Building a Socio-Technical Theory of Coordination: Why and How (Outstanding Research Award)

James Herbsleb
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
+1 412 268 8933
jdh@cs.cmu.edu

ABSTRACT

Research aimed at understanding and addressing coordination breakdowns experienced in global software development (GSD) projects at Lucent Technologies took a path from open-ended qualitative exploratory studies to quantitative studies with a tight focus on a key problem – delay – and its causes. Rather than being directly associated with delay, multi-site work items involved more people than comparable same-site work items, and the number of people was a powerful predictor of delay. To counteract this, we developed and deployed tools and practices to support more effective communication and expertise location. After conducting two case studies of open source development, an extreme form of GSD, we realized that many tools and practices could be effective for multi-site work, but none seemed to work under all conditions. To achieve deeper insight, we developed and tested our Socio-Technical Theory of Coordination (STTC) in which the dependencies among engineering decisions are seen as defining a constraint satisfaction problem that the organization can solve in a variety of ways. I conclude by explaining how we applied these ideas to transparent development environments, then sketch important open research questions.

CCS Concepts

• **Software and its engineering→Software creation and management**

Keywords

Coordination; socio-technical theory of coordination; collaboration; empirical studies; global software development; open source; transparent environments

1. INTRODUCTION

Coordination has always been one of the fundamental problems of software engineering: if the work of individuals in teams and organizations does not mesh in just the right way, the product will not work as intended. This is true of any product, but the difficulty seems greater with software, for the reasons that Brooks pointed long ago [1] – especially its invisibility and constant change.

Coordination becomes particularly challenging – and interesting as a subject of study – when organizational forms morph, evolve, or innovate. When people organize in a habitual, consistent way, for example, in collocated teams, it is easy to overlook day-to-day coordination mechanisms that are simply taken for granted. It is easy to see the importance of things such as meetings of various flavors, processes, methods, and architectural separation, which have long been studied. Other, subtler mechanisms such as informal communication, practices, habits, and shared mental models are often only made visible by their absence.

Very interesting – and often disturbing – things happen when an organization is geographically split apart. Much can be learned by observing the mayhem that often ensues when organizations are distributed, and much is revealed about what must have been happening in the collocated case that keeps such chaos more or less at bay. Adding new tools and practices in these novel organizational contexts, and seeing how the work is impacted, also helps to deepen our understanding of what coordination is and how to achieve it.

In this paper, I summarize two decades of research that colleagues and I have carried out to understand and sometimes to facilitate how work is carried out via novel and evolving organizational forms, driven by factors such as geographic distribution, collaboration in open source project communities, and open ecosystems.

The story begins with qualitative studies that throw out a wide net in order to understand the experience and difficulties of global software development (GSD) – teams operating across geographic, time zone, national, and cultural barriers. The focus shifts to quantitative studies to validate qualitative results and take a close look at one of the primary difficulties that surfaced from early results – the developers' experience that multi-site work takes much longer than comparable work at a single site. This leads in turn to a focus on finding and engaging the right people, the specific problem our quantitative results pointed to [2].

These empirical results guided our efforts to find solutions, as we developed resources and tools to assist in the development process, and evaluated them in situ. In particular, we developed an early chat tool [3, 4], an expertise location tool [5], descriptions of practices that organizations had found helpful [6, 7], and organizational models describing various ways to distribute work across sites along with their strengths, weaknesses, and criteria for when each is appropriate [8].

Another organizational form – open source development projects – caught our attention during this period. It appeared to us to be an extreme form of geographically distributed development, loosely and informally organized; yet it appeared to be free from many of the problems we observed in industry. We performed two case studies of very different communities, Apache and Mozilla, to try to understand how this new form successfully

accomplished development work [9]. In addition to findings about how the work was performed, the quality of the code, and the timeliness of support, we developed a number of hypotheses that have been tested in subsequent work.

To try to make sense of the wealth of results coming from the great variety of approaches to software development, and the varied success they achieved, we formulated a socio-technical theory of coordination [10, 11]. In essence, it views each development project as posing a particular constraint satisfaction problem that an organization must solve. I summarize several studies that serve as tests of this theory, and sketch out research questions to explore the theory and the phenomenon of coordination more comprehensively.

Finally, I will discuss our recent work on what is rapidly becoming a dominant development paradigm, ecosystem-based development. Unlike the individual open source projects, ecosystem-based development characteristically has large numbers of highly interdependent projects that must continuously coordinate. Transparent environments help developers to cope both with the scale and decentralized organizational structure, in order to take advantage of the tremendous resource pool of libraries, frameworks, and other code available in these environments. [1]

2. PROBLEMS OF GSD

Projects in Lucent Technologies experiencing conflict, misunderstandings, missed schedules, and technical issues of many kinds provided business motivation for the Bell Labs Collaboratory, a research project on GSD. Although researchers were beginning to understand why "Distance Matters," [12], the symptoms of dysfunction still presented a puzzle. Lucent (formerly AT&T) had been producing highly reliable telephony products for many years, yet it seemed that spreading development efforts across sites was shockingly disruptive. More or less the same technical work as in the past, highly qualified people, adequate budgets, yet an unprecedented level of problems. In the rest of this section, I describe our empirical studies and their increasingly sharp focus as our results began to point to delay and its causes in the difficulties in finding and engaging the right people across sites.

2.1 What's Going on Here?

Our first efforts were to initiate a qualitative, open-ended study designed to understand why things were coming off the rails. We chose two sites to work with, and visited each with the research team to introduce ourselves and to kick off qualitative data collection in the form of interviews with developers, managers, and executives, eventually expanding our scope into a qualitative study of the disastrous first release of the software for the organization's primary product [6, 7]. Based on some initial interviews and conversations with participants, we focused on the integration phase, where the problems most visibly burst into the open. We developed a rich set of findings detailing the ways in which failures of communication, differing assumptions, misunderstandings, mistrust, incompatible tools and environments led to incompatible actions across the sites and major delays as the problems are identified and fixed.

Fundamental to the problems were a lack of awareness about who knows what, is responsible for what, and is doing what across the sites, along with the near-total absence of regular informal communication which could unearth the "unknown unknowns" of key information one doesn't know one lacks. We also noted how even limited face to face contact seemed to counteract these problems, allowing subsequent distributed work to proceed more fluidly. This led to a number of recommendations about communication practices, architectural separation, assigning a liaison role, and managing uncertainty.

2.2 How to Organize?

Recognizing that different product groups within the company seemed to manage GSD rather differently, we expanded on this qualitative work by visiting and conducting interviews and artifact analysis of six different geographically distributed projects [8]. From this work, we identified four different organizational models that projects seemed to be using to distribute work across sites, identifying the benefits, problems, and typical coordination mechanisms for each model. The models were distinguished based on the principle by which they separated the development work across sites: by component, by process step, by functional area of expertise, or by core versus customization work. No model provided a "best" overall solution, nor did any appear in pure form. But each appeared to have some contextual conditions that favored and others that argued against its use.

For example, one strategy involved separating a product into its core functionality, developed and maintained at a large, central site, and customization centers around the world located near important customers. This strategy only worked when this style of modularization made technical sense, and when appropriate development resources were available in the right locations. It had the advantage of giving full technical responsibility for the largest and most complex component to a single organization that took responsibility for its maintenance and integrity. It also allowed the customization teams to get to know the customers well and gain a deep understanding of their requirements. It had several significant problems, however, as customization teams felt ignored by the core developers, who tended to make changes with little regard to how they would impact each custom project. Expertise in the core technology was sparse in the customization sites, and they had difficulty getting sufficient attention from the core to ask questions and get their problems solved.

2.3 Distance, Delay, and Social Networks

One of the most striking results from our qualitative work was the consistency with which our participants reported how GSD seemed to result in substantial and sometimes crippling delay in development. We designed a set of studies to get a better handle on the extent and possible causes of delay [2, 13, 14]. Our approach was twofold. First, we took advantage of the fact that all development work in the company we worked for was undertaken pursuant to a modification request (MR). By looking at the geographic location of each person associated with the MR, we could distinguish work that was distributed across sites from work that occurred all at a single site. We also developed and deployed a survey instrument that assessed communication patterns and social networks within and across sites.

We expected that GSD would take somewhat longer than collocated work, but we were taken aback by the magnitude of the difference we observed. On average, a work item that had

[1] I focus here very egocentrically on the work that colleagues and I have done. There is, of course, a much larger literature full of major contributions by others, but space prohibits reviewing it here. A more comprehensive review is in preparation.

participants from more than one site took about 2.5 times as long to complete. Fitting a graphical model, we statistically controlled for a number of factors that could have produced spurious results – for example, the size and diffusion (i.e., number of changes and number of files changed) of same site versus multiple site work items. Such differences in the work itself could not account for the delay we observed. We were also surprised by the consistency of this figure – data taken from two different Lucent development organizations exhibited almost precisely the same ratio. Data from our survey in which we asked (among many other things) if developers had recently experienced a delay and how long it took to resolve, gave us nearly the identical figure of intervals 2.5 times as long for delays involving multiple sites.

Digging in to possible explanations for this dramatic difference revealed further surprises. Most unexpectedly, our model showed no direct relationship between the number of sites and how long the work took. The effect we were observing was a mediated relationship involving the number of people involved in the work item. Distributed work items had a strong tendency to involve more people, and the number of people was a very strong predictor of how long it would take. Returning to our qualitative data from our previous studies, there were several possibilities, all based on lack of knowledge of expertise and current workloads, which could explain this connection. The MR owner, for example, might assign work incorrectly at another site, leading to additional assignments. Or perhaps the first assignee had expertise to do only part of the work, again leading to additional assignments. Our prior research [6, 8] suggested that knowing who to contact about what, the difficulty of initiating communication, and issues about the effectiveness of cross-site communication could all play a role.

In addition to this analysis of archival data to assess the extent and causes of delay issues, we conducted two surveys of one distributed development organization in order to better understand social networks and the frequency and intensity of interaction across sites as compared to within a single site. Confirming most of our hypotheses, developers communicated with many fewer people at other sites than at their home site, and communication was much less frequent. Our participants also found it much more difficult to identify and communicate with appropriate experts across sites, and overall received much less information from them. Moreover, cross-site colleagues were much less likely to perceive themselves as part of the same team, or to share goals.

3. GSD SOLUTIONS

As our empirical studies began to clarify the origin of problems with coordination and the resulting delay, we proposed and implemented technical and organizational solutions targeted to these difficulties. I will focus here on two tools that saw significant use, and practices that arose with their use.

3.1 Expertise Browser

Finding someone with specific expertise – in tools, technologies, or parts of a product – is a serious problem in distributed organizations, and as our results showed, caused very substantial delays in accomplishing technical work. The amount of experience, i.e., the number of software changes accomplished with a given tool, technology, or in specific project part, can be used as a serviceable approximation of expertise. This insight led to the design and development of the Expertise Browser (ExB) [5], a socio-technical visualization tool that was deployed and used by several GSD organizations.

ExB used linked displays to show a hierarchical technical view of a software product (from subsystem to file), and a social view of the supervisors, developers, and organizations that performed the work. Clicking on some unit in the technical view, representing, e.g., a file or module, would produce a filtering and ordering of the people, organization, and supervisor views to reflect their relative contribution to the code unit. Clicking on one of the social panes, e.g., a specific developer, would highlight in the code view the proportion of contributions that person had made to each visible unit of code. Thus, the tool could be used to find an expert or to explore what work was performed by individuals or organizations.

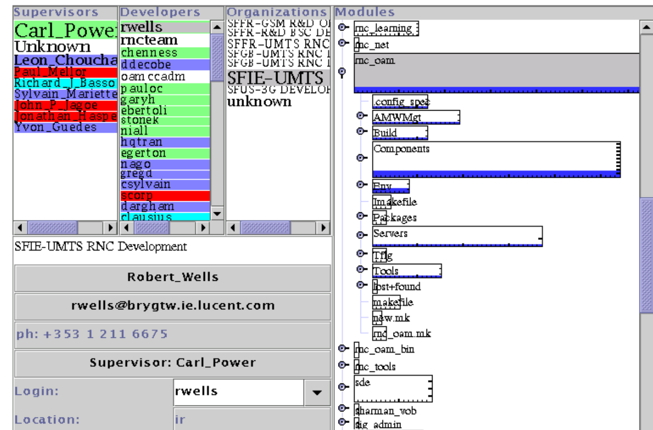

Figure 1. Expertise Browser user interface, with code units represented on the right and individuals, teams, and organizations on the left.

Our logs showed that different sites tended to use the tool somewhat differently. Smaller and newer sites tended to use ExB for locating experts, while older and more established sites seemed more often to use ExB to explore where other parts of the organization were working in the code. We also found a strong desire for a view of recent activity, to enhance awareness of potentially conflicting work that other sites were doing, and for a "quantitative resume" that would give a profile of a developer's work, including languages used, code volume, and organizations. Today, GitHub profiles provide highly developed visualizations of these kinds of information.

3.2 Rear View Mirror

In order to try to increase communication frequency and effectiveness, as well as addressing the absence of cross-site informal communication, we designed, built, and deployed an early social media application (or, depending on the definition of social media you prefer, a precursor to social media). It combined person-to-person instant messaging, persistent group chat, and presence awareness (to see who was currently active). We called it "Rear View Mirror" (RVM) to express the ambition that it would provide an unobtrusive but always available way for developers to see what was going on around them, especially at other sites. Our research focused on two different aspects of introducing RVM: patterns, issues, and tactics for adoption [4] and content analysis to indicate how it was used [3].

While it is a bit hard to imagine now, chat still seemed fairly novel in 1999, especially in a work setting. As we released the RVM application to several parts of the development organization, we spent two weeks with research team members at two different sites simultaneously training developers on the tool. For the initial deployment we chose pairs of developers who

seemed to have the greatest cross-site communication needs, in hopes of achieving a critical mass of users quickly. The results were fairly disappointing, with about half using the tool initially, dropping off over several months to a steady 10%. In addition to the typical problems of an alpha deployment, interviews revealed some interesting issues. Our training strategy had not worked very well. We concluded that we should focus on teams, not just pairs of people within an organization. Where adoption happened, it was because a large share of a team began using the tool. We also realized we needed to train teams together, since they needed not just to learn how the tool worked, but also how to *collaborate* with it. With some engineering work to address usability issues and a change to team-focused training, we were able to boost adoption to 40-50% of newly-trained teams.

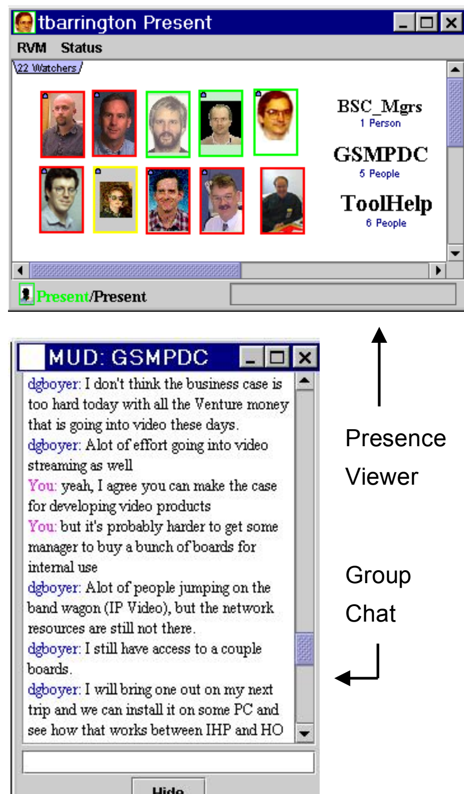

Figure 2. The presence awareness (above) and chat windows of the Rear View Mirror user interface.

We also looked at what teams were actually talking about with RVM [3]. The popular press of the day was deeply suspicious of chat and messaging tools in the workplace (e.g., [15]), seeing them as a source of interruption and distraction. While we did see small amounts of non-work content and occasional humor, the majority of messages (69%) were directed to accomplishing work. All the teams we examined showed a surprisingly consistent pattern of use, with very similar frequencies of the different types of messages. Since the tool maintained group message histories (for a limited duration), conversations were sometimes asynchronous, but most conversational turns happened in seconds or minutes. Messages tended to occur in bursts, with some days showing extensive use, and others little or none. Tool design involved several major tradeoffs, including avoiding intrusiveness versus timely notification, and customizable privacy settings versus setup time. All in all, RVM seemed to provide a means for a modest increase in communication across sites. One can see Slack and other tools providing a rich set of team-based functionality within the enterprise.

4. OPEN SOURCE: EXTREME GSD

As we were studying GSD inside the enterprise, the open source movement began to get attention as a serious competitor to the commercial development paradigm. Little was known at the time about how and why this extreme form of GSD seemed to work so well. Popular articles (e.g., [16]) reveled in the lack of structured process and provided various maxims (e.g., "release early and often"). Economists wondered about the incentives that led to voluntary work without pay (e.g., [17]), but we could find no research explaining how fully distributed development could eschew standard coordination mechanisms such as management oversight, plans, and specifications, yet produce great products, while experiencing few of the profound problems of GSD. Colleagues and I set out to try to understand this puzzle by extracting and analyzing a detailed history of the Apache server, using archival analysis techniques developed by Audris Mockus, and with the help of insights provided by Apache Group founding member Roy Fielding [9, 18].

The story we uncovered had a number of dimensions, but interestingly, it turned out that different and differently sized groups of developers performed the basic software development functions in a way that made coordination possible. A relatively small core group produced the vast majority of new functionality, while bug fixing was spread much more thinly across developers, and testing – i.e., submitting a bug report – was far more distributed yet. So the highly interdependent work of developing new functionality was coordinated informally among a small team of a dozen or so developers, while the much less interdependent work of testing and fixing engaged large numbers of people. Meritocratic selection of core members with commit rights, self-assigned release managers, mailing lists for communication, common ground in the form of a mutually understood (albeit informal) development process, and a voting mechanism for reaching decisions, together filled out the picture of how project coordination was achieved. Comparison to somewhat similar commercial projects hinted at impressive results: very high quality, rapid responses to problems, and high productivity [9].

Our observations of Apache led us to construct seven hypotheses that were speculative generalizations arising from our reasoning about the research literature and *why* we observed what we did in our Apache case study. For example, we speculated that in an open source project where the core group exceeded some threshold, perhaps 10-15 members, code ownership, not observed in Apache, would become necessary since larger groups would find it difficult to coordinate informally as a distributed team. As another example, we also reasoned that open source projects with a strong core team but with little participation beyond the core will be able to create new functionality but will fail because of insufficient resources devoted to finding and fixing bugs. As noted by Stol and Fitzgerald [19] the Apache case study was focused on the "substantive domain," i.e., striving to understand the particular phenomenon of open source. From the results, we developed what they aptly call "theory fragments," which we then used to guide the design of a second, concept-driven, case study.

We selected Mozilla as the subject of our theoretical replication, since it was very different from Apache on many dimensions, and would allow us to test a number of our hypotheses. Mozilla was much larger, had a commercial origin, a small paid staff, a roadmap document for planning, test teams, and a formalized code review process. Most of our hypotheses were supported; for

example, the Mozilla core team was much larger, and as expected, code ownership was enforced. We also modified and extended our hypotheses, as we observed other coordination mechanisms such as required code review and a more explicit development process, were also present and helped the project function at scale. We suggested, based on these theory fragments, that some open source processes, such as open work assignments, might be beneficial in commercial environments. We had the chance to try out some of these ideas, which experienced modest success [20].

5. THEORY FORMULATION AND TEST

Thus far, I have reported a collection of empirical studies aimed primarily at understanding a particular software development phenomenon:

How do developers coordinate their work?

Along with the empirical work on delay [13, 14] and open source [9, 18], tool interventions ([3-5]), as well as organizational models [6, 8] we achieved some level of understanding of how coordination was accomplished, when and how it failed, and deployed practices and tools to address the key practical question:

How can we improve coordination and overall project success?

The answers we found to both of these questions seemed like a bit of a hodgepodge. Many coordination techniques and tools were used, and it was clear that some organizations and teams were much more successful – and better coordinated – than others. Yet there did not appear to be any tools or practices, alone or in combination, that seemed always to produce good results. And while the various coordination mechanisms we explored seemed vaguely related, it was hard to express just how. For example, detailed specifications seemed sometimes to reduce the need for explicit coordination, but not always. A defined process sometimes seemed to improve communication, but there were exceptions. RVM and ExB were taken up and used enthusiastically by some groups, who gave fairly glowing accounts of how helpful they were, while other groups tried them and quickly abandoned them, or declined to use them at all because they didn't seem helpful.

These experiences pointed to the need for a theory that could help explain the relationships between all of these coordination mechanisms, give some account of how they were in some sense all addressing the "same" problem, why they were sometimes helpful and sometimes not, and to formulate questions and predictions about why they might combine effectively, substitute for each other, or conflict. This need for a unifying account led to our efforts to develop a theory of coordination in software engineering.

5.1 The Need for Theory

Theories, along with the empirical methods that lead to their development and testing, are the essence of science. Historically, in software engineering, we recognized the need for evidence to evaluate the claims we make about the impact of our technical contributions (e.g., [21]) long before we realized the limitations of a validation-centric, theory-free approach to querying reality. Accreting the results of empirical tests of claims about specific engineering contributions does not by itself add up to broad and enduring knowledge. It often seems to be the case that by the time we evaluate Development Tool A, and find it is superior under certain conditions to Development Tool B, someone has already proposed Development Tool C, and the evaluations of A and B do not, by themselves, give us any insight or evidence-based expectations about C.

When we concern ourselves with the question of how general our results are (i.e., their external validity), scientists tend to approach this in a handful of ways [22] (pp. 24-25), four of which are: generalize to cases that share some surface similarity, generalize across irrelevant differences, discriminate cases with relevant differences, and interpolate or extrapolate from known results. Each of these involves an implicit theory that informs the scientist of what dimensions of similarity matter, what conditions are irrelevant, what differences matter, and what observed cases say about those that lie between or outside them. If the implicit theory is wrong, these approaches produce invalid generalizations. The problem with implicit theorizing is that the theory is never really exposed, discussed, tested, or even specified.

The final and most substantial way of making a causal generalization is by means of an explicit theory that provides a causal explanation of the observations [22] (p. 25). The theory may not be fully articulated – theory fragments in Stol and Fitzgerald's [19] helpful terminology – but to the extent it is made explicit and the relation of the empirical observations to the theory is clear, explicit theory provides a basis for generalization to cases to which the theory applies. The more support a theory accrues, the firmer the basis for such generalizations becomes. Further studies, of course, are also likely to find boundary conditions beyond which the theory does not hold, and lead to modifications or even rejection of the theory in favor of one that provides a better fit to the evidence.

To inform the discussion that lies ahead, I'll adopt a simple definition of theory, realizing that many treatises have been written on the topic, and it is notoriously full of subtleties and philosophical land mines. For present purposes, it is enough to say a theory is (1) a set of *constructs*, or entities that enter into the theory, (2) a set of *relationships* that describe the ways in which the constructs are connected or interact, and (3) a *causal story* which explains how the constructs and relations give rise to observable phenomena of interest. A theory that is complete in some sense should have all these parts. Theories that are partly implicit and partly specified can be called theory fragments [19].

Scientists care about evidence, then, because of the support it provides (or does not provide) for a theory, i.e., it bears on whether a particular theory is true. More pragmatically, we might say that as evidence accumulates in favor of a theory: if we behave as if that theory were true we are less likely to be surprised by events in the theory's domain than if we did not have the theory. It makes the world more predictable by making it more understandable.

5.2 A Socio-Technical Theory of Coordination (STTC)

Building on prior theories of business process coordination [23, 24], distributed cognition [25, 26], distributed artificial intelligence [27], and drawing theory fragments from work on geographically distributed engineering [2, 6, 9], Audris Mockus, Jeff Robertson and I formulated what I now call a socio-technical theory of coordination (STTC) [10, 11].

In short, the theory conceives of coordination in software engineering as a distributed constraint satisfaction problem (DCSP) defined by the mutually-constraining engineering decisions for a project. People must organize to solve this problem by using capabilities and coordination mechanisms at their disposal. The better the match of the solution with the project-specific DCSP, the more effectively the project will be coordinated. This, in turn, should lead to higher quality (fewer

bugs from uncaught constraint violations) and better productivity (less time spent reworking decisions that violated constraints). We refer to this degree of match between the coordination problem and the organization's coordinating activities as congruence.

The origin of the theory [11] lies in a key observation from Hutchins's theory of distributed cognition [28]. In particular, we were inspired by Hutchins's notion that many problems that teams solve collaboratively, like the problem of navigating a ship at sea, have an irreducible core. Navigation is grounded in geometry and physics, and this grounding is completely independent of any particular problem-solving mechanism. Problem-solving systems, consisting of humans, technology, and practices, can vary dramatically. These variations, however, can only be understood and compared once one grasps how they address the core problem. For example, Hutchins [28] compares the radically different ways that navigation problems can be solved by naval officers using modern equipment or Pacific islanders using an entirely different theory of navigation and virtually no equipment. Both systems "respect" the physics of navigation problems, but have entirely different conceptual systems, practices, and tools for addressing the problem.

This view inspired us to try to characterize the "irreducible core" problem of coordination in software engineering, in order to see how different kinds of practices, tools, and processes are rooted in different ways of conceptualizing and addressing this core problem. As is the case with ship navigation, coordination among agents can be accomplished in many ways, but each solution strategy has an irreducible grounding in the decisions embedded in engineering tasks and their interdependencies [11].

Applying Yokoo's (2001) formulation, a software project consists of a large set of engineering decisions that must be taken in order to complete the project. Decisions are represented as n variables $x_1, x_2, \ldots, x_n$ whose values are taken from finite, discrete domains $D_1, D_2, \ldots, D_n$. Assigning a value to a variable represents making the decision represented by that variable [11].

A project has a set of constraints that operate over the variables that represent the engineering decisions. Given an assignment of a value for some variable, the constraints serve to limit possible values that can be assigned to other variables. Formally, constraints $p_k(x_{k1}, x_{k2}, \ldots, x_{kn})$ can be represented as predicates defined on the Cartesian product D_{k1} x D_{k2} x . . . x D_{kj}. Successfully completing a project is equivalent to finding an assignment for all variables that satisfies all constraints. [11].

In order to define a *distributed* constraint satisfaction problem, we define two relations (Yokoo 2001). Each variable x_j belongs to one agent i, represented as the relation *belongs*(x_j, i). In general, agents only know about a subset of the constraints. We can represent this relation as *known*(P_l, k), meaning agent k knows about constraint P_l [11].

Agents attempt to solve a DCSP by assigning values to variables and communicating with other agents. There are many standard algorithms for solving DCSPs, and much is known about their complexity, completeness, soundness, and performance in various constraint landscapes (see Yokoo, 2001, for an overview). Agent behaviors that give rise to these distributed algorithms differ in many ways, including what the agents communicate, when they communicate, with whom they communicate, how they decide the order in which to make decisions, and what they do when they discover a constraint violation. Since it is these agent behaviors that enable and define the various algorithms, DCSP provides a way to think about the relationship between overall project performance and the individual behaviors and communication patterns that give rise to this performance [11].

Better organizational performance – higher productivity, shorter development times, and higher quality – should result when there is a better match between the particular DCSP presented by a the engineering work and the coordination strategies adopted and applied by the development organization. We call the degree of this match socio-technical *congruence* [29].

5.3 Empirical Studies of Congruence

In order to test this theory, one has to measure the degree of congruence in a large number of items of software development work, and empirically test whether work that is more congruent is accomplished more efficiently and with higher quality. In this section, I provide a brief overview of how we accomplished this.

In order to measure congruence, we needed to characterize the topology of the dependency network and the topology of the application of coordination mechanisms in ways that would allow the degree of congruence between them to be measured. With respect to the dependency network, it is not feasible to try to fully capture all decisions and all constraints among them. Developers often make a great many decisions each day, and a complete account of the ways in which each decision constrains all other decisions would be exceedingly difficult to construct.

An appropriate aggregation of decisions, however, could perhaps provide a sufficient characterization to allow an empirical test. We could consider a file of source code to be a *clump* of decisions [10], each file being a node in an aggregated dependency network or undirected graph. Edges in the graph represent work dependencies between nodes, i.e., an edge between two nodes indicates that decisions in each node constrain, or have an *effect* [10] on decisions in the other node.

The dependencies between nodes could be measured in several ways, for example call graphs or data dependencies, but we have found logical dependencies [30] to be effective for our purposes [31, 32]. It is convenient to use a matrix representation of the task dependency network, $\boldsymbol{T_D}$ where rows and columns are nodes and cell entries are edge weights, reflecting a measure of logical dependency (i.e., the number of times two files have been changed together as part of the same work item [30]). With respect to the DCSP formulation, these dependencies provide an abstract representation of the set of *predicates* that express the constraints.

A graph representing the assignment of decisions to developers can be constructed in analogous fashion, once again aggregating decisions to the level of files. We construct a task assignment matrix $\boldsymbol{T_A}$ where each developer is a row i and each file is a column j, and the cell entry is the number of times developer i modified file j. In DCSP terminology, this provides an aggregated representation of the *belongs* relation.

The following matrix multiplication [29, 31, 33] allows us to construct a coordination requirements matrix, $\boldsymbol{C_R}$

$$\boldsymbol{C_R} = \boldsymbol{T_A} * \boldsymbol{T_D} * \boldsymbol{T_A}^{\boldsymbol{T}}$$

where $\boldsymbol{T_A}^{\boldsymbol{T}}$ is the transpose of the task assignment matrix. $\boldsymbol{C_R}$ is a square matrix where developers populate the rows i and columns k, and the cell entry reflects the extent to which developer i engages in work that has task dependencies with the work engaged in by developer k.

Actual coordination, or the use of a particular coordination mechanism by a pair of developers, can also be represented as a square matrix $\boldsymbol{C_A}$ where developers once again populate the rows

and columns, and each entry represents the extent of use of a particular mechanism by developer *i* and developer *k*. For our purposes, we have found binary cell values – a pair of developers did or did not use a particular coordination mechanism – to be sufficient. This is roughly equivalent to the *known* relation in the DCSP formulation – if two developers are actually employing a coordination mechanism, it is highly likely they know about the constraint.

Congruence, or the degree of match between $\boldsymbol{C_R}$ and $\boldsymbol{C_A}$, can be computed as the proportion of non-zero cells in $\boldsymbol{C_R}$ that are matched by actual coordination, indicated by a non-zero cell in the same location in $\boldsymbol{C_A}$. Congruence can range from 0, if *no* non-zero cell in $\boldsymbol{C_R}$ is matched by a non-zero cell in $\boldsymbol{C_A}$, to 1, if *every* non-zero cell in $\boldsymbol{C_R}$ is matched by a non-zero cell in $\boldsymbol{C_A}$.

The final step is to compute congruence for a large number of work items and construct a statistical model to assess the degree of association between congruence and desirable outcomes, particularly quality and productivity, while controlling statistically for the many other variables that can impact these outcomes. We did this originally in one commercial development organization, using a multiple regression model to assess the impact of four different coordination mechanisms on development speed [29, 34], and later assessing impact on code quality and replicating both results in a different commercial organization [33].

This empirical work provides support for STTC, but it really just scratches the surface. It has a number of important limitations. It looked only at the match between the people who needed to coordinate and their use of four different coordination mechanisms. It did not examine other coordination mechanisms (shared work history, offline communication, use of shared documentation, etc.) nor did it attempt to advance our understanding of what mechanisms are effective for what kinds of constraints. In fact, many more questions are raised by this work than are answered (which I take to be a good thing for a theory!).

Among the important questions:

- Coding is just one of many engineering tasks. What do dependency networks look like in this larger set of tasks, and how can we compute coordination requirements?
- Popular frameworks, libraries, and APIs undoubtedly impose structure on the task dependency networks of projects using them – can we capture this imposed structure and use it in various ways to facilitate coordination?
- Projects extend through time, and as decisions are made, the decision network changes, perhaps radically. What are these changes, and how can we recognize and accommodate this evolving structure with the coordination mechanisms at our disposal?
- How early in a project can we usefully predict coordination requirements and how can we use this information in planning?
- What is the full set of coordination techniques that development organizations can use?
- Can different techniques substitute for each other, e.g., relax use of a defined process if advanced collaboration technologies are used?
- How can we compose various coordination techniques to build a complete coordination solution for a given project?
- Given that we can compute or predict coordination requirements, how do we match them with appropriate coordination techniques for a given project?
- How do we know when it is appropriate to introduce particular coordination techniques to an ongoing project to address coordination issues, or to drop them when they are not needed?

5.4 STTC and Transparency

Social coding environments are introducing very substantial changes in how coordination happens, often accompanied by innovative processes, especially the continuous delivery model (see, e.g., [35]). In practice, continuous delivery often includes elements such as micro-service architectures, small teams, decentralized decision-making, requirements expressed as improving specific business metrics, and a DevOps approach to deployment [36]. There is also rapid growth in commercial use of open source software, which increasingly is developed and maintained in the context of a software ecosystem, or collection of related and interdependent projects (e.g., [37, 38].

New environments and life cycle models impact the nature of the engineering decisions, the task interdependencies that define the coordination DCSP, and the organizational capabilities and coordination mechanisms available to solve it. The constraints among engineering tasks that seem most critical from a coordination point of view are those that arise from dependencies among different repositories. A quick examination of dependencies in any sizeable project (e.g., by examining the package manager for the language in which the project is written) generally shows large numbers of dependencies, especially if one looks at the transitive closure. Since changes in any project one depends on, directly or indirectly, could impact one's project (i.e., violate one or more constraints), and since these other projects are under the control of other developers who can change them at will, the situation is much less predictable than traditional commercial environments that use techniques such as roadmapping to help ensure that code changes do not break code that depends on them.

In the face of constraints arising from widespread, diffuse, and largely "unmanaged" (in the traditional sense) dependencies, social coding environments are extremely useful. A central novel characteristic of these environments is *transparency*, or the "accurate observability, of an organization's low-level activities, routines, behaviors, output, and performance" [39] p. 181). Transparency is key to coordinating work where decision-making is decentralized and developers take a large share of responsibility for creating and managing dependencies [40]. Social coding environments allow explicit social media style connections to repositories and people, capture a detailed history of development activity in a repository and its forks in ways that are readily browsed, searched, and shared. Asynchronous communication is supported through comments on artifacts. Contribution to external repositories and code review are made simpler by a pull request mechanism. These environments and the necessity of managing change have given rise to practices and policies that support a number of quite distinct styles of coordination based in a community's values [41].

In a qualitative study of developers using GitHub [40], we studied the kinds of decisions developers made and how they related to the information available in the environment and the activities of other developers. We found that developers used several kinds of information in deciding whether to create a dependency on a project, including the recency and volume of activity, and whether pull requests were handled in a timely way. They used signals of attention, such as number of watchers and forks, to gauge the quality and importance of a project. They attended to commits to

identify potentially contentious or troublesome commits that might break their code. If breaking changes occurred, they often communicated with the owner of the breaking code, sometimes submitting a pull request to the external project to modify the code that was causing their problem.

External code submissions, often to provide fixes or enhancements desired by the users of one's code, presented the decision of whether to accept the code. In a quantitative study of pull request acceptance [42], we found that both following technical norms (e.g., include test cases, keep changes small) and having a social connection (e.g., submitter follows pull request closer, previously submitted pull request) substantially increased the likelihood of acceptance. Lengthy discussions, which often question either the intent or the solution quality of a pull request [43], sharply decreased the odds of acceptance, except when the submitter had a social connection with the project.

In future work, we plan to move toward a congruence approach, as we identify the ways in breaking changes propagate across repository boundaries, to see if we can establish how the various kinds of coordination mechanisms (e.g., comments, subscriptions via watching and starring, observing changes in forks) match up with different technical coordination problems and how they impact outcomes.

6. CONCLUSION

Coordination is one of the fundamental problems of software engineering. I have argued that it is a fundamentally socio-technical phenomenon, where one must take into account both the technical dependencies among engineering tasks, which collectively define the problem, and the ways that people organize to find a solution. This can be nicely characterized as a distributed constraint satisfaction problem, which I think captures the irreducible core of coordination in software engineering, allowing us to see the common underlying impact of all of the varied means of coordination, from software process and collaboration technology, to the coordination implications of traditional design strategies such as modular product structure and architectural styles. These are sets of practices with different underlying conceptual structures all addressing parts of the same irreducible core problem.

I think it is clear that theory is necessary in order for us to take a scientific approach to understanding the complexity of the pervasive role of humans in software engineering. Progress will be fragmented and it will be very difficult to cumulate results into a deeper understanding, unless our research is grounded in theory. Mary Shaw, in her eloquent keynote talk at ICSE 2016, in assessing the progress of software engineering toward a true engineering discipline, noted that engineering is preferentially based in science. The science we need, as I have argued elsewhere, [44], requires theory, and because of the fundamentally socio-technical nature of key phenomena, will also extend well into the human domain, being based as much on behavioral science as computer science.

7. ACKNOWLEDGEMENTS

This work was supported by NSF awards 1633083, 0534656, 0414698, 0943168, 1546393, 1111750, 1322278, a grant from the Alfred P. Sloan Foundation, The Google Open Source Program Office, IBM, Siemens, Bosch, Accenture, and the Center for the Future of Work, Heinz College, Carnegie Mellon University. I would also like to thank all of my extraordinarily talented collaborators.

8. REFERENCES

[1] Brooks, F. P. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20, 4 (1987), 10-19.

[2] Herbsleb, J. D. and Mockus, A. An Empirical Study of Speed and Communication in Globally-Distributed Software Development. *IEEE Transactions on Software Engineering*, 29, 3 (2003), 1-14.

[3] Handel, M. and Herbsleb, J. D. What is Chat Doing in the Workplace? In *Proceedings of the Conference on Computer-Supported Cooperative Work* (New Orleans, LA, 2002).

[4] Herbsleb, J. D., Atkins, D. L., Boyer, D. G., Handel, M. and Finholt, T. A. Introducing Instant Messaging and Chat into the Workplace. In *Proceedings of the ACM Conference on Computer-Human Interaction* (Minneapolis, MN, 2002).

[5] Mockus, A. and Herbsleb, J. D. Expertise Browser: A Quantitative Approach to Identifying Expertise. In *Proceedings of the International Conference on Software Engineering* (Orlando, FL, 2002).

[6] Herbsleb, J. D. and Grinter, R. E. Splitting the Organization and Integrating the Code: Conway's Law Revisited. In *Proceedings of the 21st International Conference on Software Engineering (ICSE 99)* (Los Angeles, CA, May 16-22, 1999). ACM Press,

[7] Herbsleb, J. D. and Grinter, R. E. Architectures, Coordination, and Distance: Conway's Law and Beyond. *IEEE Software*, Sept./Oct. (1999), 63-70.

[8] Grinter, R. E., Herbsleb, J. D. and Perry, D. E. The Geography of Coordination: Dealing with Distance in R&D Work. In *Proceedings of the GROUP '99* (Phoenix, AZ, November 14-17, 1999).

[9] Mockus, A., Fielding, R. T. and Herbsleb, J. D. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11, 3 (Jul 2002), 309-346.

[10] Herbsleb, J. D. and Mockus, A. Formulation and Preliminary Test of an Empirical Theory of Coordination in Software Engineering. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)* (Helsinki, Finland, September 1-5, 2003).

[11] Herbsleb, J. D., Mockus, A. and Roberts, J. A. Collaboration in Software Engineering Projects: A Theory of Coordination. In *Proceedings of the International Conference on Information Systems* (Milwaukee, WI, 2006).

[12] Olson, G. M. and Olson, J. S. Distance Matters. *Human-Computer Interaction*, 15 (2000), 139-178.

[13] Herbsleb, J. D., Mockus, A., Finholt, T. A. and Grinter, R. E. *Distance, Dependencies, and Delay in a Global Collaboration*. City, 2000.

[14] Herbsleb, J. D., Mockus, A., Finholt, T. A. and Grinter, R. E. *An Empirical Study of Global Software Development: Distance and Speed*. IEEE Press, City, 2001.

[15] Guernsey, L. *You Can Surf, but You Can't Hide*. City, 2002.

[16] Raymond, E. S. *The Cathedral and the Bazaar*. O'Reilly, Sebastopol, Cal., 2001.

[17] Lerner, J. and Tirole, J. Some simple economics of open source. *The journal of industrial economics*, 50, 2 (2002), 197-234.

[18] Mockus, A. u., Fielding, R. T. and Herbsleb, J. D. A Case Study of Open Source Software Development: The Apache Server. In *Proceedings of the International Conference on Software Engineering* (Limerick Ireland, June 5-7, 2000).

[19] Stol, K.-J. and Fitzgerald, B. Theory-oriented software engineering. *Science of Computer Programming*, 101 (2015), 79-98.

[20] Gurbani, V. K., Garvert, A. and Herbsleb, J. D. A case study of a corporate open source development model. In *Proceedings of the International Conference on Software engineering* (Shanghai, China, 2006).

[21] Basili, V. R., Selby, R. W. and Hutchens, D. H. Experimentation in software engineering. *IEEE Transactions on Software Engineering*, 12 (1986), 758-773.

[22] Shadish, W. R., Cook, T. D. and Campbell, D. T. *Experimental and quasi-experimental designs for generalized causal inference*. Wadsworth Cengage learning, 2002.

[23] Malone, T. W., Crowston, K. and Herman, G. A. *Organizing Business Knowledge: The MIT Process Handbook*. MIT Press, Cambridge, MA, 2003.

[24] Malone, T. W. and Crowston, K. The interdisciplinary study of coordination. *ACM Computing Surveys*, 26, 1 (1994), 87-119.

[25] Hollan, J., Hutchins, E. and Kirsh, D. Distributed Cognition: Toward a New Foundation for Human-Computer Interaction Research. *ACM Transactions on Computer-Human Interaction*, 7, 2 (June 2000), 174-196.

[26] Hutchins, E. *The Technology of Team Navigation*. Lawrence Erlbaum, City, 1990.

[27] Durfee, E. H. Organisations, Plans, and Schedules: An Interdisciplinary Perspective on Coordinating AI Systems. *Journal of Intelligent Systems*, 3, 2-4 (1993), 157-187.

[28] Hutchins, E. *Cognition in the Wild*. The MIT Press, Cambridge, MA, 1995.

[29] Cataldo, M., Wagstrom, P. A., Herbsleb, J. D. and Carley, K. M. Identification of coordination requirements: implications for the Design of collaboration and awareness tools. In *Proceedings of the Computer supported cooperative work* (Banff, Alberta, Canada, 2006).

[30] Gall, H., Hajek, K. and Jazayeri, M. Detection of Logical Coupling Based on Product Release History. In *Proceedings of the International Conference on Software Maintenance* (Bethesda, Maryland, 1998).

[31] Cataldo, M., Herbsleb, J. D. and Carley, K. M. Socio-Technical Congruence: A Framework for Assessing the Impact of Technical and Work Dependencies on Software Development Productivity. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement* (Kaiserslautern, Germany, 2008).

[32] Cataldo, M., Mockus, A., Roberts, J. and Herbsleb, J. Technical dependencies, work dependencies and their impact of failures. *IEEE Transactions on Software Engineering*, 35, 6 (2009), 864-878.

[33] Cataldo, M. and Herbsleb, J. D. Coordination Breakdowns and Their Impact on Development Productivity and Software Failures. *IEEE Transactions on Software Engineering* 39, 3 (March 2013), 343-360.

[34] Cataldo, M., Herbsleb, J. D. and Carley, K. M. Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity. In *Proceedings of the ESEM '08: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement* (New York, NY, USA, 2008). ACM.

[35] Humble, J. and Farley, D. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[36] Bass, M. Software Engineering Education in the New World: What Needs to Change? In *Proceedings of the 2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)* (2016). IEEE.

[37] Jansen, S., Finkelstein, A. and Brinkkemper, S. A sense of community: A research agenda for software ecosystems. In *Proceedings of the International Conference on Software Engineering-Companion* (2009). IEEE.

[38] Bosch, J. From software product lines to software ecosystems. In *Proceedings of the 13th international software product line conference* (San Francisco, CA, 2009). ACM.

[39] Bernstein, E. S. The transparency paradox a role for privacy in organizational learning and operational control. *Administrative Science Quarterly*, 57, 2 (2012), 181-216.

[40] Dabbish, L., Stuart, C., Tsay, J. and Herbsleb, J. Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository. In *Proceedings of the Computer-Supported Cooperative Work* (Seattle, WA, 2012).

[41] Bogart, C., Kästner, C., Herbsleb, J. and Thung, F. How to Break an API: Cost Negotiation and Community Values in Three Software Ecosystems. In *Proceedings of the Foundations of Software Engineering* (Seattle, WA, 2016).

[42] Tsay, J., Dabbish, L. and Herbsleb, J. Influence of social and technical factors for evaluating contribution in GitHub. In *Proceedings of the Proceedings of the 36th International Conference on Software Engineering* (2014). ACM.

[43] Tsay, J., Dabbish, L. and Herbsleb, J. Let’s talk about it: Evaluating contributions through discussion in GitHub. In *Proceedings of the ACM International Symposium on Foundations of Software Engineering* (2014).

[44] Herbsleb, J. D. Beyond computer science. In *Proceedings of the International Conference on Software Engineering* (St. Louis, MO, 2005). IEEE.