An Analysis of Congruence Gaps and Their Effect on Distributed Software Development

Kate Ehrlich
IBM Research
katee@us.ibm.com

Mary Helander
IBM Research
helandm@us.ibm.com

Giuseppe Valetto
Drexel University
valetto@cs.drexel.edu

Stephen Davies
IBM Software
sdavies@ie.ibm.com

Clay Williams
IBM Research
clayw@us.ibm.com

ABSTRACT

Software projects are frequently distributed across multiple sites. While this practice takes advantage of local expertise and differential cost structures, there is a hidden coordination cost. Modularity and even agile methods have had mixed success in addressing distributed development. Recent studies on congruence provide a new way to understand coordination on software projects. We used a variation of congruence that focuses on gaps in communication to examine coordination in distributed development teams. Using archival data from a distributed software project we found that an increase in the number of gaps was associated with an increase in code changes. We also found that distributed pairs of developers had more gaps than collocated pairs. However, brokers, who communicated with people across multiple sites, had significantly fewer gaps than other developers. We discuss the implications of these findings in terms of opportunities to close gaps.

Keywords
Congruence, social networks, software development, distributed teams

1. INTRODUCTION

For the past several years it has become common for US companies to staff software development projects offshore. A recent study reports that more than 70% of US firms have outsourced some kind of business process. But, cross-site communication and coordination issues can cause substantial delay and increase the amount of time for individual work [26, 27]. Distributed work items can take more than twice as long as collocated work items to complete, in part because time zone differences reduce opportunities for real-time collaboration, and response time increases considerably when working hours at remote locations do not overlap [36]. Distance reduces the communication richness when people experience problems that they try and resolve with communication tools rather than face-to-face [38]. As a result, members of distributed teams are less likely than members of co-located teams to perceive themselves as part of the same team [25].

It is clear from existing research that distributing work across multiple sites may save in labor costs but incurs coordination costs which may negate the other savings. Software projects have tried various approaches to overcome effects of distance and bring teams closer together. However, as documented in previous research e.g. [24], these methods often require a greater investment in planning for risks and contingencies than developers are willing to provide. Importantly, the methods underestimate the importance of informal communication in negotiating decisions on a project [24].

Several studies have emphasized the importance of informal communication for coordination [26, 28, 35]. However, these and other studies have primarily focusing on interpersonal communication rather than explicitly tying the communication to demands of the task domain.

Recent research on congruence in software teams offers a new approach to informal communication in distributed teams [8]. That research combined data on software dependencies and task assignments to determine coordination requirements between pairs of people. Coordination requirements were met when people communicated with each other thereby establishing congruence between the social and technical dimensions of work. Additionally, there was a performance benefit of congruence; when coordination requirements were met it took less time to fix bugs. Similar findings of congruence and performance benefits have been reported for product development teams [39, 40].

In our previous research [45], we extended [8] by using graph-theoretic methods to focus on those inter-personal communications that are most important for congruence. As a result, we are also able to rank communication gaps, according to their potential contribution to the congruence measure. A gap denotes a lack of communication between two developers who work on the same artifact, or on inter-dependent artifacts, and therefore have a need to coordinate their work. Since this approach can identify and prioritize gaps, it enables the team to focus remedial actions on the high priority gaps that need to be closed, rather than on generally improving communication and collaboration across the whole team in an effort to improve coordination.

This paper reports the results of an empirical study that used archival data from a distributed software team to address three issues: (1) The ranking of gaps in terms of size (number of gaps) and their relationship with performance; (2) Distribution of gaps amongst collocated and distributed developers; (3) The role of brokers, here called brokers, in reducing gaps.

The topic of gaps brings together several strands of research from the social network and software engineering literature on team communication and coordination as well as recent research on congruence. We first review that research and then describe the current study and results. The final section of the paper considers a variety of remedial actions that might be taken to close gaps.

---

1 Copyright 2008 IBM Corp. Permission to copy is hereby granted provided the original copyright notice is reproduced in copies made.
2. PREVIOUS RESEARCH

2.1 Informal communication in software teams

Despite advances in automated tools, technologies and methods, software development remains a largely human process in which informal communication plays a critical role. Software engineers spend a significant amount of time in communication and coordination activities. When an individual is not collocated with coworkers, there are fewer opportunities for spontaneous, informal interaction [33]. Moreover, coworkers tend not to form close relationships with distant colleagues, trust them or make charitable attributions about them [12, 26, 27]. Lack of informal interactions also increases the risk of communication breakdowns that result in asymmetrical information distribution among sites, further exacerbating coordination problems and lack of team spirit [7]. Distributed work impacts communication by making shared social contexts more abstract [31], which in turn influences the quality and level of shared understanding associated with communication [12]. Several studies have suggested that distributed work is accomplished most effectively through spontaneous, informal communication since it fosters awareness, spontaneous conversation, project focus, problem solving, and the development of social relationships [29, 31, 35]. When informal interactions are established, they enhance trust and rapport between remote counterparts [30].

2.1.1 Social Network Analysis

Given the importance of informal communication for collaboration, coordination and knowledge acquisition, it is helpful to understand what factors may play a role in determining “who talks with whom” especially in software teams. Here we turn to studies that have used social network analysis (SNA) to gain insights into the myriad of relationships that influence communication patterns.

Social network analysis provides a rich and systematic means of assessing informal networks by mapping social relationships, or ties, among people [13], and by demonstrating the importance of factors such as awareness and accessibility on communication [14, 18]. Awareness and access both decline over distance which is one reason for reduced communication across geographic locations [15, 28, 31].

In general, software teams rely on their social networks to help coordinate their work [9, 28], locate expertise [18], and share information [16]. Researchers have also used social networks to understand the dynamics of software teams (for an overview, see [46]) and represent the complex nature of coordination [19, 42].

Within the social network framework, research has recognized the importance of brokers for building the social capital necessary to foster improved communication and collaboration [5, 6]. Brokers also play an important role in transferring knowledge and information within and across boundaries [2]. In particular, research has shown that complex, technical information is best transferred using strong ties [23]. Thus, brokers represent a new and interesting way to consider facilitating coordination especially in distributed teams.

2.2 Congruence

Socio-technical congruence is a quantitative measure that assesses the “fit” between the structure of a software system and the structure of the organization that develops it [11]. Recent work by Cataldo et al. [8] established a means to test Conway’s Law by introducing a mathematical definition to compute the congruence measure as a scalar between zero and one with one signifying a fully congruent socio-technical structure. This research postulates that highly congruent organizations may enjoy a performance advantage in their software development task, with respect to less congruent teams, and presents initial statistical evidence confirming that hypothesis.

In our previous work [45], we looked at congruence by constructing and examining a graph that captures the structural dependencies among modular software elements, the personal interactions among members of the development team, and the work actions on the software. Our method to measure congruence is mathematically equivalent to the matrix-based method described in Error! Reference source not found.. However, since it traverses the graph representing the socio-technical structure of the project, our algorithm is able to isolate “patterns” of interest in the graph, that is, topological instances that show coordination requirements between the members of the development team (see Section 3.3 for a more thorough discussion of coordination requirements). Of particular interest are graph patterns that we call gaps, i.e., those instances of coordination requirements that show absence of communication. Our previous research proposed a method to rank gaps with respect to their contribution to the overall congruence metric, were they filled.

A number of recent studies have sought to apply the concept of socio-technical congruence to various software engineering problems, mainly related to distributed software development. In [3], for example, Design Structure Matrices representing product modularity are employed as a unified model of social and technical dependencies and a means to optimize them in the direction of better congruence. Damian and her colleagues [16] also discuss implications of (lack of) congruence in distributed teams on team members’ awareness, during the requirements engineering phase.

3. STUDY METHOD

3.1 Research setting and participants

The data for the present study were collected from a team of 27 people charged with developing a web-based Learning Management System in Java Enterprise Edition. Of the 27 people, the majority, 15, classified themselves as developers. In addition there were was 1 project manager, 3 architects, 4 technical leads, 2 build engineers as well as a data loader, a DB expert and a tester.

The history of the project included the addition of a small group based in the mid-west and two other US groups on the East coast who were responsible for the initial release of the project prior to its move to Ireland which became the main site after the initial release. The data we collected represented development after the initial release. The project manager was on the east coast, the architects were based in the mid-west and the developers were distributed across all sites.
The principal method of cross-site communication was through weekly or bi-weekly meetings which were conducted throughout the duration of the project. Remote developers were always expected to check-in to these meetings. The usual format of this meeting was about an hour long, and usually new bugs and PMR’s were discussed as well as new features, important dates and other administrative details. In the development part of the meeting, complicated bugs were often discussed and would very often be moved between developers if say one developer owned up to being more knowledgeable in a particular area. Generally speaking, an SPR could only formally be handed over between developers with the agreement of the manager or the project manager.

One of the outcomes of these meetings were follow-ups which was one of the main ways in which cross-site developer-developer communication occurred. The communication was controlled to a degree in that if an SPR was handed over between developers, the manager or project manager would make a note of the handover and the fact it was agreed between the 2 developers. This was sometimes done very informally and sometimes done very formally depending on the importance of the SPR and its urgency. For instance, a developer might agree to take a look at an SPR without actually agreeing to take on the SPR, but with the intention of coming back with a list of suggestions on how to fix the bug.

### 3.2 Data collection

#### 3.2.1 Software

To study socio-technical congruence, we analyzed the work carried out by the development team by means of traces persisted in its various software repositories.

We collected archival data from the commit log history from the source management system, and the change request database, for the period from Jan 2002 to July 2005. We also had access to the entire code base of the project. We used these information sources to construct a socio-technical software network for the project [45]. Such a network has the following three components:

- Dependencies between software artifacts, which are mapped as directed arcs between nodes representing software;
- Communication interactions between developers which are mapped as undirected links between nodes representing people;
- Task assignments, which are mapped as directed arcs between the people nodes and the software nodes and account for the work done by developers on the software artifacts during the course of the project.

In this case study, we constructed the socio-technical network by using a combination of techniques. For the software dependencies we used static dependencies between source code artifacts involved in the J2EE application. Other studies have chosen to construct the representation of artifact interdependencies in different ways, for instance by considering what artifact consistently appear together in the same “change sets”, as made evident by commit logs [48]. We have opted for static code analysis as it leads to a more persistent and exhaustive representation of artifact dependencies. However, our approach does not depend on the technique chosen for this particular step.

The source code of the application is largely written in Java. We used the open source tool Dependency Finder [1] for the automated extraction of dependencies from the Java code which results in a total of 2456 artifacts, all Java classes. That is a significant portion of the overall code base, which also includes other types of artifacts, such as: XML configuration files and database setup and migration scripts. Those artifacts, however, could not be consistently processed together with Java files, since it is hardly possible to trace code dependencies between Java files and those other artifacts, nor is it possible to establish any semantically equivalent relationships across those heterogeneous artifact types.

For the representation of the work assignments, we mined the commit log history of the source management system used in the project (Microsoft Visual Source Safe), which consists of 47834 commit actions for the period taken into consideration.

Finally, we took advantage of the Change Request database in two ways. First, since the development team had embraced the convention of inserting the unique ID of each Change Request (CR) record into the commit log of source code modifications that were carried out to satisfy that CR, we have been able to use that piece of traceability information to associate those code modifications, where available, (and their collective size) to each of the 800+ resolved CRs we have examined. Furthermore, we have processed the CR records to try and accurately estimate the time required to complete each of them. To that end we have examined the workflow entries associated with each resolved CR. Each CR underwent a process of triage, analysis, assignment, development work, and solution verification: that process is recorded by time stamped entries that are an integral part of its record. By looking at those entries, it should be possible, in principle, to isolate periods of active development work within the interval in which a CR was open, as opposed to periods in which no development work was carried out on the CR. However, the granularity with which events are recorded in this particular project made it difficult to obtain a clear and precise picture of the workflow from relatively noisy data. We are therefore still validating the data on CR completion time, and looking at reasonable heuristics in that respect.

One of the authors of this paper, who was a full-time member of the team, has provided us with access to the archival data and been available throughout the course of the study to help us produce all the information we have mined, as well as validate the data and our interpretation of it. That “insider knowledge” has been extremely important in a study of this kind, to help formulate sensible hypotheses, and confirm them or disprove them based on the reality of the project and the process practices embraced by the team.

#### 3.2.2 Communication

Since we did not have access to contemporaneous traces of communication for this project which could have been mined, we collected communication data from responses to a survey. The survey, which was administered to the team of 27 people around April 2007, allowed us to capture data on different types of communication. We had complete responses from 20 of the 27 (74%). Almost all the missing data resulted from developers who had been on the team during the time of our data collection, but who had subsequently left the company. We used a roster method
to elicit responses from each team member on 3 network questions.

- **TeamWork.** Please indicate how closely you worked with this person on project Sigma. (Dichotomized at highest level response = “we worked together on related parts of the project”)
- **Communication.** Please indicate how frequently you communicated with this person for any reason when you worked together (Dichotomized at medium level, response = “we communicated several times a month”)
- **Files.** Please indicate how frequently you worked on shared files with this person. (Dichotomized at medium level, response = “I worked with this person on shared files a few times”)

We computed a new measure of communication by combining responses from the 3 networks as follows. The data from each individual network were dichotomized as indicated above to represent what we considered a minimal acceptable relationship. Since we were not interested in the direction of a tie, and each relationship can reasonably be considered symmetric, we symmetrized the data such that as long as one person indicated a connection, we inferred that the connection was reciprocated. This method was also used to reduce effects of the missing responses by seven team members. We then combined these three networks such that each cell had a value ranging from 0 to 3 which indicates that the pair worked together on related parts of the project AND communicated several times a month AND worked with the other person on shared files at least a few times. We dichotomized the data from this combined matrix to form 3 levels of communication

- **GE1.** Two people were counted as communicating if at least one of the 3 conditions was true
- **GE2.** Two people were counted as communicating if any 2 conditions were true
- **GE3.** Two people were counted as communicating if all 3 conditions were true.

The resulting communication matrix is therefore a composite of several ways in which there could be a relationship and thus is broader than any one communication medium. When combined with the data on related files, we generate evidence of congruence – when the communication mirrors the artifact dependency structure, or gaps when there is no communication. Unless stated otherwise, we used an absence of congruence at GE1 as a conservative measure of a gap since this measure asserts that there is communication at all.

### 3.3 Study Variables

**Communication:** The degree of communication was computed by combining the data from the survey questions as described earlier.

**Location:** Members of the team were located at four sites. People were regarded as collocated if they worked at the same site otherwise distributed. We computed collocation from a similarity matrix of location.

**Coordination requirement:** A coordination requirement exists when there is a dependency between software artifacts and each artifact can be traced to a developer in our study through a work assignment. A coordination requirement was scored as present or absent and computed using the measures described in [45]: any communication requirement that shows a missing communication arc between the involved developers is counted as a gap (see below). An example of a pattern in the socio-technical graph that signifies a coordination requirement is shown in Figure 1. Nodes Sa and Sb represent software artifacts that have a dependency, while nodes Pi and Pj represent members of the team, who are assigned to development tasks impacting Sa and Sb respectively. The dotted line between Pi and Pj signifies a coordination requirement: a communication arc between Pi and Pj would satisfy that requirement, whereas the absence of an arc would result in a gap.

**Gaps:** A gap corresponds to a coordination requirement not being fulfilled by communication. We computed the number of gaps using the algorithm described in [45]. These gaps were aggregated to pairs of developers, to individual developers and to individual software artifacts. Unless stated otherwise, our analyses are for developer pairs only.

**Gap rank:** Gaps are ranked according to their potential contribution to the congruence measure, in case they were filled by means of communication. It is possible to rank any element of the graph (either an arc or a node), by simply counting the number of gaps in which it is involved. In this study, we have found convenient to consider the rank of communication arcs: an arc with a high rank is involved in a large number of gaps; therefore, by closing it, many coordination requirements would be fulfilled, and congruence would be correspondingly increased.

**Performance:** We measured performance by the degree of change in a commit history file. It was computed by taking the difference between versions and then eliminating all non-semantic changes. The degree of change can be interpreted as a proxy for the amount of development.

### 4. RESULTS

There were 351 unique pairs in our sample of 27 people. A total of 204 pairs (58%) participated in at least one instance of a coordination requirement. The rest of our analyses are limited to these 204 pairs unless otherwise noted.
4.1 Gap importance

Across the 204 developer pairs, there were, on average, 35.5 instances of a coordination requirement. In 11.5 of these instances there was a gap and in 24 cases there was no gap.

We are interested in the distribution of gaps based on their rank. By focusing on rank we expect that a few pairs of developers will participate in a large number of gaps while the majority will participate in a few gaps. As shown in Figure 2 below, this was indeed the pattern that merged. The majority of pairs (92%) had fewer than 25 gaps while only 7 pairs had more than 100 gaps (3%). Although this finding may not be surprising, it provides important justification for a targeted strategy to improve communication amongst a few people in a software team who have a high number of gaps than a strategy of trying to improve communication amongst a broader range of people.

We also expect gaps to negatively affect performance. We tested the effect of gaps on performance by comparing the number of gaps for each pair of people against the degree of code change in the software artifacts. Because of the high variability in the range of gaps and the range of code changes, we converted both measures to a log scale. The scatterplot is shown in Figure 3. The correlation between gaps and code changes was significant (QAP correlation = 0.445, p < 0.01) supporting our claim that as the number of gaps increase there is a corresponding increase in the amount of code change. A correlation between the number of gaps and the amount of code change, computed for the individual software artifacts, was also significant (Pearson correlation = 0.46, p <0.01).

4.2 Effects of distributed development

Gaps are defined as the absence of communication amongst pairs for whom there is a coordination requirement. Given the generally lower communication amongst distributed pairs [26, 27, 32], we would expect to find more gaps for distributed pairs. Table 1 shows the number of gaps and congruence (absence of gaps) for distributed and collocated pairs. There was a significant correlation between being distributed and the presence of a gap (QAP correlation = 0.39, p < 0.01) indicating that coordination requirements likely to result in a gap for distributed than collocated pairs. Subsequent investigation of the data revealed that one person, P 17, had contributed a large number of gaps and had not responded to the survey. A reanalysis with that person removed did not change the result (QAP correlation = 0.41, p < 0.01). Distributed pairs were not only more likely to have a gap but they had more proportionally more gaps than collocated pairs as shown in Figure 4. The difference in the total number of coordination instances was significant (F 1,202 = 11.43, p < 0.001); the difference in gaps was not.

4.3 Brokers

The negative effects of being distributed can sometimes be mitigated by routing communications through brokers who occupy a special position in the social network by connecting people across boundaries. Previous research [9] suggested that distributed software teams may form somewhat

---

2 As social network data are not independent and do not satisfy assumptions of statistical inference for traditional regression, we use a special procedure called QAP (Quadratic Assignment Procedure) [3, 20] which uses re-sampling to compute the correct standard error.
independent sub-groups that are linked together by people who are playing a brokerage role. In that study, we found that the brokers were often architects or tech leads rather than project managers or developers.

Using the single communication network from the original survey, we applied a brokerage routine [21] to identify the top 5 people who were strongly connected to others across multiple sites. These 5 people are shown in green in Figure 5 below. The other colors represent the four sites. Two of the people, P10 and P7 were especially strong as brokers. Both were architects on the project. P22 was the project manager. P5 and P13 were part of the main site in Ireland. P5 was a technical lead and P13 who had the least connections across multiple sites was the only developer who was also playing some brokerage role. These data suggest that people who are brokers are often in roles which require a broader outlook.

Following a recent study on brokerage [8], we also examined whether brokers had an advantage over other developers in generating fewer gaps. We computed a “gap rate” by dividing the number of gaps by the total number of instances for each developer. Using this measure, our 5 brokers had a gap rate of 27% compared with a gap rate of 53% for the remaining 19 developers. As before, we dropped P17 from this analysis. The difference between brokers and non-brokers was significant (t-test = 1.72, p = 0.05). In line with the previous study, we also computed a measure of constraint [5] for each developer. This shows the extent to which each person is connected to other people who themselves are connected. The lower the constraint metric the more interconnected someone is. However, the correlation between constraint and gap rate was only marginally significant (Pearson correlation = 0.31, p < 0.10) due, perhaps, to our small sample size; only 24 people provided both constraint and gap rate data.

Another way of looking at the impact of brokers is by comparing the performance of pairs which included a broker, with pairs that did not. Table 2 and 3 below shows the number of instances where a coordination requirement results in a gap or no gap for pairs with and without one of the five brokers we identified earlier. There was a significant effect of brokerage on gaps in both distributed (Chi-squared = 15.9, p < 0.001) and collocated pairs (Chi-squared = 7.2, p < 0.01) but the data were more pronounced for the distributed pairs. These data clearly show that coordination between developers working on related artifacts is less likely to result in a gap when one of those developers is a broker.

Following the groundbreaking work of Cataldo et al [8] who provided an operational definition of congruence, we have argued that relationships between work artifacts – software dependencies – drives communication and coordination.

<table>
<thead>
<tr>
<th></th>
<th>Gap</th>
<th>No Gap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broker</td>
<td>20</td>
<td>45</td>
</tr>
<tr>
<td>No Broker</td>
<td>37</td>
<td>18</td>
</tr>
<tr>
<td><strong>Table 2: Number of coordination instances for distributed pairs</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Gap</th>
<th>No Gap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broker</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>No Broker</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td><strong>Table 3: Number of coordination instances for collocated pairs</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5. DISCUSSION

It is widely recognized that poor communication is at the root of many problems in software and other product development teams [22, 24-26, 39, 40] and that these problems are exacerbated when teams are distributed [3, 12, 30, 36]. But what does that really mean and what is the appropriate response by the team? Following the groundbreaking work of Cataldo et al [8] who provided an operational definition of congruence, we have argued that relationships between work artifacts – software dependencies – drives communication and coordination.

![Figure 5: Combined communication network (at GE2)](image)
Based on the earlier research, we reformulated the computations to focus on gaps which occur when pairs of people who have one or more coordination requirements fail to communicate [45]. By ranking gaps according to their size, we can also identify which are the most important gaps to close and focus remedial actions on those developers. As expected, we found that performance declined as the size of the gap increased. We also found that gaps increased for distributed pairs, despite a greater preponderance of coordination requirements for colocated pairs. This extends previous work on congruence to cover distributed teams. Finally, we began to explore the role of brokers as an alternate method of information transfer. Brokers, sometimes called brokers, are people who, by virtue of their formal role or personal preferences, have ties to people outside their immediate group. In our study, that grouping was defined by geography. As individuals, the brokers had fewer gaps than others. More interestingly, when a coordination requirement included a broker there was less likely to be a gap than when there was no broker. In other words, partnering with a broker may help close some gaps. Given these findings we next explore remedial actions that might be taken to close or reduce the number of gaps.

5.1 Closing Gaps
Software teams have a variety of options for improving team performance ranging from better communication, to software refactoring, to planning and scheduling of development activities. The analysis of and the selection among those options require knowing how to balance their costs and risks against their potential benefits [44].

Considering communication gaps, we consider three promising approaches that address communication issues without necessarily adding more people or more communication. While closing gaps by having the involved team members communicate directly may seem like the obvious solution, it may not be always advisable, as pointed out by Brooks [4], or even feasible.

5.1.1 Improving awareness
Highly-ranked gaps indicate communication paths in the organization that are missing, and may be critical for project execution. One way to close these gaps is by improving awareness of the dependency relationship or simply improving awareness of what others on the project are currently doing. Ariadne [42], an Eclipse plug-in, collects information about technical dependencies and work assignments which are displayed as network diagrams variously showing combinations of the technical dependencies and the work assignments. Where other systems might just provide a list of developers who have worked on particular code, Ariadne displays the code modules the other developer has used. By displaying dependencies, Ariadne seeks to help developers find the right person to talk to about a particular piece of code.

Another approach is to provide developers with tools that offer general awareness of what others are working on. Systems such as Palantir [37] provide individual developers with awareness of their immediate collaboration context. Other research has focused on supporting individual team members in establishing the right connections with their colleagues, based on the kind of project-related knowledge and experience they need in order to carry out their assigned tasks [34]. Similarly, Li-Te Cheng and colleagues developed a lightweight synchronous communication tool as part of an integrated Eclipse environment to help individual developers maintain awareness of their colleagues’ current work to facilitate rapid and contextual communication [10].

Tools that increase awareness can affect coordination and reduce gaps by helping developers understand what others are currently doing and in so doing stimulate communication around related work. These tools, whether visualizing dependencies directly or simply providing a representation of what others are working on, become especially important in distributed teams where awareness of remote developers is impaired.

5.1.2 Coordination through shared artifacts
Not all instances of coordination gaps need to be resolved by direct communication especially where there is no negotiation involved. Software artifacts can be leveraged to provide a means of indirect communication. For instance, application programming interfaces (APIs) are commonly used as a way to hide implementation details but keep access open for extension and adaptation. In a series of studies, Cleidson de Souza [41, 43] has explored how APIs are used to facilitate collaboration and coordination. An advantage of an API is that it segments the work and establishes a clear contract. As long as the segmentation makes sense, there is a common shared understanding of the rationale behind the API and the API doesn’t change, there may be no need for additional direct communication between the developers. However, a reliance on APIs as an intermediate collaboration object breaks down when there are unanticipated changes in the API.

It is also possible to facilitate indirect information sharing through searchable knowledge repositories about the project and its byproducts to facilitate indirect knowledge sharing across a socio-technical network, as discussed by Ye [47]. Since the collection and maintenance of exhaustive project documentation and knowledge is notoriously hard and expensive to attain at a project-wide scale, this kind of effort should rather be directed where it is most needed. Gap analysis and ranking can become useful for that purpose: priority should be given to knowledge creation and sharing regarding tasks and artifacts that are involved in highly-ranked gaps, especially in case the alternative of directly closing or eliminating those gaps is difficult to achieve, by intervening on the organization structure, or the software modular decomposition, or the project schedule.

5.1.3 Indirect communication through brokers
Finally, when gaps cannot be closed, or are too expensive or risky to close directly, there may be ways to improve indirect communication especially for distributed teams for whom congruence is both more of a problem and more of a challenge to solve. Brokers represent a structural solution to addressing issues of information transfer. As noted by other researchers, distributed software teams may need a different information and governance structure than colocated teams [28].

Brokers don’t just provide a channel for communication they can provide additional value through interpretation and filtering of information as well as recommendations to additional sources [2, 17]. In software teams, brokers are often architects or others who have senior technical roles in the team. However, brokers as we have defined them here, emerge within the structure of the team. Not all brokers are architects and not all architects are brokers. In fact, previous research has noted that brokers (or intermediaries)
are often invisible in the organization [17] suggesting that it is not sufficient to redefine a formal role without confirming the informal, network role.

Research on congruence is still in its infancy. It is therefore not especially surprising that there are as yet only a few remedies that might be expect to improve congruence and close gaps. Moreover, none of these approaches has been tested in the field in a way that demonstrates improved performance. As we continue to evolve a research agenda for congruence, one of the challenges will be to link remedial actions to performance.

5.2 Caveats

We close with some comments about data collection methods associated with congruence. The normal challenges of collecting data from distributed software teams are compounded for studies on congruence. In addition to collecting data from software repositories for software artifacts, researchers must also collect data about work assignments and communication. This latter can be especially problematic if there are no archival records of communication that took place at the same time as the work. There are various approaches that might be taken to acquire those data or compensate if they are not available. In this study we collected communication data through a survey instrument. This has the advantage of allowing us to tailor the questions so that we can be more accurate on what kind of communication data we are collecting. Another advantage of surveys is that the communication is not specific to a particular method or tool. This is important in software development since effective "communication" can take place indirectly through the artifacts or brokers, as discussed above.

One of the disadvantages of our survey, however, is that the data is retrospective and not directly linked with what the communication is about. Furthermore, we don’t have direct and conclusive evidence of whether pairs of people communicated about their related artifacts; in other words, we cannot obtain fine-grained information of the work context in which that communication has taken place. We addressed this shortcoming by using a combined measure from the survey. Our focus on gaps is also less sensitive to accuracy of communication, since it is dependent on the absence of communication. However, we recognize that this retrospective method is not optimal. Fortunately, as team-oriented development environments, such as Jazz [20], begin to get adopted, and integrate tools for communication tools, task management and artifact configuration management, communication data - along with the other data that can be used to compute congruence - will become at the same time more readily available and more easily contextualized.

6. ACKNOWLEDGMENTS

We extend our appreciation to the software team we studied and to our colleagues who provided comments and advice on this study and the paper.

7. REFERENCES

1. Dependency Finder.


34. Minto, S., Murphy, G.C.: Recommending Emergent Teams. The 4th International Workshop on Mining Software Repositories (MSR’07), Minneapolis MI (2007)


