work is centred around the RTC technical entities, which include plans, source code, work items, and comments. RTC's awareness mechanisms feature a developer-centred dashboard that reports changes to the workspace, built-in traceability, user notifications, regularly-generated reports, and an optional web browser interface. For example, when a change-set is created, it is attached to a work item, thus ensuring that people who are involved with the work item receive notification of this change-set. These automatic notifications cut down the amount of explicit communication and allow people to coordinate implicitly.

Coordinating using the workspace is well-known in the computer-supported cooperative work domain [106]. Open-source developers, in particular, coordinate around source code [29] and mailing lists [107], [108] because there is little opportunity for face-to-face interaction. RTC shares many characteristics with open-source development, such as a distributed team and a transparent development process.

In light of these results, we believe that, using our conceptualization, the RTC team requires a congruence of only 0.2–0.3 for their tasks to be completed. Much of the need for explicit, point-to-point communication is mitigated by implicit communication and the use of the workspace to coordinate. We expect that the remaining congruence is covered through the RTC workspace, and through face-to-face communication, instant messenger, and phone communication. Though our congruence value appears low for the RTC team, the coordination in reality may be higher. Future studies should keep in mind that congruence may be lower than expected because of conceptualizations that cannot include every type of coordination in a project.

2) *Coordination and Geographic Distribution:* As RTC is a distributed team, geographic distribution has an effect on team performance, though the RTC environment helps mitigate

some of these effects [80].

We learned from the RTC project that continuous and nightly builds should involve mainly a co-located team, and that integration builds involve multiple components from RTC teams in different locations. Our results suggest that congruence best benefits builds that occur within co-located teams; however, the design of our study does not allow us to draw a firm conclusion about the influence of both co-location and congruence on build success probability.

It appears that involving too many individuals when coordinating the activities of various teams may harm build success due to information overload [36], especially when the team members are distributed. To negate this effect, development leaders and build managers that have an overall view of the project are suited to coordinate teams to ensure build success [37].

3) *Project Maturity and Build Success:* We found that early builds exhibited a different type of relationship between congruence and build success probability than later builds (Section VI-C1a). Over the course of the study, we observed 13 internal milestones; the last milestone in our observed builds was a public beta release for end users.

Build success probability decreased significantly over time for continuous builds and stayed roughly the same for integration builds (Figures 7). However, the early builds in the project behaved contrary to later builds in the project (Figure 7a and 7a). The RTC software early in its lifetime is in a state of change. Integration builds are not a priority, and features are being added to the project. This means that dependencies are changing rapidly, as well as the expertise among team members, making it difficult to solidify coordination needs.

In addition to interactions between congruence and type, and congruence and date, we observed a significant interaction effect between build date and work items. We found that early in the project (Figure 11a), builds with large numbers of work items have a high probability of failing, but late in the project (Figure 11b), these builds succeed. Because the latest release was focused on a public release, a build linked to numerous work items may indicate that a bug is highly problematic or a feature is highly desired, and therefore received more attention.

E. Conclusions

We end this chapter by bringing it back to the initial research question we set out to answer:

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

We conducted two investigations: (1) the investigation of the influence of the socio-technical congruence index on build success and (2) the investigation of gaps uncovered by socio-technical networks and their influence on build success. Both avenues of investigations showed that they expose an influence on build success.

These findings, especially that gaps within socio-technical networks influence build success opens the door to formulating an approach on how to leverage the concept of socio-technical congruence to improve the communication among software developers. Thus in the next chapter we will detail an approach

that we propose to improve developer communication using the concept of socio-technical congruence.

VII. THE APPROACH

In this chapter we propose an approach to leverage the concept of socio-technical congruence to improve communication (social interactions) among developers to improve build success. We base this approach on the findings presented in the previous two chapters that team communication (Chapter V) and socio-technical gaps (Chapter VI) influence build success.

A. Approach to Leveraging Socio-Technical Congruence

The main contribution of this thesis lies with the approach to creating actionable knowledge from the socio-technical congruence concept as Cataldo et al [22] described it in their seminal paper. Thus, we derived the following approach from the first two research questions:

Our approach encompasses five steps:

- 1) Define scope of interest.
- 2) Define outcome metric.
- 3) Build social networks.
- 4) Build technical networks.
- 5) Generate actionable insights.

B. Define Scope

Each network, social, technical and thus socio-technical, needs to be built with a specific scope in mind. For example, throughout this thesis we focus on software builds. This focus defines the source and communication artifacts that are used to construct the social and technical networks. In Chapter IV Figure 2 we showed how we conceptualized social networks and how defining software builds let us select communication artifacts for the construction of a social network for a specific build.

C. Define Outcome

In order to derive useful insights from the constructed networks the scope used to construct them needs to be complimented with an outcome metric. For example, in this thesis we are interested in build success, a binary variable that states whether a build is of acceptable quality. With an outcome measure at hand we can contrast networks to gain insights. Note, that the outcome and scope are closely related as we need to attach the outcome to the social and technical network whose construction depend on the scope.

D. Construct Social Networks

After defining the scope and outcome, the next step is to construct the social networks. This involves identifying all communication channels that are programatically accessible in real time. Some examples of communication channels that can be tapped into in real time are emails, forum style discussions, or text chats. Gathering all to the selected scope relevant communication artifacts than yields a social network. In Chapter IV Figure 2 shows a detailed example on how we construct a social network given a build as the scope of interest using data from the Rational Team Concert development team.

E. Construct Technical Networks

To complement the social networks and thus create socio-technical networks we need to produce technical networks. The main issues is not to rely on information that is only available after the work has been completed and been submitted to a version repository, but to gather information to construct networks in real time. For instance, Blincoe et al [30] proposed to use Mylyn⁷ events to accomplish the extraction of interactions with source code in real time. With respect to software quality it suffices to analyze complete changes and give feedback once the implementation work has been submitted to a central code repository.

F. Generate Insights

In Chapter VI we showed that gaps in socio-technical networks influence build success. We take this as starting point for generating insights by breaking down socio-technical networks into triples that consist of a developer pair together with how they are connected, which can be either through a technical dependencies, a social dependency, both, or none.

Since we are focusing on improving the social interactions we focus on identifying those gaps (unmet coordination needs) that are most likely to increase build success. For this purpose we split the triples derived from all socio-technical networks into two groups, those triples that originate from a socio-technical network of a failed build and those that originate from a socio-technical network of a successful build.

For each triple we can now apply statistical tests to see if a given triple is more often observed with failed or successful builds. Those that are statistically significant for failed builds should be broken by suggesting to developers to communicate. To ensure not to worsen the odds of a successful build we contrast the developer pair forming the gap with the same pair being connected both technically and socially.

G. Conclusion

The approach we described builds on our work on software builds and the influence of communication and unmet coordination needs as shown in the previous two chapters. Note that this approach can also be applied to identify those met coordination needs that warrant further examination, by applying the same insight generation to pairs of developer that are connected both through a technical and social dependency.

The following three chapters aim at identifying the usefulness of the approach we detailed in this chapter. We start by first investigating whether we can using this approach generate recommendation that are statistically significant (Chapter VIII). Following, we diverge from the path of exploring the validity of the recommendations and focus on whether developers are open to such recommendations (Chapter IX). Finally, in a large student project in a course offered at the University of Victoria, Canada, and Aalto University, Finland, we explore whether we could generate recommendations that might have prevented builds from failing (Chapter ??).

⁷<http://tasktop.com/mylyn/>

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

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

VIII. SOCIO-TECHNICAL CONGRUENCE AND FAILURE

The first step we take towards exploring the usefulness of our approach is to test whether we can generate recommendations that break patterns related to build failure. Thus we focus on the following research question:

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

Intuitively, the idea is that two developers that share a technical dependency should talk, but in some cases when the technical relationship is trivial instead of preventing failures such a recommendation might only create additional communication overhead. Thus we explore to what extent the history of a project can be used to highlight those cases where developers that shared an unmet coordination need can be brought more often into relation with failed builds than with successful builds.

In this chapter, we start by briefly presenting the methodology that is relevant to exploring our research question (Section VIII-A and VIII-B). Then, we present the analysis and results we obtained in Section VIII-C followed by a discussion of the results Sections VIII-D. We conclude this chapter with offering an answer to our research question and leading into the subsequent chapter (Section VIII-E).

A. Socio-technical coordination

We build our socio-technical networks according to the approach outlined in Chapter VII:

- 1) Define scope of interest.
- 2) Define outcome metric.
- 3) Build social networks.
- 4) Build technical networks.

As our scope we select the software build and as outcome metric whether a build failed or succeeded. We construct the social and technical networks from the Rational Team Concert development team repository as outline in Chapter III.

B. 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 [24]. 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*.

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

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

We analyze the technical pairs in relation to build failure. Our analysis proceeds in four steps:

- 1) Identify all technical pairs from the socio-technical networks.
- 2) For each technical pair count occurrences in socio-technical networks of builds that failed.
- 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 XII we illustrate the analysis of the technical pair (Adam, Bart). This pair appears in 3 successful builds and in 13 failed builds. Thus it does not appear in 224 successful builds, which is the total number of successful builds minus the number of successful builds the pair appeared in, and it is absent in 86 failed builds. A Fischer Exact Value

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 XV: Logistic regression only showing the technical pairs from Table XIII, the intercept, and the confounding variables, the model reaches an AIC of 706 with all shown features being significant at $\alpha = 0.001$ level (indicated by ***).

test yields significance at a confidence level of $\alpha = .05$ with a p-value of $4.273 \cdot 10^{-5}$.

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

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

$$p_x = \frac{\text{pair}_{\text{failed}} / \text{total}_{\text{failed}}}{\text{pair}_{\text{failed}} / \text{total}_{\text{failed}} + \text{pair}_{\text{success}} / \text{total}_{\text{success}}} \quad (5)$$

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

C. Results

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

We rank the failure relating *technical* pairs (see Tables XIII) by the coefficient p_x . This coefficient indicates the strength of relationship between the developer pair and build failure. For instance, the developer pair (Adam, Bart), appears in 13 failed builds and in 3 successful builds. This means that $\text{pair}_{\text{failed}} = 13$ and $\text{pair}_{\text{success}} = 3$ with $\text{total}_{\text{failed}} = 99$ and $\text{total}_{\text{success}} = 227$ result in $p_x = 0.9016$. Besides that we report the number of successful builds the pair was observed with ($\# \text{successful}$) as well as the number of failed builds the pairs was observed with ($\# \text{failed}$). 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 XIV we show the socio-technical pairs that match the 20 technical pairs shown in Table XIII as well as the same information as in Table XIII. 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 12 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.

D. Discussion

These results show that there is a strong relationship between certain technical developer pairs and increased likelihood of a build failure. Out of the total of 120 technical pairs that increase the likelihood of a build to fail, only 23 had an existing corresponding socio-technical pair. Of these, none were statistically related to build failure. This means that 97 pairs of developers that had a technical dependency did not communicate with each other and consequently increased the likelihood of a build failure. Our results not only corroborate past findings [22], [23] that socio-technical gaps have a negative effect in software development. More importantly, they indicate that the analysis presented in this paper is able to identify the specific socio-technical gaps, namely the actual developer pairs where the gaps occur and that increase the likelihood of build failure.

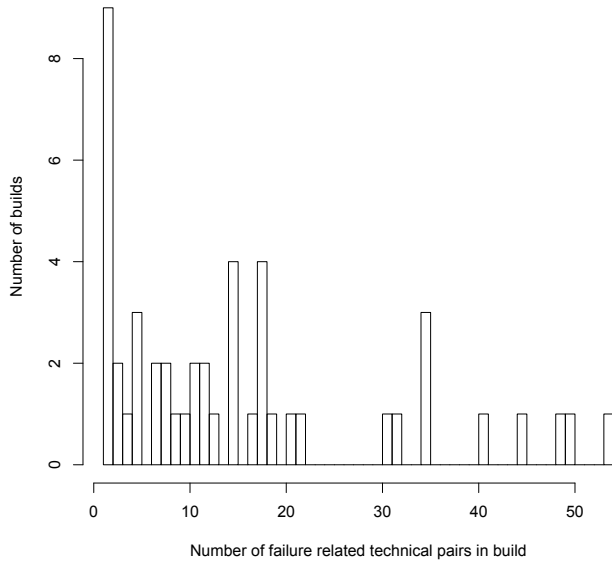


Fig. 12: Histogram of how many builds have a certain number of failure-related technical pairs.

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 [19] 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 Rational Team Concert team strongly emphasizes communication regardless of team boundaries, it still seems that organizational distance has an influence on its communication behaviour.

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

Overall the logistic regression confirms the effect of the developer pairs even in the presence of confounding variables, such as number of files changed and numbers of developers (AIC of 706). Table XV 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 XV as well as the coefficients for the four control features.

All features shown in Table XV are significant at the $\alpha = .001$ level (indicated by the p-value and ***). We checked for all the other technical pairs that were reported significant using the approach described in the previous section and found that they are all significant in the regression model as well.

Moreover, the four control variables of number of developers per build, number of work items per build, number of change sets per build, and number of files changed per build are also significant.

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

E. Conclusions

We end this chapter by bringing it back to the initial research question we set out to answer:

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

The results we presented in Section VIII-C show that there are developer pairs with unmet coordination needs that negatively influence build success. Furthermore, we showed that the corresponding developer pair that fulfills its coordination need is less likely to be associated with build failure and thus theoretically decreasing the likelihood of build failure. To put it in perspective if any of the twenty most harmful patterns is present in a build the build had at least an 74% chance to fail.

Besides the technical side of the approach to generate statistically relevant recommendations as Murphy and Murphy-Hill [31] pointed out developers need to accept such recommendation for them to be ultimately useful.

IX. ACCEPTABILITY OF RECOMMENDATIONS

Knowing that our approach can generate statistically significant recommendations is only one necessary step towards showing its usefulness. As Murphy and Murphy-Hill [31] pointed out recommendation become of little use once developers reject them. Thus in this chapter we pursue the following research question:

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

We want developers to communicate as soon as we suspect that otherwise the likelihood for a build to succeed diminishes. But currently our investigation only took a very technical stance and ignores many other criteria that might be of relevance to developers. For instance, it is unlikely that every build is of equal importance. Furthermore, we also think that developers opinion of each other play a role, such as trust and respect. Therefore, it is important to explore those less technical aspects of developer relationship to gain a better understanding on when to deliver a recommendation.

In this chapter, we start by detailing the study design that is relevant to exploring our research question (Section IX-A). Then, we present the findings we obtained in Section IX-B. We conclude this chapter with offering an answer to our research question and leading into the subsequent Chapter ?? (Section IX-C).

A. Study Design

We complement our previous analysis of the Rational Team Concert (RTC) development team repositories with rich qualitative data from interviews and observations, considering that much information on a project's lifecycle is not recorded in its repositories [109]. We focused on studying what factors cause a developer to seek information about a change-set, since change-sets are the smallest units of change to a project that directly impact other developers.

Our research question calls for a mixed methods empirical study. Therefore, we collect data about the team's information-seeking behaviour from three different sources. First, the author of this thesis was embedded in a sub-team of the RTC team as a participant-observer for four months. Second, we deployed a survey with the entire development team to validate some of the findings from the observations. Third, we conducted interviews with the developers from one component team to obtain richer information about their communication behaviour.

1) *Data Collection:* We used a mixed methods approach to answer our research question. We obtained data from three sources: participant observation, a survey, and a set of interviews with RTC team members.

a) *Participant-Observation:* The author of this thesis joined one of the RTC development sites in Europe for a four-month period in the Fall of 2010. He was involved as an intern with this development team helping with minor bug fixes, feature development, and testing, and thus was in a position to directly participate in the project development and communication activities. Our data collection opportunities were thus much richer than in typical observations conducted over shorter periods of time, or that do not involve active participation in the project. During his period as a participant-observer, Adrian kept a daily activity log, recording any information of relevance to the communication behaviour of his team.

The observation period coincided with the months prior to a major release and during which the team focused on extensive testing rather than new feature development. As such, the majority of our observations are concerned with activities around testing the integration of RTC with other IBM Rational products, and it offered the experience of collaborating with many developers from different teams and across the remote sites. Finally, in the one month following the release, Adrian also participated in the development of a feature for the upcoming version of the product.

The team in which he was embedded had the responsibility to develop the task management and the planning components of RTC. There were ten developers at the site, including two novices that had recently joined the team and eight experienced developers. The team includes RTC's three senior project management members, who play a major role in planning the entire product's architecture and development.

b) *Interviews:* We conducted interviews with ten developers of the RTC team to inform our observational findings. The setting for the interviews was slightly unusual, and beneficial from a research perspective, since the first author's

participant-observation allowed him to develop working relationships during his stay at the team, and to delve into a discussion of complex communication issues without needing to spend time understanding the basic context of the team.

In the interviews, we requested developers to provide details into their communication dynamics through a narration of "war stories" that directly related to communication among developers. "War stories" are events that left an impression on the respondent. War stories emphasize specific events that they witnessed or took part in [100]; their narrative is particularly powerful at uncovering the elements of the stories that are relevant to the respondents. The interviews lasted between thirty minutes and one hour, and they were conducted at the end of the observation period.

c) *Survey:* To complement the insights we obtained from the observations and interviews, we deployed a survey with the entire RTC team at the end of our observation period. The survey was designed iteratively and piloted with a European team, and intended to collect input about which factors increase the likelihood of a developer requesting information about a change-set from the change-set author. The items we included in the survey were in the categories of code-related, developer-related and process-related items. Each category included 14 items, as shown in Table XVI. *Code-change related* items describe the change-set delivered or the code that the change-set modifies or affects. *Developer-related* items relate to the developers that delivered a change-set, the developer requesting the information, and their relation to each other. Finally, *process-related* items relate to the development phase in effect when the change-set was delivered, and to items relating to process requirements or practices. For each of the three categories, the survey asked the developers to rank each item in the category according to how strongly it increases their likelihood of requesting information about a change-set.

We initially formulated our list of items from an analysis of the current literature on coordination and communication, as well as from our own previous research. We then refined the list of items through piloting and discussions with the development team. An initial version of the survey included fifty-nine items; the refined list had forty-two. Besides reducing the number of items in our list, the pilot survey led us to add several process-related items, as well as to change the format of the question (the initial survey had a battery of questions with Likert-scale answers; the final survey asked respondents to rank each item in comparison to the rest in its own category).

We deployed the survey through a web form, and invited the entire distributed RTC development team to participate. We sent a reminder one month later to increase the response rate. We obtained 36 responses to the survey; approximately a 25% response rate. Of the 36 respondents, 26 are located in North America, 5 are located in Europe, and 5 are located in Asia. Furthermore, 22 of the respondents are developers, 5 team leads, 5 component leads and project management committee, and 4 respondents withheld their information.

2) *Analysis:* For our data analysis, we transcribed the interviews, and then coded both the interviews and the participant-observation notes. From the codes we created categories,

Survey Items	Interview Quotes (I)/Participant Observation Quotes (O)	Mean Rank
Release endgame	(O) <i>"Adrian, in the endgame we only do minimal changes. Is your change minimal?"</i>	3.79
To review the change	(I) <i>"I often lack sufficient understanding of the part of the code I'm reviewing."</i>	5.00
To approve the change	—	5.11
Late milestone	—	5.14
During iteration endgame	(O) <i>"Adrian, in the endgame we only do minimal changes. Is your change minimal?"</i>	5.42
Obtain a review for the change	(O) <i>"It is demotivating to get a reject, thus I talk to my reviewer beforehand."</i>	5.55
Obtain an approval for the change	(O) <i>"I already fixed it, but I need to convince my team lead to give me approval."</i>	5.72
Related work item has high severity	—	6.00
Verify a fix	(I) <i>"Often I need to ask how I can tell that a change-set actually fixed the bug."</i>	6.37
Topic of work item the change is attached to	—	7.64
Priority set by your team lead	—	8.55
Role of the committer (e.g. developer)	—	9.00
Early in an iteration	(I) <i>"... there are weeks I sometimes don't talk to my colleagues at all."</i>	11.24
Early milestone	—	12.10
(a) Process-related items and quotes		
Survey Items	Interview Quotes (I)/Participant Observation Quotes (O)	Mean Rank
Author is inexperienced	(O) <i>"Adrian, please put me always as a reviewer on you change-sets."</i>	2.60
Author recently delivered sub-standard work	(I) <i>"She just changed teams and still needs to get used to this component."</i>	3.04
Author is not up to date with recent events	(I) <i>"After you return from vacation we ensure you follow new decisions."</i>	3.15
You do not know the change-set author	(I) <i>"I'd be very irritated if someone other than [...] would touch my code."</i>	5.09
Author is currently working with you	(O) <i>"We sometimes discuss changes we made to brag about a cool hack."</i>	5.56
Author part of same feature team	—	6.00
Author part of your team	—	7.00
Worked with author before	—	7.77
Busyness of yourself	(I) <i>"If I see a problematic change-set, I'll ask for clarifications even if I'm busy."</i>	8.05
Busyness of the author	(I) <i>"If I need to know why an author made that change I just contact him." [Even if he is busy.]</i>	8.55
Met in person	(I) <i>"I've worked with him for 5 years now but never even met him."</i>	9.57
Physical location	(O) <i>"Here, America or Asia, I don't care, I ping them when they are online."</i>	10.14
Author is experienced	—	11.00
Author recently delivered high-quality work	—	11.09
(b) Developer-related items and quotes		
Survey Items	Interview Quotes (I)/Participant Observation Quotes (O)	Mean Rank
Changed API	(O) <i>"Adrian, why did you change that API?"</i> [The RTC team avoids changing API.]	2.84
Don't know why code was changed	(O) <i>"I don't see a reason why you changed that code, you sure you needed to?"</i>	3.00
Affects frequently used features	(I) <i>"Before I ok a fix [even to a main feature], I ask if there is a workaround."</i>	4.10
Complex code	—	4.89
Introduced new functionality	—	6.14
Is used by many other methods	(O) <i>"Why did you fix this part? The bug is not in this part of the code, it is used and tested a lot."</i>	6.41
Your code was changed	(I) <i>"I'd be very irritated if someone other than [...] would touch my code."</i>	6.42
Stable code was changed	(O) <i>"Why did you fix this part? The bug is not in this part of the code, it is used and tested a lot."</i>	6.62
Change unlocks previously unused code	—	7.42
A bug fix	—	8.61
A re-factoring	(O) <i>"I separate a re-factoring from a fix so that people can ask me questions about the fix."</i>	9.15
Frequently modified code was changed	(I) <i>"I like to know where the 'construction sites' of the project are."</i>	10.35
Code is used by few other methods	—	10.96
Simple code was changed	—	12.57
(c) Code-change-related items and quotes		

TABLE XVI: This table (in three parts) summarizes the data we collected from our three data sources: 1.) our participant-observer's log-book; 2.) survey responses from 36 developers situated around the world; 3.) interviews with 10 developers. These items are ranked by their average rank. Whenever possible, we provided quotes from interviews (I), or quotes from the participant-observation period (O) to exemplify the items.

which we cross-referenced to the survey rankings. We used an open coding approach, during which we assigned codes to summarize incidents in sentences and paragraphs of the transcribed interviews that relate to our research question.

We calculated the ranking of the survey items for each category by calculating the mean over all ranks given by developers to a survey item (see Table XVI). Table XVIII shows the distribution of the rankings for each survey item.

During our analysis, we derived a set of factors affecting communication behaviour based on our list of survey items and qualitative data from our interviews and observations.

Whenever possible, we triangulated our findings using all our data sources. We tried to ensure that all of our findings were supported by at least two data sources; none of the findings we report were challenged by any of our data sources.

B. Findings

The evidence collected from our survey, interviews, and participant observation allowed us to derive seven factors that affect communication behaviour in software organizations, which we outline below. In the remainder of this section, text in *italics* refers to survey items that can be found in Table XVI.











































Process-related items	
	Release endgame
	To review the change
	To approve the change
	Late milestone
	During iteration endgame
	Obtain a review for the change
	Obtain an approval for the change
	Related work item has high severity
	Verify a fix
	Topic of work item the change is attached to
	Priority set by your team lead
	Role of the committer (e.g. developer)
	Early in an iteration
	Early milestone
Developer-related items	
	Author is inexperienced
	Author recently delivered sub-standard work
	Author is not up to date with recent events
	You do not know the change-set author
	Author is currently working with you
	Author part of same feature team
	Author part of your team
	Worked with author before
	Busyness of yourself
	Busyness of the author
	Met in person
	Physical location
	Author is experienced
	Author recently delivered high-quality work
Code-change related items	
	Changed API
	Don't know why code was changed
	Affects frequently used features
	Complex code
	Introduced new functionality
	Is used by many other methods
	Your code was changed
	Stable code was changed
	Change unlocks previously unused code
	A bug fix
	A re-factoring
	Frequently modified code was changed
	Code is used by few other methods
	Simple code was changed

TABLE XVIII: This table contains the distribution of ranks for each survey item. The leftmost point of each sparkline represents the amount of respondents that ranked the item first; the rightmost point represents the amount that ranked it last (14th). No survey item received the same rank from more than ten respondents.

Text in *italics* and around quotation marks denotes interview fragments.

1) *Development Mode*: 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, as shown in Figure 13.

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

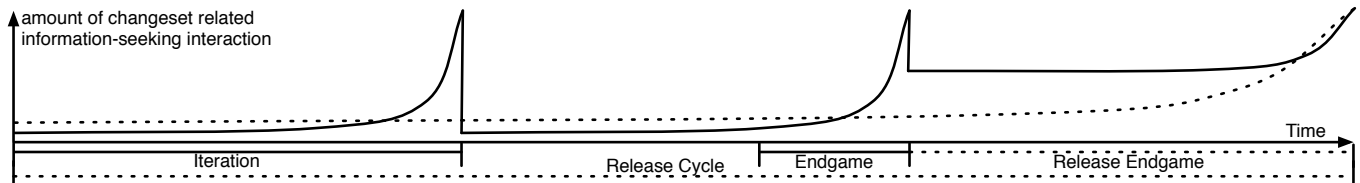


Fig. 13: The pattern of information-seeking interactions throughout several iterations of a release cycle. Every release cycle consists of a number of iterations; each iteration includes an endgame phase. change-set-based interactions are more frequent during endgame phases and during the last iteration of the release cycle.

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. In Table XVIa, we can see that overall, for the RTC development team at large, 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 the change-set Author*: One key determinant for a developer to seek information about a change-set consists of what the developer knows about the background of its author. Several aspects of the author’s background seem to be important: the quality of her recent work, her level of experience, and her awareness of recent team events or decisions.

First, these factors are important in part because they point to potential problems with the stability of the product. If a change-set author has *recently delivered sub-standard work*, other developers will want to ensure that the new change-set will not deteriorate their product, and therefore they will try to find more information about it.

Similarly, when the author is *inexperienced*, or is not *up to date* with recent team events for any reason, the risk of introducing problems into the product increases. One developer that interacts frequently with a newly founded team told us that “most of the work that I need to review is of very low quality and needs several iterations before it is up to our standards.” Novices, generally speaking, face a ramp-up problem that takes a significant amount of time to overcome [110]. This was the case with Adrian’s work, too: his first change-sets were subjected to more scrutiny due to his unfamiliarity with

the code base.

The survey data backs up these observations. The three top items in Table XVIb consist of factors related to an unfavourable perception of the background of the change-set author. In comparison, the two least important items in that table refer to favourable perceptions of said background.

In essence, these items point to the relevance of trust in software development teams. For developers, trust in the skills of their colleagues is important. If this trust is not-established (for instance, if a developer does not know the change-set author), developers are more likely to check the change-set and request information to ensure that the modifications were truly necessary and appropriate. But when trust exists (that is, when the author has a good record of delivering high-quality work, or is known to be an experienced team member), the need to seek information about a change-set decreases.

3) *Common Experience and Location*: Shared work experiences affect the likelihood of information-seeking behaviour in the RTC distributed team. This is partly related to the trust and perception issues discussed above, but also to the establishment of interpersonal relations and to the intertwining work responsibilities and expectations.

According to our interviews, this is the case particularly for senior team members. Although they work in a globally distributed team, they have managed to establish personal relationships with developers at many different sites. Through continuous interaction and through social events, they have learned more about each other, and thus feel more comfortable to initiate contact with them. Junior team members, in contrast, know very few of their teammates globally, and since they simply do not have these interpersonal connections with the rest of their team, it is less important to them whether they know the author of a change-set when needing to contact them.

Generally speaking, despite the lack of opportunities to build personal relationships with off-site people, for most developers sharing common experiences makes it easier to contact teammates to request information. One team lead told us: “I have one or two contact people in each team we usually work with whom I ask for information and then often enough get referred to the right person in their team.”

Our participant-observation data confirm this. At one point, Adrian needed to set up and test a specific component in the RTC product. The development team in charge of that component was located on a different site. However, since Adrian had previously had the opportunity to establish relationships with some of the developers of that team through a previous research study, it was easier for him to contact them and ask

for help in his setup task.

Our survey data corroborate these observations. It suggests that the developers that form relationships (on a personal or work level) with other developers gave more emphasis on previous experiences with them. For instance, it made a difference whether the developers *are currently working together* (Table XVIb). Additionally, whether they have shared work experiences seems to be a more determinant factor than having *met in person* or sharing the same *physical location*.

This last finding would seem to indicate that the RTC team has managed to overcome obstacles created by geographical distance. Developers state that the physical location of a change-set author is not an important factor to seek more information about the change-set. Nevertheless, we should note two things. First, this finding merely points out that developers have no greater need to inform themselves of work output of their remote peers than of their local peers; it says nothing about the ease with which such information is acquired. Second, the RTC team has determined to carry out as many of its interactions as possible through textual, electronic media. This neglects the natural advantages provided by proximity in favour of uniform accessibility of interactions, no matter the physical location of the interlocutors.

4) *Risk Assessment*: A central factor at the heart of the decision to seek information is a concern with the quality of the product and components developed by the team at large. Both managers and developers share this concern. Because of it, every team member is constantly evaluating whether there are significant risks involved in accepting a change-set and including it into the final product. This concern is greatest close to releases or major milestones, and less important when the team is in normal iteration mode.

Developers request information more frequently about change-sets that touch on code that has a high customer impact. According to our interviews and observations, code parts with a high customer impact are those that are directly related to frequently used features or changes to the API that might be used by customers to customize the RTC product. In the case of the impact of API changes, there is an extensive knowledge exchange in the jazz.net forums between customers and developers about how to use the API to build extensions to RTC, which gives developers plenty of information about the possible impact of API changes to the customers. Consequently, change-sets that modify code that has less impact on the customer are of less concern and less likely to be discussed. For example, when scanning fixes to reported bugs, developers perform a risk assessment to determine whether the bug fix is even needed. In the words of one team member, “*with every tenth bug fix you introduce another bug to the system, so unless the bug does not have a workaround and is something most customers would experience, we give it low priority.*”

The ranking of items in Table XVIc provides additional insights into the kinds of change-sets that carry a risk for developers. A *changed API* comes at the top of the list, code that *affects frequently used features* comes third, immediately followed by changes to *complex code*, changes that *introduced new functionality*, and changes to code that *is used by many other methods*.

5) *Work Allocation and Peer Reviews*: In the later stages of the development cycle, developers have two process-based constraints that prompt them to interact with their peers. First, they need their peers to review their change-sets before they are included in the product. Second, they need an approval to start working on a task (a triaging process needs to take place so that developers are allocated to the most important work items).

In the change-set review process, there are two ways the change-set author and the reviewers interact. Either the change-set author submits his change-set for review, or the change-set author discusses the change-set with the reviewer before submitting it for review. The main discriminant between meeting with the reviewer to discuss the change-set before submitting it or submitting without prior discussion seems to be whether the reviewer and the author are co-located.

Adrian’s mentor during his time with the RTC team explained: “*It is very demotivating to get a change-set rejected, that’s why we usually try to discuss things before and then the reviewing just becomes all about quickly testing and accepting the change-set.*” Of course, this does not prevent an already discussed change-set from being rejected, but it signals that the process-based constraint is not as strict as to forbid discussions about a change-set between authors and reviewers in the interest of an unbiased code review.

In the case of developers seeking approval to start working on a task, the process is similarly flexible. The approval is supposed to be issued before the developer begins work on the task. However, we observed several times that developers performed quick fixes, and later went to their team leads to acquire approval for the fixes they had already performed.

Several items in our survey point to the importance of information-seeking around the processes of approval and peer review. In Table XVIa, the second- and third-highest items were information needed because the respondent needed to *review the change* or to *approve the change*. The need to *obtain a review for the change* or to *obtain an approval for the change* were of moderate importance. *Verifying a fix* was not judged as important by our respondents.

6) *Type of Change*: The three main reasons for a developer to commit a change-set are to (1) do feature development, (2) fix bugs, and (3) re-factor. Among them, feature development seems to be the one that prompts developers to seek information the most. In our survey (Table XVIa), code that *introduced new functionality* is ranked fifth among the code-change related factors, while changes that consist of a *bug fix* and a *refactoring* ranked far lower on the same list.

This may be due to the fact that feature development has less tool support than bug fixing (which is supported by debugging tools and automated test frameworks) and refactoring (which is essentially an automatic process today). The limited tool support and the inherently difficult nature of the task itself makes change-sets that introduce new functionality more difficult to understand or assess. Hence, developers are more likely to request information about them.

7) *Business Goals vs Developers’ Pride*: Senior team members (managers and team leads among them) often ask about the purpose and the real need of a change-set in order to

minimize unpleasant surprises that might be introduced with further code changes, and simply to have an economically feasible product. One senior development lead told us: *“often I need to stop my developers from fixing every little bug otherwise we would never be able to ship.”*

Although every team member is concerned with the quality of their product, a more realistic, business-oriented perspective seems to be more prominent in the senior team members’ information-seeking behaviour: they may request clarifications on the business case of change-sets or work items if they are not immediately clear in the existing documentation.

More junior team members, however, do not have this concern. They express their pride in the actual product and want it to be as bug free as possible. A developer commented to us: *“I want to be proud of what I deliver and I am fairly certain if we don’t fix it now it will come back to haunt us because it might upset some customers.”* Not only does this lead the developer to try to convince her manager that including change-sets for seemingly unimportant fixes is valuable, it also makes her information-seeking behaviour different with respect to that of the more senior team members. Broadly speaking, managers will be more interested in seeking clarifications as to the business case of a work item if it is not clear in its description, while developers will not care as much about this.

C. Conclusions

We end this chapter by bringing it back to the initial research question we set out to answer:

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

The findings we presented in Section IX-B show that there are different factors that influence developer to inquire about change-sets. Although not directly asked for, the developer with reporting on those factors answered our research question in a positive way. Our approach would provide developers with recommendations either on a per change-set level or while they are working on the code. We found that developer do talk about change-sets which represent our level of recommendations as well as they show interest in build outcome towards the end of a release cycle. Thus we can say that the level of our recommendation seems to be appropriate when keeping it to the change-set level. Examining our finding about risk assessment, which is practices by every lead and developer we are confident that developer would welcome notifications on how risky with respect to build success the changes they are currently applying are to give them a chance to either talk to the recommended person or reconsider the change altogether. Hence, we are confident the developer answers both validated that they would accept recommendations both in the form we present them, namely per change-set submitted, as well as lending our outcome metric more credibility as they show a keen interest in build success when the date to ship the product approaches.

In the next chapter we return to our path of a more direct analysis of our recommendations by showing a proof of concept that the recommendations do not only statistically

relate to build outcome but that recommendation could actually prevent build failures.

X. DISCUSSION

In this chapter we discuss the approach and how the research we presented in the last three chapters support it (Section XI). Followed by our second contribution which takes the form of the individual insights gained by the five studies we presented (Section XII). Before we end this thesis with concluding remarks as well as some future work (Section XIV) we discuss the threats to validity (Section XIII).

XI. AN APPROACH FOR IMPROVING SOCIAL INTERACTIONS

We derived the approach presented in Chapter VII through two case studies that investigate the usefulness of social and socio-technical networks to predict build outcome (Chapters V and VI). We conducted a study to see if the approach can generate relevant recommendations in Chapter V. The studies we conducted in the subsequent Chapters IX and ?? 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. The approach we presented in Chapter VII consists of five steps:

- 1) Define scope of interest.
- 2) Define outcome metric.
- 3) Build social networks.
- 4) Build technical networks.
- 5) Generate actionable insights.

In Chapter 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 (Chapter VI).

In those two studies we already applied the first four steps of the approach presented in Chapter 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 Chapter III in both Chapter V and VI. Chapter 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.

The three studies we presented in Part ?? of this thesis focused on the last step to *generate actionable insights*. The study in Chapter VIII 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 Chapter 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 [31] 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.

Through an in-class study with several students in Canada and Finland we investigated whether we can collect the necessary data to compute recommendations at the appropriate time as well as if those recommendation actually could prevent builds from failing (Chapter ??). We recognize that a student project is substantially smaller in size and complexity than an industrial project such as Rational Team Concert, nevertheless, the student project was to work on an existing open source project that is used by several companies. Furthermore, the in class study gave us the chance to see the possible effect of our recommendations more clearly as the smaller complexity allowed build to fail for less reasons and therefore making the impact of the recommendations clearer.

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 Chapter III). 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.

XII. CONTRIBUTIONS THROUGH EMPIRICAL STUDIES

Each study by itself contributed to the overall body of knowledge of software development team coordination. We present in the order of our five research questions the contribution of each study.

A. Using Build Success as Communication Quality Indicator

We started our investigation by exploring whether there exists a relationship between build success and communication by using prediction models (Chapter V). Our models can be used by Jazz teams to assess the quality of their current communication in relation to the result of their upcoming integration. If a team is currently working on a component and an integration build is planned in the near future, the measures of the current communication in the team can be provided as input to our prediction model and the model will predict whether the build will fail with a precision shown

in Table III. For example, if team P is working towards a build and our model predicts that the structure of its current communication leads to a failed build, the team can have a 76% (see Chapter V Table III) confidence that the build is going to fail. This information can be used by developers in monitoring their team communication behaviour, or by management in decisions with respect to adjusting collaborative tools or processes towards improving the integration.

B. Unmet Coordination Needs Matter

The relationship between communication structure and build failures however significant has only a small effect on the overall success rate of software builds, the outcome metric we studied. This lead us to include information about the system by adding technical dependencies as expressed by the source code among software developers. Backed up by findings in the research area of socio-technical congruence we hypothesized that the technical relationships help to zero in onto the important relationships among developers that relate to build failures. As the relationship between socio-technical congruence and productivity suggested influence on software quality, we showed in Chapter VI that it actually predicts build failures with varying accuracy depending on the type of build. Thus not meeting coordination needs as demanded by technical dependencies among software developers has a negative effect on build success.

C. Developers That Induce Build Failures

Being able to predict whether a build fails already helps developers to plan ahead with respect to future work, such as stabilizing the system in contrast to working on new features, but ultimately we want to be able to prevent builds from failing. To that purpose we need to influence the socio-technical network such that it takes a structure that is more favourable to build success. We found that certain constellations within a socio-technical network, to be more precise pairings of software developer and their respective relationship, seem to be correlating with build success (Chapter VIII). This evidence can be used to recommend action before the build is commenced in the sense that developers can investigate their relationship by for example discussing the code changes that created a technical relationship between them.

Thus, our findings have several implications for the design of collaborative systems. By automating the analyses presented here we can incorporate the knowledge about developer pairs that tend to be failure related in a real-time recommender system. Not only do we provide the recommendations that matter to the upcoming build, we also provide incentives to motivate developers to talk about their technical dependencies. Such a recommender system can use project historical data to calculate the likelihood that an upcoming build fails given a particular developer pair that worked on that build without communicating to each other.

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 behaviour of the

pairs we found in Rational Team Concert. This may indicate developers that are harder to work with or too busy to coordinate appropriately, prompting management to reorganize teams and workloads. This would minimize the likelihood of a build to fail, by removing the underlying cause of a pair to be failure related. Similarly, as another example from our study, most developer pairs consisted of developers that were part of different teams. In such situations management may decide to investigate reasons for coordination problems that include factors such as geographical or functional distance in the project.

D. Recommender System Design Guidelines

In our first qualitative study (Chapter IX) we explored whether developers would accept recommendations produced by our approach. It turns out, that developers are generally open to recommendations on a low level, such as on a change-set basis, but it depends on external factors such as the development process. For instance, we found depending on how close a development team to a software release is the more they focus on the implications of individual changes, whereas developers focus more high level reusability issues at the beginning of a release cycle.

Nakakoji et al [69] formulated nine design guidelines for systems that support seeking information in software teams. Some of them deal with minimizing the interruptions experienced by the developers who are asked for information, while others refer to enabling the information-seeker to contact the right people. Our findings help us refine Nakakoji et al guidelines:

a) *Guideline #1: Recommender systems should adjust to the development mode.* 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 [111] and Ensemble [95] 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 reducing overhead further for developers, recommendation systems should either automatically adjust to the development mode or feature customizable templates that can easily be switched.

b) *Guideline #2: Recommender systems should account for perceived knowledge of other developers.* 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 [111] and Ensemble [95]. 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.

Implementing filtering mechanisms based on author characteristics such as experience and quality of previously delivered work can help developers focus on the information that is important to them.

c) *Guideline #3: Recommender systems should assist in non-implementation tasks such as code reviews and risk assessment.* We observed, as described in the fourth, fifth, and sixth findings, that developers are highly engaged in discussions when performing risk assessments or reviews of change-sets.

In software engineering, most recommendations are focused on providing information to support concrete tasks such as bug fixes or re-factorings, but not for tasks such as reviews. To provide information for non coding tasks, recommender systems should be configurable to display relevant information beyond the tasks that they are intended to support, so that developer can easily access the information provided by recommender systems when performing code reviews or risk assessments.

d) *Guideline #4: Recommender systems should account for business goals.* Our last finding points to internal conflicts within teams and among developers caused by the desire to create a flawless product under the restriction of a set of business goals such as shipping the product on time. Thus, developers often need to be reminded that they must focus their efforts on fulfilling business goals rather than on polishing the product as they see fit. Existing recommenders that use code-related metrics such as quality or productivity may shift attention away from fulfilling business goals.

To support developers to focus on business goals, systems supporting the information-seeking behaviour of developers should be able to prioritize information related to tasks that are mission-critical to the organization, helping the team focus its attention on the most relevant problems for the upcoming release.

E. STC in real time

Knowing that socio-technical congruence lends itself to produce actionable knowledge that has an acceptable form to support developers in the wild, leads us to our last study (Chapter ??). In this study we showed the feasibility of generating recommendations at the right time, by gathering

data to generate socio-technical congruence in real time. Thus, we showed that socio-technical congruence can be used in real time to create actionable knowledge that might be of use to developers.

XIII. THREATS TO VALIDITY

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

A. External Validity

In part of this work we draw on information from observational studies (Chapters IX and ??) and studies relying on development repositories (Chapters V, VI, and VIII) 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 an 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 prevent 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 necessarily 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 generated 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

Chapters VI and VIII demonstrated that constructing the socio-technical networks is feasible and in Chapter VIII 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 Chapter VIII and the actual build failures and to what extent 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 affects the development process. We presented in Chapter 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 (Chapter IX and ??) 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.

XIV. CONCLUSIONS AND FUTURE WORK

In this thesis we illustrated an approach to leverage the concept of social-technical congruence to generate actionable knowledge. This five step approach focuses on defining two key parameters up front: (1) the scope of interest and (2) the outcome metric of interest. The first parameter scope helps with constructing the social networks (the third step) and constructing the technical networks (the fourth step) by supporting the selection of the best data sources. The outcome metric guides the analysis to produce actionable knowledge in the form of indicators that positively or negatively influence the outcome metric (step 5).

The work presented in this thesis lend it self 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. Molken 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-Oriented 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 Committees 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] M. Cataldo, J. D. Herbsleb, and K. M. Carley, "Socio-Technical Congruence: A Framework for Assessing the Impact of Technical and Work Dependencies on Software Development Productivity," in *Proceedings of the Second International Symposium on Empirical Software Engineering and Measurement*. New York, NY, USA: ACM, 2008, pp. 2–11.
- [24] G. Valetto, M. Helander, K. Ehrlich, S. Chulani, M. Wegman, and C. Williams, "Using Software Repositories to Investigate Socio-technical Congruence in Development Projects," in *Proceedings of the Fourth International Workshop on Mining Software Repositories*. Washington, DC, USA: IEEE Computer Society, 2007, p. 25.
- [25] K. Ehrlich, M. Helander, G. Valetto, S. Davies, and C. Williams, "An Analysis of Congruence Gaps and Their Effect on Distributed Software Development," in *Proceedings of the First International Workshop on Socio-Technical Congruence*, 2008, pp. 1–10.
- [26] I. Kwan, A. Schröter, and D. Damian, "Does socio-technical congruence have an effect on software build success? a study of coordination in s software project," to appear in *IEEE Transactions on Software Engineering*, 2011.
- [27] C. Bird, N. Nagappan, H. Gall, B. Murphy, and P. Devanbu, "Putting It All Together: Using Socio-technical Networks to Predict Failures," in *Software Reliability Engineering, 2009. ISSRE '09. 20th International Symposium on*, nov. 2009, pp. 109–119.
- [28] D. D. I. Kwan, "A Weighted Congruence Measure," in *2nd International Workshop on Socio-Technical Congruence STC'09*. Vancouver, Canada: in conjunction with ICSE 2009, May 19 2009, pp. 1–10.
- [29] F. Bolici, J. Howison, and K. Crowston, "Coordination without Discussion? Socio-Technical Congruence and Stigmergy in Free and Open Source Software Projects," in *Proceedings of the Second International Workshop on Socio-Technical Congruence*, 2009, pp. 1–10.
- [30] K. Blincoe, G. Valetto, and S. Goggins, "Proximity: A Measure to Quantify the Need for Developers' Coordination," in *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, ser. CSCW 2012. New York, NY, USA: ACM, 2012, pp. 1351–1360. [Online]. Available: <http://doi.acm.org/10.1145/2145204.2145406>

- [31] 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>
- [32] V. Maraia, *The Build Master: Microsoft's Software Configuration Management Best Practices*. Addison-Wesley Professional, 2005.
- [33] B. Curtis, H. Krasner, and N. Iscoe, "A Field Study of the Software Design Process for Large Systems," *Communication of the ACM*, vol. 31, no. 11, pp. 1268–1287, 1988.
- [34] R. E. Kraut and L. A. Streeter, "Coordination in Software Development," *Communications of the ACM*, vol. 38, no. 3, pp. 69–81, March 1995.
- [35] M. Cataldo, M. Bass, J. D. Herbsleb, and L. Bass, "On Coordination Mechanisms in Global Software Development," in *Proceedings of the 2nd International Conference on Global Software Engineering*, 2007, pp. 71–80.
- [36] 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.
- [37] 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.
- [38] A. Sarma and A. van der Hoek, "Towards Awareness in the Large," in *Proceedings of the International Conference on Global Software Engineering*, 2006, pp. 127–131.
- [39] J. D. Herbsleb and R. E. Grinter, "Splitting the Organization and Integrating the Code: Conway's Law Revisited," in *Proceedings of the 21st International Conference on Software Engineering*, 1999, pp. 85–95.
- [40] J. D. Herbsleb, A. Mockus, and J. A. Roberts, "Collaboration in Software Engineering Projects: A Theory of Coordination," in *International Conference on Information Systems*, 2006, pp. 59–69.
- [41] S. R. Fussell, R. E. Kraut, J. Lerch, W. L. Scherlis, M. M. McNally, and J. J. Cadiz, "Coordination, Overload and Team Performance: Effects of Team Communication Strategies," in *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*, ser. CSCW 1998. New York, NY, USA: ACM, 1998, pp. 275–284. [Online]. Available: <http://doi.acm.org/10.1145/289444.289502>
- [42] R. E. Grinter, J. D. Herbsleb, and D. E. Perry, "The Geography of Coordination: Dealing with Distance in R&D Work," in *Proceedings of the International Conference on Supporting Group Work*, 1999, pp. 306–315.
- [43] C. R. B. de Souza, D. Redmiles, L.-T. Cheng, D. Millen, and J. Patterson, "How a Good Software Practice Thwarts Collaboration: The Multiple Roles of APIs in Software Development," *Software Engineering Notes*, pp. 221–230, 2004.
- [44] W. E. Souder, "Managing Relations Between R&D and Marketing in New Product Development Projects," *Journal of Product Innovation Management*, vol. 5, no. 1, pp. 6–19, 1988.
- [45] M. B. Pinto and J. K. Pinto, "Project Team Communication and Cross-Functional Cooperation in New Program Development," *Journal of Product Innovation Management*, vol. 7, no. 3, pp. 200–212, 1990.
- [46] A. Griffin and J. R. Hauser, "Patterns of Communication Among Marketing, Engineering and Manufacturing—A Comparison Between Two New Product Teams," *Management Science*, vol. 38, no. 3, pp. 360–373, 1992.
- [47] R. Burt, *Structural Holes: The Social Structure of Competition*. Harvard University Press, August 1995.
- [48] L. C. Freeman, "Centrality in Social Networks: Conceptual Clarification," *Social Networks*, vol. 1, no. 3, pp. 215–239, 1979.
- [49] 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.
- [50] 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.
- [51] S. Wasserman and K. Faust, *Social Network Analysis: Methods and Applications*. Cambridge University Press, November 1994.
- [52] 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.
- [53] 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.
- [54] 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.
- [55] 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.
- [56] 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.
- [57] R. M. Bell, "Predicting the Location and Number of Faults in Large Software Systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 340–355, 2005.
- [58] 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>
- [59] 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.
- [60] S. Sawyer, "Software Development Teams," *Communication of the ACM*, vol. 47, no. 12, pp. 95–99, 2004.
- [61] C. R. B. de Souza and D. F. Redmiles, "An Empirical Study of Software Developers' Management of Dependencies and Changes," in *International Conference on Software Engineering, Leipzig, Germany*, May 2008, pp. 241–250.
- [62] 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.
- [63] 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.
- [64] M. A. Cusumano and R. W. Selby, "How Microsoft builds software," *Communication of the ACM*, vol. 40, no. 6, pp. 53–61, 1997.
- [65] J. D. Herbsleb, "Global Software Engineering: The Future of Socio-technical Coordination," in *Future of Software Engineering in conjunction with ICSE 2007, Minneapolis, USA*, 2007, pp. 188–198.
- [66] 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.
- [67] S. Marczak, D. Damian, U. Stege, and A. Schröter, "Information Brokers in Requirement-Dependency Social Networks," in *Proceedings of the 16th International Conference on Requirements Engineering*, vol. 0. Los Alamitos, CA, USA: IEEE Computer Society, 2008, pp. 53–62.
- [68] K. Ehrlich and K. Chang, "Leveraging Expertise in Global Software Teams: Going Outside Boundaries," in *Proceedings of the First International Conference on Global Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 149–158.
- [69] K. Nakakoji, Y. Ye, and Y. Yamamoto, "Supporting Expertise Communication in Developer-Centered Collaborative Software Development Environments," in *Collaborative Software Engineering*, A. Frinkelstein, J. Grundy, A. van der Hoek, I. Mistrik, and J. Whitehead, Eds. Springer-Verlag, 2010, ch. 11, pp. 219–236.
- [70] 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.
- [71] J. D. Herbsleb and A. Mockus, "An Empirical Study of Speed and Communication in Globally Distributed Software Development," *IEEE Transactions on Software Engineering*, vol. 29, no. 6, pp. 481–494, 2003.
- [72] S. Faraj and L. Sproull, "Coordinating Expertise in Software Development Teams," *Management Science*, vol. 46, no. 12, pp. 1554–1568, 2000.
- [73] 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.
- [74] 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.
- [75] 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.
 - [76] 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.
 - [77] 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>
 - [78] 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>
 - [79] 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.
 - [80] 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.
 - [81] 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.
 - [82] 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>
 - [83] N. Ducheneaut, “Socialization in an Open Source Software Community: A Socio-Technical Analysis,” *Computer Supported Cooperative Work*, vol. 14, no. 4, pp. 323–368, 2005.
 - [84] K. C. Desouza, Y. Awazu, and P. Baloh, “Managing Knowledge in Global Software Development Efforts: Issues and Practices,” *Software, IEEE*, vol. 23, no. 5, pp. 30–37, 2006. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1687858
 - [85] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, “Recurring Bug Fixes in Object-Oriented Programs,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE 2010. New York, NY, USA: ACM, 2010, pp. 315–324. [Online]. Available: <http://doi.acm.org/10.1145/1806799.1806847>
 - [86] T. Zimmermann and N. Nagappan, “Predicting Defects with Program Dependencies,” in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 435–438. [Online]. Available: <http://dx.doi.org/10.1109/ESEM.2009.5316024>
 - [87] M. Pinzger, N. Nagappan, and B. Murphy, “Can Developer-Module Networks Predict Failures?” in *Proceedings of the 16th International Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 2008, pp. 2–12.
 - [88] A. Meneely, L. Williams, W. Snipes, and J. Osborne, “Predicting Failures with Developer Networks and Social Network Analysis,” in *Proceedings of the 16th International Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 2008, pp. 13–23.
 - [89] G. M. Olson and J. S. Olson, “Distance Matters,” *Human-Computer Interaction*, vol. 15, pp. 139–178, September 2000. [Online]. Available: http://dx.doi.org/10.1207/S15327051HCI1523_4
 - [90] E. Trainer, S. Quirk, C. de Souza, and D. Redmiles, “Bridging the Gap Between Technical and Social Dependencies with Ariadne,” in *ETX ’05: Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange*. New York, NY, USA: ACM, 2005, pp. 26–30.
 - [91] A. Sarma and A. van der Hoek, “Palantir: Coordinating Distributed Workspaces,” in *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International*, 2002, pp. 1093 – 1097.
 - [92] A. Sarma, L. Maccherone, P. Wagstrom, and J. Herbsleb, “Tesseract: Interactive Visual Exploration of Socio-Technical Relationships in Software Development,” in *Proceedings of the 31st International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 23–33.
 - [93] A. Borici, A. Schröter, D. Damian, K. Blincoe, and G. Valetto, “ProxiScientia: Toward Real-Time Visualization of Task and Developer Dependencies in Collaborating Software Development Teams,” in *Proceedings of the 5th International Workshop on Cooperative and Human Aspects of Software Engineering*, ser. CHASE 2012. New York, NY, USA: ACM, 2012, pp. 932–935.
 - [94] M. Kersten and G. C. Murphy, “Mylar: A Degree-of-Interest Model for IDEs,” in *Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, ser. AOSD 2005. New York, NY, USA: ACM, 2005, pp. 159–168. [Online]. Available: <http://doi.acm.org/10.1145/1052898.1052912>
 - [95] P. F. Xiang, A. T. T. Ying, P. Cheng, Y. B. Dang, K. Ehrlich, M. E. Helander, P. M. Matchen, A. Emper, 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>
 - [96] J. van Gurp and J. Bosch, “Design Erosion: Problems and Causes,” *Journal of Systems and Software*, vol. 61, no. 2, pp. 105–119, Mar. 2002. [Online]. Available: [http://dx.doi.org/10.1016/S0164-1212\(01\)00152-2](http://dx.doi.org/10.1016/S0164-1212(01)00152-2)
 - [97] 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.
 - [98] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth, “Hipikat: A Project Memory for Software Development,” *IEEE Transactions on Software Engineering*, vol. 31, pp. 446–465, 2005.
 - [99] 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.
 - [100] 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.
 - [101] S. Siegel, *Nonparametric Statistics for the Behavioral Sciences*, 1st ed. McGraw-Hill Humanities, 1956.
 - [102] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 3rd ed. Springer, July 2003.
 - [103] D. Damian, S. Marczak, and I. Kwan, “Collaboration Patterns and the Impact of Distance on Awareness in Requirements-Centered Social Networks,” in *International Requirements Engineering Conference, New Delhi, India*, October 2007, pp. 59–68.
 - [104] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & Hall/CRC, 2000.
 - [105] A. MacCormack, J. Rusnak, and C. Y. Baldwin, “Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code,” *Management Science*, vol. 52, no. 7, pp. 1015–1030, July 2006.
 - [106] K. Schmidt and C. Simone, “Coordination Mechanisms: Toward a Conceptual Foundation of CSCW Systems Design,” *Computer Supported Cooperative Work*, vol. 5, pp. 155–200, 1996.
 - [107] C. Gutwin, R. Penner, and K. Schneider, “Group Awareness in Distributed Software Development,” in *Conference on Computer-Supported Cooperative Work, Chicago, USA*, November 2004, pp. 72–81.
 - [108] A. Mockus, R. T. Fielding, and J. D. Herbsleb, “Two Case Studies of Open Source Software Development: Apache and Mozilla,” *ACM Transition on Software Engineering Methodology*, vol. 11, no. 3, pp. 309–346, 2002.
 - [109] J. Aranda and G. Venolia, “The Secret Life of Bugs: Going Past the Errors and Omissions in Software Repositories,” in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, May 2009, pp. 298–308.
 - [110] A. Begel and B. Simon, “Struggles of New College Graduates in Their First Software Development Job,” in *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, ser. SIGCSE 2008. New York, NY, USA: ACM, 2008, pp. 226–230. [Online]. Available: <http://doi.acm.org/10.1145/1352135.1352218>
 - [111] 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>

PLACE
PHOTO
HERE

Adrian Schröter Biography text here.

PLACE
PHOTO
HERE

Daniela Damian Biography text here.