COLLABORATION IN SOFTWARE ENGINEERING PROJECTS: A THEORY OF COORDINATION

ICIS 2006 General Topics

James D. Herbsleb
Institute for Software Research, International
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213 USA
jdh@cs.cmu.edu

Audris Mockus
Avaya Labs Research
233 Mt. Airy Road
Basking Ridge,NJ 07920 USA
audris@avaya.com

Jeffrey A. Roberts
John F. Donahue Graduate School of Business
Duquesne University
Pittsburgh, PA 15282 USA
jeff@roberts.net

Abstract

Coordination of engineering decisions is a central concern of software engineering. We present a theory in which coordination of engineering decisions is modeled as a distributed constraint satisfaction problem (DCSP). We derive six hypotheses, predicting how the distribution of decisions over developers and the density of constraints among decisions will affect development time, probability that a file contains a field defect, and developer productivity. We test these hypotheses using data from a commercial project. We find support for all hypotheses predicting detrimental effects from poor distribution of decisions over developers. The effects of constraint density were mixed, showing that dense constraints slowed development, but did not significantly affect productivity. Dense data dependencies increased the chances that a file contained a field defect, but very surprisingly, dense call dependencies significantly lowered the chances that a file contained a field defect. We discuss the implications of these findings.

Keywords: Constraint satisfaction, coordination, collaboration, geographically distributed software engineering

Introduction

If the core intellectual activity of software engineering is technical decision-making, then the primary goal of software engineering research is to improve our ability to make good technical decisions. We are making steady progress toward this goal as we achieve a better understanding of how technical choices influence software properties, and we create increasingly more powerful tools and techniques that give us new choices.

One of the most difficult aspects of making good technical decisions is a consequence of the fact that in all systems of significant scale, decision-making by multiple individuals and by multiple teams proceeds in parallel. Technical decisions often constrain other technical decisions in ways that are difficult to fully understand. Quite independent of how wise a decision is when considered in isolation, the quality of any technical decision generally also depends on how that decision interacts with other decisions. Given the concurrent nature of decision-making and the conditions of imperfect information that exist in realistic environments, ensuring that decisions correctly take account of other past, present, and future decisions is a very difficult problem. In keeping with the terminology used to describe such problems in many disciplines (Malone & Crowston 1994), we will call this the problem of coordination.

Coordination in Software Engineering

Empirical studies of large-scale projects have shown that coordination is one of the most difficult and pervasive problems (Curtis et al. 1988; Walz et al. 1993). Coordination issues appear at many different levels, including within a team, within a project, across the corporation, and across the business milieu (Curtis et al. 1988). Coordination breakdowns occur when a decision is made that does not properly take other decisions into account. Such breakdowns can cause a variety of problems that differ somewhat depending on the organizational level at which the breakdown occurs. For example, issues at the team level concern mostly the system design and implementation, while issues at the corporate level typically concern product attributes, progress, schedules, and resources (Curtis et al. 1988). Requirements gathering and analysis seem particularly prone to coordination and communication failures (cf. Walz et al. 1993; Damian & Zowghi 2003).

Coordination is more difficult when projects are "technically uncertain," i.e., unstable, and confronting non-routine technical problems (Kraut & Streeter 1995). The problems are also made worse when teams are geographically distributed (Herbsleb & Grinter 1999). In a finding that suggests that team-level coordination has a powerful effect on project cycle time, it has been shown that the number of people involved in a work item is strongly related to how long the work takes to accomplish, controlling for such factors as the size and complexity of the work item (Herbsleb & Mockus 2003).

In general, organizational theorists talk about two distinct approaches to coordinating work: coordination by "programming" (March & Simon 1958) and coordination through mutual adjustment (Thompson 1967). In coordination by programming, explicit agreements of various sorts are established to guide the work. In software engineering, things like interface specifications, software process, architectures, and project plans all fall into this category (Osterweil 1986). Coordination by small "agile" teams (Beck and Andres 2004) who rely on frequent communication to ensure their decisions are consistent provides an example of coordination through mutual adjustment.

Software engineering research has taken diverse approaches to solving coordination problems, including automatically enforcing API policies (Ball & Rajamani 2002), documenting and enforcing programmers' intentions (Greenhouse & Scherlis 2002), incorporating collaboration and awareness capabilities into the development environment (Cheng et al. 2003; Sarma et al. 2003), and physically co-locating development teams to improve communication (Teasley et al. 2002).

In addition to approaches that single out and specifically address coordination concerns, many of the foundational ideas of software engineering also address coordination in addition to their primary purpose of understanding how to design software with desired properties. For example, the notion of modularity (Parnas 1972) has important non-coordination benefits, such as permitting design of software that more easily accommodates anticipated change, and that is easier to understand. Yet modularity also addresses coordination problems. In fact, in the paper in which Parnas (1972) introduced the concept of information hiding for modular design, he chose to emphasize the coordination aspect, noting that by "module" he meant a "work item," not a subprogram. Developers concurrently

making decisions about the internal functioning in two different modules are unlikely to constrain each other in unknown ways, so long as the module interfaces are unchanged.

Another example of a key area of software engineering that also addresses coordination is the notion of architectural style (Shaw & Garlan 1996). Adopting an architectural style not only helps to ensure that the designed system has desirable technical properties; it also helps each developer make decisions consistent with those of other developers. Teams of developers who all know they are working within a pipe-and-filter style, for example, have a much better chance of making decisions that are mutually compatible than if they are trying to make the same technical decisions not having agreed to any particular style.

In realistic situations, many forms of coordination are employed in concert. It is not unusual for a project to employ information hiding, well-defined component interfaces, use of standards, scheduled technical review meetings, shared artifacts, software processes, project plans, ad hoc communication, work breakdown structure, management directives, coding standards, and more. While we understand intuitively that all of these activities contribute to coordination, the mix varies substantially across different parts of a project, and from one project to the next. For example, some interfaces may be very well-defined and stable while others are not as mature or well understood; some groups may be much more familiar with the particular artifact types used in a project than are other groups; some plans are based on data from previous similar projects while others, for unprecedented systems, are full of guesswork; communication is frequent and effective for co-located teams, but infrequent and very difficult among geographically distributed teams.

Projects will also vary substantially in their need for coordination of technical work across the project. Systems with very stringent performance or reliability requirements, for example, will tend to have many dependencies because many decisions can affect these global properties.

Finally, organizations are strongly influenced by past projects. Patterns of communication tend to become fixed, and developers accustomed to working on various parts of the product develop powerful habits dictating what sorts of information they attend to and what they ignore (Henderson & Clark 1990; Sosa et al. 2004). Significant changes in the architecture for a new project can result in a serious mismatch between these organizational capabilities and the coordination needs of a project. In fact, such mismatches have been observed not only to make projects fail, but to put companies out of business (Henderson & Clark 1990).

In the face of this very complex situation, what we need is a unifying view of coordination. We need to understand how new projects give rise to coordination needs and how we can analyze a project in order to determine what sorts of coordination mechanisms need to be in place for the project to progress efficiently. We need to understand the relationships among the various means of coordination a project can employ. If some interfaces are sketchy and unstable, can we compensate with a more rigorous software process, or more frequent sharing of artifacts, or increased use of communication tools? All of the various ways of coordinating projects have a cost associated with them, and we have no good way at this point of understanding for a given project and a given organization what sorts of coordination costs we should be willing to pay, and what particular means of coordination are likely to prove effective.

In the next two sections, we review two approaches to a unifying view of coordination that have been applied in related domains.

Interdisciplinary Theory of Coordination

Coordination problems in many fields have similar properties (Malone & Crowston 1994). For example, the problems of humans competing for floor space and programs competing for memory have similar characteristics, since both are instances of a resource conflict. Independent of discipline, one could theoretically catalog all types of dependency patterns, and identify mechanisms (e.g., scheduling) that can resolve each type of conflict (Crowston 2003; Dellarocas 1996).

This view of coordination gives us something to which we can aspire in software engineering. Indeed, there has been work capturing “organizational patterns” for software teams (Coplien and Harrison 2004). While we are beginning to get a systematic view of the types of dependencies that can exist between units of software (see Clements et al. (2002) and Dellarocas (1996) for examples), we do not at this point have a formulation of the possible types of dependencies that can exist among technical decisions in software engineering. An important goal of our research is to move toward such systematization, but it is an aspiration, not a starting point.

Distributed Cognition

Cognition in psychology has traditionally been thought of as an activity that occurs purely "in the head." This view has changed dramatically in recent years (cf. Hutchins 1995). Many complex tasks are best understood as cognitive or problem-solving processes that are distributed over individuals, artifacts, and time (Hollan et al. 2000; Hutchins 1990), and partially embedded in the habits, practices, and routines of the people who carry out the problem-solving activities (Brown & Duguid 2001). The view of problem solving achieved by communication and sharing of artifacts, and embedded in a social and organizational context, has much in common with our theory.

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 (Hutchins 1995), 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 (1995) compares the radically different ways that navigation problems can be solved by naval officers using western 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 for addressing the problem.

A related line of work, often called distributed artificial intelligence (DAI), has specified frameworks for coordinating activities among distributed agents for tasks such as regulating traffic signals or factory floor robots (cf. Durfee 1993). Many problems in DAI can be formalized as distributed constraint satisfaction problems (Yokoo 2001), and we borrow this insight for the formal core of our theory. As is the case with ship navigation, coordination among agents can be accomplished in many ways, but each coordination problem has an irreducible grounding in the tasks to be accomplished and their interdependencies.

Formulating a Theory of Coordination

In this paper, we take a step toward creating a unifying view of coordination in software engineering. The need for coordination arises from the core technical activity of making design decisions where the alternative chosen for any given decision potentially influences how one evaluates some number of other decisions. The full extent of these influences is often quite difficult to determine, hence decision-making is generally performed with some degree of uncertainty and imperfect information.

We theorize that the irreducible interdependence among tasks in software engineering can usefully be thought of as a distributed constraint satisfaction problem (DCSP), hence, coordination can be thought of as execution of an "algorithm" that solves a DCSP. In a DCSP, decisions are embedded in a network of constraints, and are potentially owned by many agents. Finding a solution then generally requires cooperation among the agents. Given the boundedly rational nature of such decision-making, solutions will generally be satisficing rather than optimal (Simon 1981). As with many coordination problems, such as ship navigation described above, coordination in software engineering can be carried out in a great many ways, involving a variety of design methods, social processes, communication regimens, communication tools, and so forth. Yet, like navigation, the problem has an irreducible core – if incompatible decisions are made, then the software will not work properly.

We will now be a bit more precise about defining a DCSP, using Yokoo's (2001) formulation, applied to the software engineering domain. 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.

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 $D_{k1} \times D_{k2} \times \ldots \times D_{kj}$. Successfully completing a project is equivalent to finding an assignment for all variables that satisfies all constraints.

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. .

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.

In particular, in a DCSP, it is critical for certain agents to be able to communicate efficiently. Essentially, three types of communication are important. First, agents that own mutually-constraining decisions need to coordinate the decisions they make. If they know about the constraint, and know what other agents own decisions affected by the constraint, the coordination problem is relatively simple, since they know whom they must communicate with about what. Second, agents may be affected by constraints they do not know about, but must discover as development proceeds. This presents a very difficult problem, since the agents need to coordinate, but it is not clear with whom or about what. Third, agents may adjust their problem-solving strategies based on their observations, e.g., to avoid excessive backtracking. Communication is often required to agree on changes in problem-solving strategies.

We do not expect that decision-making and communication in any actual software engineering project will follow any easily-described DCSP algorithm. Nevertheless, if software engineering is appropriately modeled as a DCSP, then if a project is successful, the developers have in fact executed some algorithm capable of solving a DCSP. The algorithm is emergent, and is shaped by a variety of factors (see Figure 1). We would expect, in addition to communication, mentioned above, other factors would influence how the DCSP is solve, including

- geographic distribution of the developers,
- the structure of the organization,
- the social networks among developers,
- organizational culture that impacts communication and how constraints are discovered and handled,
- business influences that dictate tradeoffs, e.g., between speed and quality, and
- the tool infrastructure that influences ease of discovering constraint violations and communicating effectively so as to avoid or correct them.

There are no doubt other factors as well.

The advantage of viewing coordination as a DCSP is that it provides a unifying framework for phenomena that seem quite different. A conversation over lunch, a formal annotation of programmer intent, and review of a project artifact are all mechanisms by means of which a particular set of technical decisions may be resolved in a way that satisfies a particular set of constraints. All such phenomena, in this view, have their effects by influencing the emergent DCSP algorithm.

Focusing for the moment on the basic DCSP structure of decisions and constraints, this network is generated and shaped by the patterns of dependencies in the software (Baldwin & Clark 2000; Dellarocas 1996). The landscape of constraints (hence the number and density of possible solutions) can have an enormous impact on solution time (Cheeseman 1991). So the patterns of interdependencies among agents and the density of dependencies in the constraint landscape should be particularly important factors if coordination is appropriately modeled as a DCSP. These factors should substantially influence the properties of the global algorithm implemented by the overall project. In the research reported here, our goal is to test several hypotheses to see if coordination in software engineering behaves in very basic ways as one would expect if DCSP describes its fundamental properties.

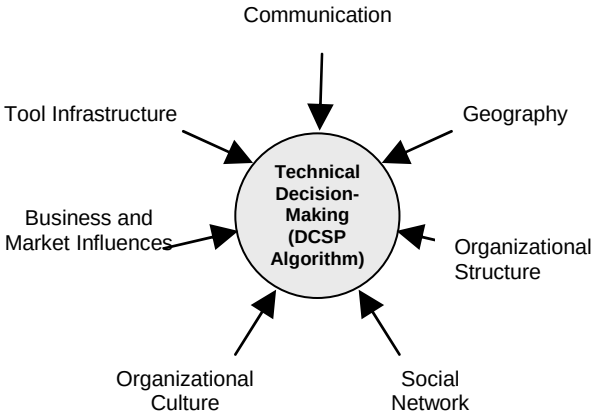

Figure 1. Some factors that help to shape the emergent DCSP algorithm.

In section 2, we present our research questions and hypotheses. The methods and results are presented in section 3. We discuss the implications of the work and open research questions in section 4.

Research Questions

As mentioned above, a reasonable first step in testing the utility of a theory of coordination based on DCSP is to see if the coordination processes behave, in very basic ways, as one would expect if executing a DCSP algorithm. In essence, we want to investigate the central circle in Figure 1, relatively free of the influences represented by the arrows. For this reason, we examine work items within a single software engineering project, thereby holding the influences shown in Figure 1 relatively constant.

Constraint violations occur when decisions are made that fail to satisfy one or more constraints. When constraint violation occurs, there are two possible outcomes, assuming that the project eventually completes and delivers a product. The constraint may remain unsatisfied at delivery, in which case the constraint violation has (by definition) resulted in a defect in the product. Alternatively, the constraint violation may be detected and corrected. Depending on the constraint structure of the problem and the constraint satisfaction algorithm being applied, this correction may induce backtracking of other decisions and additional problem solving activity to find a new solution. In any case, correction of a constraint violation requires an expenditure of resources and consumption of calendar time in order to correct the breakdown.

This view of constraint violations can be given a quantitative interpretation (see Figure 2). Other things being equal, one would expect that for any unit of work, the more constraint violations that occur, the more defects the result will contain, the longer the development time, and the more effort that will be required.

In addition to these effects of constraint violation, Figure 2 also shows two factors that our theory suggests should play a causal role, i.e., increase the likelihood of constraint violations. Recall that in our theory each decision is embedded in a network of decisions, where edges represent predicates that constrain the values each decision may take. High constraint density should, other things being equal, increase the likelihood of constraint violations, since there are simply more constraints that could potentially be violated.

Second, the view of coordination as DCSP provides some predictions about how decisions are distributed among developers in the development process. It might be expected, other things being equal, that assigning a work item to two people rather than one would speed up the rate at which work is done, since two people can work concurrently. On the other hand, for work representing sets of decisions that are embedded in relatively dense constraint networks, assigning the work to two people rather than one requires them to coordinate their work in order to ensure that all constraints are satisfied. We hypothesize, therefore, that under conditions of high constraint density, distributing decisions over more people, will cause more constraint violations, potentially negating any benefits of concurrent

work. Having more people work sequentially on work items should similarly impose the need for more coordination and generate more constraint violations, in this case without the potential benefits of concurrency.

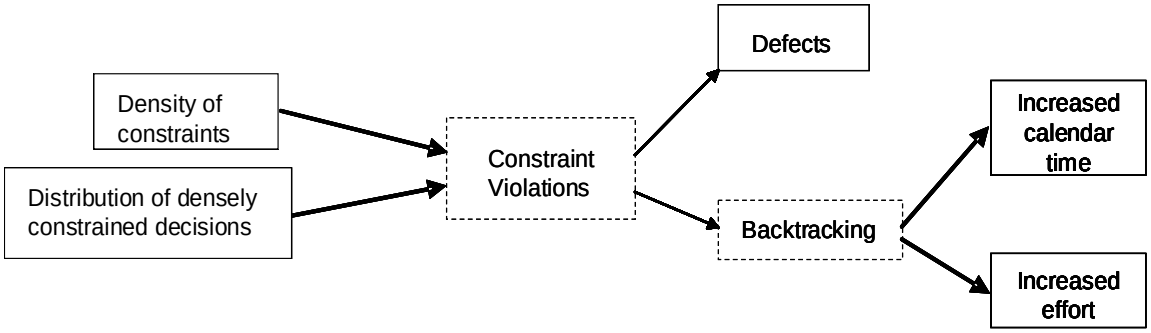

Figure 2. Quantitative model of some causes and effects of constraint violations.

Figure 2 summarizes these hypothesized relationships. We do not directly measure the number of constraint violations or the amount of backtracking, as these are not observable in data sets to which we have access. These represent the mechanisms that we theorize are responsible for observed quantitative relations between the causes of constraint violations (on the left of Figure 2) and their effects (on the right).

The hypotheses of our study follow from the relationships shown in Figure 2. Hypotheses 1-3 concern the density of constraints:

H1: Higher constraint density increases the time required to make a modification to the code.

H2: Work performed in the context of high constraint density is more likely to produce errors than work in the context of lower constraint density.

H3: Work performed in the context of high constraint density requires more effort than work in the context of lower constraint density.

Hypotheses 4-6 concern the distribution of densely constrained decisions amongst people:

H4: Distributing densely constrained decisions over more people increases the time required to make a modification to the code.

H5: Distributing densely constrained decisions over more people increases the likelihood of errors.

H6: Distributing densely constrained decisions over more people requires more effort than when the decisions are distributed over fewer people.

Empirical Methods and Results

Project and Site

We study an embedded software product for a communications device of a leading enterprise telecommunications company during the period from April, 2001 to May, 2005. The product consists of a user interface running on a popular embedded operating system – VxWorks. The product is written in the C and C++ languages and the delivered version consists of more than one million lines of code. Product development during the period under study was carried out by 40 software engineers. The greater part of the product development takes place in the eastern United States (39 engineers) with the balance taking place in Australia (1 engineer). The majority of the software developers assigned to the project have extensive software development experience. The project began in early 2000 and has since undergone significant changes in hardware requiring extensive software development during the period under study. At the time of the writing, the product is approaching its fourth major release.

Modification Requests

Modification request (MR) and version control systems (VCS) are used by virtually all software projects to coordinate the work of the project participants and to allow parallel work on several releases and patches. This dataset is a typical example of the data that is usually available from VSC and MR systems.

A slightly simplified description of an MR process follows. The developers are assigned (or, more often, assign themselves) a new feature or a defect on which to work. In the case of defects, they investigate the problem, make necessary changes and submit an MR for integration. In the case of new features, additional tasks such as low level design and design review are performed prior to coding. After coding is complete the MR is submitted for integration by the developer. If the MR prevents system build, it may be rejected by the integrator and then the developer makes needed modifications and resubmits it. The code inspection may be done afterward and any issues are resolved with additional MRs. The MRs may originate from customers, testers, or developers themselves. Often developers will find an issue to work on in the regular course of their activities. In some cases developers reassign MRs to other developers if they can not resolve the problem on their own. More than half MRs do not lead to changes. They include such things as duplicate reports, and problems that are not reproducible or that are not high enough on the priority list.

The product under study used the Sablime configuration management system which uses the SCCS version control system to manage code changes. We extracted workflow relationships by processing Sablime MR history files for changes made during the period under study. An MR history file contains a record of all transactions on an MR, including information about by whom, when, and what fields were changed. We extract MR creation, all MR assignments, and all MR submission/rejections.

Variables, Data Analysis, and Results

Modeling constraint density

The density of constraints operating within a software development project figure prominently in several of our hypotheses and subsequent empirical models. Our operationalization of constraint density relies on determining the extent to which software features interact and the nature of those interactions. Our operationalization is premised on the idea that the functionality of a software system is determined by the software functions and data comprising that system. Software systems are commonly decomposed using various strategies into smaller functional units designed carry out specific tasks (Pressman 2004). Depending on the programming language, such units are variously referred to as functions, subroutines, methods, etc. Additionally, related units are often grouped into modules (i.e., source files) allowing for easy manipulation by version control and configuration management systems. The propensity for one unit to rely on, use or *call* another represents a dependency between the units that serves to constrain the manner in which changes can be carried out. For example, changes to the output of a function must consider how the change may affect other functions that call this function and thus may *depend* on the original output format. Likewise, the data manipulated by the system can be organized in various ways that result in more (or less) dependency between the data elements. By measuring the call and data level dependencies that exist within and between software modules we can approximate the density of the constraints facing developers working within the context of those modules. In the present case, we refer to the context that exists within the set of modules comprising an MR.

In order to extract the call and data dependencies within the source files comprising an MR we adapted the C-REX extractor, an existing tool suite for static source code analysis (See Hassan & Holt 2004 for a detailed description of C-REX.) Our adaptation of the extractor generates information about the call and data dependencies that exist within the source code. For example, consider a modification request comprised of *file1.c* and *file2.c*. The extractor might report that symbols (or tokens) representing a global string `format` and functions `init()` and `usage()` are defined in *file1.c* and that the function `print()` is defined in *file2.c*. It might also determine and report that the MR contains 1 call and 1 data dependency. For example that function `init()` calls function `usage()` and function `print()` uses the datum `format`.

To compute the call and data dependencies within the source files comprising an MR, the basic procedure outlined in the example above is repeated for all modification requests in our sample. First, measures of the call and data dependencies that exist *within* each MR are retained. Next, symbols unresolved by MR-level information (i.e., *internal* to the MR) are reconciled against all symbols known to exist within the project. Symbols resolved at the

project-level are consider to be *external* to the MR but still within the scope of the project. Unresolved symbols are deemed to be outside the scope of the project and are discarded.

Modeling development time

We tested our six hypotheses by constructing and estimating three regression models. Hypotheses *H1* and *H4* make predictions about the time required to develop or make changes the software. We tested both of these hypotheses in a single regression model. The unit of analysis was the modification request. A log transformation was applied to all continuous variables, which, in all cases, produced a good approximation to a normal distribution. The dependent variable, *development time*, was computed as the seconds between the first change and the last change within a modification request.

Table 1 – Descriptive statistics for model variables

Panel a	Mean	(1)	(2)	(3)	(4)	(5)	(6)
development time (1)	10.915	1.0000					
work distribution (2)	1.282	0.5957	1.0000				
new functionality (3)	0.158	-0.0548	-0.1823	1.0000			
number of changes (4)	1.187	0.1335	-0.0422	0.1157	1.0000		
number of lines (5)	3.786	0.1399	-0.0435	0.1353	0.5705	1.0000	
constraint density (6)	5.599	0.1677	0.0837	-0.0205	0.4255	0.2810	1.0000
N = 1448							
Panel b							
bugginess (1)	0.237	1.0000					
number of people (2)	1.495	0.5402	1.0000				
lines of code (3)	6.340	0.2886	0.3716	1.0000			
data dependencies (4)	3.024	0.2642	0.3364	0.6294	1.0000		
call dependencies (5)	3.586	0.2229	0.4410	0.5912	0.8210	1.0000	
N = 557							
Panel c							
productivity (1)	9.951	1.0000					
indegree (2)	2.570	0.0771	1.0000				
outdegree (3)	5.578	0.5615	0.6078	1.0000			
lineschanged (4)	9.813	0.7868	0.4452	0.8292	1.0000		
N = 40							
Panel d							
productivity (1)	9.627	1.0000					
indegree (2)	2.742	0.6371	1.0000				
constraint density (3)	5.478	0.1497	0.3188	1.0000			
lineschanged (4)	9.838	0.8785	0.7571	0.1594	1.0000		
N = 33							

Notes: Dependent variables show in italics. Panel a – model variables used to test *H1* and *H4*. Panel b – model variables used to test *H2*.and *H5*. Panels c and d – model variables used to test *H3* and *H6*.

Underlying *H4* is the assumption that changes made to the software as part of a single MR are highly interdependent. That is, we contend that MRs reflect a software environment that is more densely constrained and thus more interdependent than one would expect by chance. To test the assumption of the interdependence of MRs, we conducted a monte carlo analysis consisting of 100 randomly selected MR sample frames each of which recreated the empirical distribution of the MRs used in our sample (N=1448). We operationalize constraint density as function of internal and external MR call and data dependencies outlined above. Recall that internal dependencies are those that can be resolved within the scope of files comprising the MR and the external dependencies are resolved outside of the MR but within the system under consideration. We tested the null hypothesis of equal mean constraint density between MRs and the random samples against the alternative hypothesis that mean MR constraint is greater than the samples. A *t*-test of the means easily rejects the null hypothesis in favor of the alternative (t = 27.57; $p < .001$) thus supporting our contention.

We computed values for several predictor variables of interest:

- *work distribution* is the number of developers who checked in code as part of the work for the modification request.
- *constraint density*, as note previously, is operationalized as a function of internal and external MR call and data dependencies. Preliminary analysis revealed a high degree of correlation among the four constraint density measures; accordingly, we summed the measures to form a single constraint density measure (*Cronbach Alpha* = .96.)

Several control variables were also included in the model:

- *number of changes* which is the number of delta, or changes to individual files, in the MR,
- *number of lines*, which is the number of lines of code added and modified as part of the MR work.
- *new functionality* was a dichotomous indicator variable indicating that the change was a new feature.

Refer to Table 1 – panel (a) for descriptive statistics and correlations of the variables comprising this model.

Table 2 shows the results of this logistic regression model. *H1* stated that constraint density would increase the time required to make changes to the software. *H1* is supported as the constraint density coefficient is positive and significant (b=.123, p<.05). *H4* stated that distributing densely constrained decisions over more people would increase the time required to make the change. *H4* is supported as the work distribution coefficient is positive and significant (b=4.280, p<.001). Thus, the results provide support for both *H1* and *H4* showing that both work distribution and constraint density are associated with significantly longer development times.

Table 2 - Results of regression model of development time

Intercept	7.675 ***
work distribution	4.280 ***
constraint density	0.123*
number of changes	0.522***
number of lines	0.167 ***
new functionality	0.305
number of files	0.387**
R^2:	0.398

Notes: N=1448. *p<.05; ** p<.01; ***p<.001

Modeling file "bugginess"

In order to test *H2* and *H5*, we constructed a logistic regression model to predict defects. For each file, we determined the number of times that it had been changed in response to a defect reported in the field. The distribution of this variable was quite skewed; therefore, we constructed a dichotomous variable (*bugginess)* to represent whether the file had ever been changed in response to a field defect – 1 for yes 0 otherwise.

Our predictor variables were

- *Data dependencies*. The total number of internal and external data dependencies occurring within the file.[1]
- *Call dependencies*. The total number of internal and external call dependencies occurring within the file.
- *Number of people*. The number of people who have made changes to the file.

We also included a control variable, since one would expect that the larger a file, the greater the chance that it contains a bug:

- *Lines of code.* Total lines of code in the file, measured at the end of the considered period.

Refer to Table 1 – panel (b) for descriptive statistics and correlations of the variables comprising this model.

Table 3 – Results of logistic regression model of file "bugginess."

Intercept	-6.716***
data dependencies	0.638***
call dependencies	-0.796***
number of people	2.720***
lines of code	0.282*

Notes: N=557. *p<.05; ** p<.01; ***p<.001

Table 3 shows the results of the logistic regression model. The results partially support and partially contradict *H2*. We expected both data and calls dependencies to increase the chances that a file would be found to contain a field defect. Our prediction was confirmed with respect to data dependencies (b=.638, p<.001), which were a highly significant predictor of "bugginess." Indeed, the coefficient reflects that, all else equal, the odds of a defect increase by a factor of 1.89 ($e^{0.6375}$) for each additional data dependency that occurs in the file. We were quite surprised, on the other hand, to see that calls dependencies not only did not increase the chance that a file would contain a field defect, but significantly reduced it (b=.795, p<.001). Here, the coefficient reflects that, all else equal, the odds of a defect decrease by a factor of .451 ($e^{-0.7959}$) for each additional call dependency that occurs in the file. In other words, the odds for a file with one more call dependency to contain a field defect are only about .451 times as high as a file without this additional call dependency. In the discussion section we provide several somewhat speculative interpretations for this unexpected result as well as some research directions these results suggest.

We found unambiguous support for *H5*. The more people who changed a file, the more likely the file was to contain field defects (b=2.7203, p<.001). Once more, the coefficient reflects that, all else equal, the odds of a defect increase by a factor of 15.19 ($e^{2.7203}$) for each additional developer that makes changes in the file.

[1] In contrast to MR-level dependencies, file-level data and call dependencies were not highly correlated and thus no single measure of constraint density could be formed.

Modeling developer productivity

In order to test *H3* and *H6*, we constructed two regression models to predict developer *productivity*, computed by dividing the total number of lines of code added by a developer and dividing by the amount of time she spent on the project. The amount of time was estimated as the number of days from the first contribution to the last contribution.

In our first regression model, we computed values for one predictor variable for each developer:

- *Indegree* is the number of people whose lines the developer has modified.

We expected *Indegree* to significantly reduce a developer's productivity. Two modifications of the same line of code are highly mutually constraining, and according to *H6*, this distribution of highly constraining decisions should take more effort, hence reduce productivity.

We also included two control variables in the model.

- *Outdegree* is the number of people who have modified the developer's lines.

We included it in order to control for the general tendency to share files in the developer's environment.

In order to have interpretable results, we also need to control for the absolute number of lines written by others (as opposed to the number of people who wrote those lines) that a person changes and the number of lines a person writes that others change, since we want to focus on the number of people over whom the work is distributed. We control for these factors with the variable that represents the part of a developer's code where the developer interfaces with other developers' decisions expressed in code

- *Lineschanged* which is the sum of the lines of other people that a developer changed and the lines of the developer that other people changed.

We tested *H3* by regressing *Indegree*, *Outdegree*, and *Lineschanged* on *productivity*, using ordinary least squares regression. All variables in the model were subjected to a natural log transform in order to generate distributions that more closely approximate the normal distribution. Refer to panel (c) of Table 1 for descriptive statistics and correlations of the variables comprising this model.

Table 4 - Results of regression model of developer productivity

Intercept	3.19**
indegree	-0.800**
outdegree	0.100
lineschanged	0.920**
Adjusted R^2	.69

Notes: N=40. *p<.05; ** p<.01; ***p<.001

The results, shown in Table 4, indicate that Indegree had a significant negative effect on developer productivity (b=-0.80, p<.01).

This model did not include a measure of constraint density. We excluded it because, as mentioned above, we were not able to compute a measure of constraint density for files other than C and C++. There were several developers who performed significant work in other languages, specifically, java and assembler, who had to be excluded from the constraint density analysis, permitting only a weaker statistical test. The model just presented allowed us to test the significance of the predictor *Indegree* on the full sample of developers.

In order to test the effects of constraint density, we computed a constraint density measure as follows.

- *Constraint density*. Similar to our handling of constrain density at the MR-level, constraint density was operationalized here as an average of internal and external call and data dependencies for the MRs on which the developer worked weighted by the number of deltas for each MR. Again, analysis revealed a high correlation among the four density measures; accordingly, we summed the measures to form a single measure of average constraint density (*Cronbach Alpha* = .97.)

We examined both data and calls dependencies, and found them to be very highly correlated (greater than .90), making it appropriate to aggregate them into a single measure. We simply summed them to create our developer-level measure of *Constraint density*.

We added this constraint density measure to the previous model, and restricted the data to developers who exclusively wrote C or C++ code. We also excluded *Outdegree*, as the previous analysis had shown it had no effect. Refer to panel (d) of Table 1 for descriptive statistics and correlations of the variables comprising this new model. The results are shown in Table 5. Neither of the effects of interest was significant in this model.

Table 5 - Results of an augmented regression model of developer productivity

Intercept	2.449**
indegree	-0.228
constraint density	0.025
lineschanged	0.779***
Adjusted R^2	.75

Notes: N=33. *p<.05; ** p<.01; ***p<.001

Discussion

Implications for a DCSP-based theory of coordination

In this paper, we have proposed a theory of coordination in software development. The key idea of this theory is that coordination in software engineering can usefully be described as a distributed constraint satisfaction problem. We also generated six hypotheses from this theory, and used data from a large commercial project to test them. The results were mostly supportive, but with a few intriguing anomalies which we address below.

Our predictions concerning the distributions of highly mutually constraining decisions (*H4-H6*) were uniformly supported. We measured distribution of highly mutually constraining decisions in several different ways. Viz., making changes (more or less) concurrently in the same MR; modifying the same line of code, often at very different times; and modifying the same file, over the life of the file. In each case, distributing these sets of decisions over more people had significant harmful effects, including longer development time, increasing likelihood that code is "buggy," and reduced developer productivity.

These results point out the importance of how work is distributed over developers and teams in a software project. In other types of product design, it is clear that the product architecture has an enormous impact on patterns of communication (Sosa et al., 2004), and that changes in product architecture can seriously challenge the development organization's ability to effectively coordinate the design work (Henderson and Clark, 1990). Divisions of the work suggested by product architecture are often used to split software projects across sites (Grinter et al., 1999), and techniques have been developed to compute an optimal partitioning of software using change history data (Mockus and Weiss, 2001). Melvin Conway suggested in 1968 (Conway, 1968) that the structure of products ends up resembling the structure of the organizations that design them. Our work has shown that splitting up work that is highly mutually constraining affects effort, time, and quality.

The theory provides a unifying explanation for all these results, suggesting that there is substantial delay or loss of information in the communication of constraints and/or decisions. This interpretation is consistent with recent work by other researches who found that development time was shorter when people whose work was interdependent actually communicated around the time the work was performed (Cataldo et al., 2006).

Our hypotheses concerning the effects of constraint density, however, had more mixed results. As we expected, our measure of constraint density, which is based on familiar ideas of call and data dependencies, was a significant predictor of development time and supported *H1*.

H3 which stated that work performed in the context of high constraint density requires more effort was neither supported nor contradicted. It may be that we did not have sufficient statistical power to observe an effect of

constraint density on developer productivity, or it may simply be that there is no such effect. We plan to replicate this analysis on a larger data set.

The results for *H2* were the most puzzling and intriguing. For the other hypotheses concerning constraint density, *H1* and *H3*, we found that when computing data dependencies and call dependencies over MRs, (the number of dependencies of all files touched in the MR) and people (the weighted average of dependencies of all MRs the person participated in), these two types of dependencies were very highly correlated (.95 and above). But for data and call dependencies computed over files, these measures are not strongly correlated. Moreover, and most puzzling, they have opposite effects on the likelihood that a file contains a field defect.

There are several possible reasons for this unexpected result. It may be that call dependencies are simply easier to follow. One can see the function calls, at least from the caller's side, quite easily, and development environments typically have tools that simplify location of the functions and methods. Data dependencies may simply be much more subtle and harder to identify and locate.

While this may explain why data dependencies are "worse" than calls dependencies (i.e., increase the likelihood of defects), it still does not explain why higher numbers of call dependencies decrease the likelihood of defects. Recall that our measures are taken at the file level, so calls internal to the file are not counted. Low numbers of calls dependencies may be associated with large numbers of (uncounted) internal calls, which may be reflect poorly structured code. We are not terribly convinced by our speculations about this issue – good answers await further research.

What we do take from this puzzling result is that research in software engineering needs to work toward a deeper understanding of how design structure is related to task structure (Baldwin and Clark, 2000). Even if two modules of software interact extensively, if the nature of the interaction is well understood and does not change as the design work progresses, there may be little need for people implementing these two modules to coordinate their work. This suggests that uncertainty (Kraut and Streeter, 1995) about precisely how two product modules will need to interact may moderate the relationship between product modularity and work modularity. In other words, the complexity of interdependencies among parts of the product may produce a need for intense coordination of the work on those parts only when there is uncertainty about the precise nature of the intended interaction in the product. A better understanding of how to predict (invisible) work dependencies from (visible) dependencies in the architecture, design, and code of the product would allow us do a better job of distributing the work among people which, as we have seen, significantly influences development time, productivity, and quality.

We regard these results as promising. They establish that in some key respects, particularly regarding the various effects of distributing mutually-constraining decisions and the effects of constraint density on development time, coordination in software engineering appears to act as one would expect if DCSP is an appropriate model.

As with any theory, of course, it provides a simplified view of the phenomenon. For example, we assume that constraints are binary, i.e., that decisions are either compatible or they are not. The truth is often more complex, involving tradeoffs on multiple dimensions. Further research will show if we need to introduce additional complexity into theory, e.g., by viewing coordination as optimization rather than simpler constraint satisfaction.

The empirical work also has inevitable limitations, based as it is on a single project. In particular, we need to replicate on additional data sets to see if the relations we found here are in fact general.

Prospects for a unified view of coordination

We have investigated some fundamental DCSP-relevant properties of coordination in software engineering, and clearly more work is required before we can regard the fit of this theory to the phenomenon as firmly established. Nevertheless, we think it is useful even at this early point to outline how we believe a DCSP-based theory can provide a unified view of coordination.

The focus of the theory is exclusively on coordinating the technical work. The fundamental agent behaviors – making decisions, communicating decisions, and communicating constraints – identify the critical points at which other factors such as social, cultural, organizational, and technological factors influence coordination.

We will illustrate with an example from our research group, examining the roles of computer-mediated communication, geography, and team structure on coordination. Cataldo et al (2006), while not providing a theoretical interpretation of their results, found that when coordination behaviors matched coordination

requirements, development time was reduced. A description of their methodology will make the relationship to the current theory clear.

As in this paper, Cataldo et al (2006) conceptualized a task as making a change to files containing code. When files are frequently changed together in the same modification request, this is an indication that decisions about changes to those files are highly mutually constraining. If we generate a symmetric file by file matrix, where each $cell_{ij}$ represents the number of times files *i* and *j* were changed in the same modification request, we have a task dependency matrix, indicating the extent to which the task of changing each file constrains the task of changing each other file. Without reproducing all the details here, this task dependency matrix can be combined with task assignment information to produce a set of coordination requirements, i.e., a computation of the extent to which the tasks each person performs constrain the tasks performed by others. Our investigations have shown that development work is faster when those performing mutually constraining work 1) are on the same team, 2) are located at the same site, 3) communicate using an asynchronous text tool, or 4) communicate in a chat room.

This example illustrates how one can investigate the effects of factors such as communication, team structure, and geography on coordination, within the context of our theory. At the heart of the approach is the notion that software engineering work consists of making decisions, these decisions are embedded in a network of constraints, and a wide variety of factors can potentially influence the effectiveness and efficiency of the distributed constraint satisfaction algorithm the organization is executing.

Acknowledgements

The authors wish to acknowledge support from NSF grants IIS-0414698 and IIS-0534656, as well as the Software Industry Center at Carnegie Mellon University and its sponsors, especially the Alfred P. Sloan Foundation.

References

Baldwin, C.Y. and Clark, K.B. Design Rules: The Power of Modularity, Cambridge, MA: The MIT Press, 2000.

Ball, T. and Rajamani, S.K. "The SLAM Project: Debugging System Software via Static Analysis," in ACM Symposium on Principles of Programming Languages, Portland, OR, 2002

Beck, K. and C. Andres (2004). Extreme Programming Explained: Embrace Change, Addison-Wesley.

Brown, J.S. and Duguid, P. "Knowledge and organization: A social-practice perspective," Organization Science (12:2), 2001, pp. 198-213.

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," To appear in ACM Conference on Computer-Supported Collaborative Work, Banff, Alberta, Canada, 2006.

Cheeseman, P., Kanefsky, B., and Taylor, W. "Where the really hard problems are," in International Joint Conference on Artificial Intelligence, 1991.

Cheng, L.T., de Souza, C.R.B., Hupfer, S., Patterson, J., Ross, S. "Building Collaboration into IDEs," Queue (1:9), 2003, pp. 40-50.

Clements, P. Documenting Software Architectures: Views and Beyond, Addison-Wesley, 2002.

Conway, M.E. "How Do Committees Invent?", Datamation (14:4), 1968, pp. 28-31.

Coplien, J. O. and N. B. Harrison (2004). Organizational Patterns of Agile Software Development. Upper Saddle River, NJ, Prentice-Hall.

Crowston, K. "A taxonomy of organizational dependencies and coordination mechanisms," in Tools for Organizing Business Knowledge: The MIT Process Handbook, T.W. Malone, K. Crowston, and G. Herman (Eds), MIT Press, Cambridge, MA, 2003.

Curtis, B., Krasner, H., and Iscoe, N. "A Field Study of the Software Design Process for Large Systems," Communications of the ACM (31:11), 1988, pp. 1268-1287.

Damian, D.E. and Zowghi, D. "Requirements Engineering challenges in multi-site software development organizations," Requirements Engineering Journal (8), 2003, pp. 149-160.

Dellarocas, C. "A Coordination Perspective on Software Architecture: Towards a Design Handbook for Integrating Software Components," Ph.D. Dissertation, in Department of Electrical Engineering and Computer Science. Massachusetts Institute of Technology, Cambridge, MA, 1996.

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

Greenhouse, A. and Scherlis, W.L. "Assuring and Evolving Concurrent Programs: Annotations and Policy," in International Conference on Software Engineering. Orlando, FL, 2002.

Grinter, R.E., Herbsleb, J.D., and Perry, D.E. "The Geography of Coordination: Dealing with Distance in R&D Work," in GROUP '99. Phoenix, AZ, ACM Press, 1999.

Hassan, A.E. and Holt, R.C. "C-REX: An Evolutionary Code Extractor for C," in CSER Meeting. Canada, 2004.

Henderson, R.M. and Clark, K.B. "Architectural innovation: The reconfiguration of existing product technologies and the failure of established firms," Administrative Science Quarterly (35:1), 1990, pp. 9-30.

Herbsleb, J.D. and Grinter, R.E. "Splitting the Organization and Integrating the Code: Conway's Law Revisited," in 21st International Conference on Software Engineering (ICSE 99). Los Angeles, CA, ACM Press, 1999.

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, pp. 1-14.

Herbsleb, J.D. and Mockus, A. "Formulation and Preliminary Test of an Empirical Theory of Coordination in Software Engineering," in ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), Helsinki, Finland, 2003.

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), 2000, pp. 174-196.

Hutchins, E. Cognition in the Wild. MIT Press, Cambridge, MA, 1995.

Hutchins, E. "The Technology of Team Navigation, in Intellectual Teamwork," in J. Galegher, R.E. Kraut, and C. Egido (Eds), 1990, Lawrence Erlbaum, Hillsdale, NJ, 1990, pp. 191-220.

Kraut, R.E. and Streeter, L.A. "Coordination in Software Development," Communications of the ACM (38:3), 1995, pp. 69-81.

Malone, T.W. and Crowston, K. "The interdisciplinary theory of coordination," ACM Computing Surveys (26:1), 1994, pp. 87-119.

March, J.G. and Simon, H. Organizations. 1958, John Wiley and Sons, New York, New York.

Mockus, A. and Weiss, D.M. "Globalization by Chunking: A Quantitative Approach," IEEE Software (Jan-Mar), 2001, pp.

Osterweil, L. "Software Processes are Processes Too," in 9th Conference on Software Engineering, Monterey, CA, 1986.

Parnas, D.L. "On the Criteria to be Used in Decomposing Systems into Modules," Communications of the ACM (15:12), 1972, pp. 1053-1058.

Pressman, R., Software Engineering: A Practitioner's Approach, 6th Edition, 2004, McGraw-Hill, New York, New York.

Sarma, A., Noroozi, Z., and Hoek, "A.v.d. Palantír: raising awareness among configuration management workspaces," in International Conference on Software Engineering. Portland, Oregon, 2003.

Shaw, M. and Garlan, D. Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, Upper Saddle River, NJ, 1996.

Simon, H. A. (1981). The sciences of the artificial. Cambridge, MA, The MIT Press.

Sosa, M.E., Eppinger, S.D., and Rowles, C.M. "The Misalignment of Product Architecture and Organizational Structure in Complex Product Development," Management Science (50:12), 2004, pp. 1674-1689.

Teasley, S.D. Covi, L., Mayuram S. Krishnan, M.S. Olson, J. "Rapid Software Development through Team Collocation," IEEE Transactions on Software Engineering (28:7), 2002, pp. 671-683.

Thompson, J.D. Organizations in Action: Social Science Bases of Administrative Theory. McGraw-Hill, New York, New York, 1967.

Walz, D.B., Elam, J.J., and Curtis, B. "Inside a Software Design Team: Knowledge Acquisition, Sharing, and Integration," Communications of the ACM (36:10), 1993, pp. 62-77.

Yokoo, M. Distributed Constraint Satisfaction: Foundations of Cooperation in Multi-agent Systems. Springer, New York, 2001.