Formulation and Preliminary Test of an Empirical Theory of Coordination in Software Engineering

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

Audris Mockus
Avaya Labs Research
233 Mount Airy Road
Basking Ridge, NJ, USA 07920
+1 908 696-5608
audris@avaya.com

ABSTRACT
Motivated by evidence that coordination and dependencies among engineering decisions in a software project are key to better understanding and better methods of software creation, we set out to create empirically testable theory to characterize and make predictions about coordination of engineering decisions. We demonstrate that our theory is capable of expressing some of the main ideas about coordination in software engineering, such as Conway's law and the effects of information hiding in modular design. We then used software project data to create measures and test two hypotheses derived from our theory. Our results provide preliminary support for our formulations.

Categories and Subject Descriptors
D.2 [**Software Engineering**]: Management – *productivity, programming teams, software process models.*.

General Terms
Management, Measurement, Performance, Design, Economics, Experimentation, Theory.

Keywords
Empirical theory, coordination, engineering decisions, Conway's Law, empirical studies.

1. INTRODUCTION
Coordination of the engineering work of individual software engineers is a central concern of the discipline of software engineering. Just as the work performed by a program can be partitioned into computation and coordination [10], the engineering work of a software project can be partitioned into the decision-making of individual engineers and the coordination of those decisions so as to produce software with the required characteristics.

There are several kinds of evidence one can adduce in support of the proposition that coordination of engineering work is central to software engineering. Empirical studies of large-scale software development, for example, show that coordination of engineering work is one of the most difficult and pervasive of the problems (see, e.g., [7, 26]). Moreover, many of the foundational ideas of software engineering primarily address coordination problems. For example, the notion of modularity [20], clearly one of the foundational ideas in software engineering, concerns primarily the coordination of software engineering work. Modules, or "work items" as Parnas defined them to be, address how work may be split among teams in a way that does not impose unreasonable requirements for coordination and communication among teams. Modularity is important only because it influences the ability of humans to understand and coordinate their work.

While there is probably little doubt among researchers or practitioners that coordination of engineering work is key for successful software engineering, the idea of coordination is often frustratingly elusive. While it seems clear that coordination often involves good communication, and that coordination concerns constraints among engineering decisions, it is not so clear what it means to enhance coordination, how to tell if good coordination is present in a project, and what precisely are the implications of effective and ineffective coordination.

In this paper, we try to build upon the research literature and upon various common intuitions about what coordination in software engineering is, in order to formulate a reasonably rigorous, clear, and testable "empirical" theory of coordination in software engineering. In the remainder of this section, we distinguish the idea of an "empirical theory" from the usual conception of "theory" in computer science in order to clarify our objective. We then briefly discuss prior work on coordination from which our approach draws. In section 2, we present our theory of coordination in software engineering. In section 3, we present the empirical methods and results from a field study in which we performed a preliminary test of hypotheses derived from our theory. We present our discussion in section 4, and conclude the paper in section 5.

1.1 "Empirical" Theory
"Theory," in the generally accepted use of the term in mathematics and computer science, is quite distinct from "theory" as the term is used in the physical and social sciences. For present purposes, let it suffice to say that in computer science, theory is a rigorous means of reasoning mathematically from axioms, in order to prove theorems that ascribe interesting properties to software systems or models of software systems. In general, there is no sense in which such theories require empirical evidence in order to be believed. Given the axioms and correct proofs, the conclusions follow logically, not contingently.

"Theory," as the term is used in science, differs in several important respects. Scientific theories, for example, may or may not be formulated mathematically. Theories are generally expressed mathematically in physics, as they sometimes are in other branches of science, such as certain areas of biology and psychology. Despite the many advantages of mathematical formulations, there are many key scientific theories that are simply not susceptible of mathematical expression. In either case, whether the theory is or is not expressed mathematically, a theory does not stand on its axioms and the validity of its arguments alone. One must be concerned always to ask if the phenomena of interest actually are accurately described by the theory. The most brilliant and rigorous of theories runs the risk that it may simply be wrong, i.e., an inaccurate description of how the world works.

We will call theories of the sort found in science "empirical," drawing attention to the fact that they stand or fall not only on their internal consistency, but on the weight of evidence. For the remainder of this paper, when we use the term "theory" we mean "empirical theory," unless otherwise indicated.

Empirical theories are ordinarily tested by drawing out the implications of the theory for observable phenomena, i.e., generating hypotheses, then observing or constructing situations in which these hypotheses can be tested. Hypothesis testing requires both that such hypotheses can be formulated, and that the relevant observations can be made. Evaluation of empirical theories is almost never achieved with one or even a few tests of hypotheses, since disconfirmation can usually be accommodated with adjustments in the theory, and confirmation can generally be explained by more than one theory. Theories evolve or are replaced on the basis of sustained research programs.

It is important to note that testing hypotheses derived from empirical theories is not the only basis for doing important and valid empirical research in software engineering. The vast majority of empirical work is what we would call empirical "validation" research that has as its main purpose validating claims made in support of the advantage of some new innovation, be it a tool, a process, or a methodology. There has been a trend in recent years toward more emphasis on careful validation, which we wholeheartedly applaud. We see this as distinct from theory-based empirical research. This is a distinction we develop throughout the paper, and which we hope will become much clearer after we provide an example of an empirical theory.

In light of this discussion of empirical theory, we can restate our goals:

- to formulate an empirical theory of coordination in software engineering,
- to identify testable hypotheses that follow from this theory, and
- to show examples of precisely how one can go about empirically testing such hypotheses.

The theory we propose has roots in several lines of research, each of which we will briefly and selectively review in the remainder of this section.

1.1.1 Interdisciplinary Theory of Coordination

Coordination problems in many fields have similar properties [17]. 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 [6, 8].

While such a broad, general approach has considerable appeal, we do not believe that software engineering is sufficiently mature, except in a few fairly isolated cases, for an approach that requires exhaustive cataloging of dependency patterns among engineering decisions. Design work, in particular, is extraordinarily complex, and is structured by very complicated patterns of constraints among engineering decisions (see, e.g., [23]). While we may now be able to identify general patterns underlying some of them (e.g., resource conflicts), and we may eventually identify many more, the current state of engineering is such that dependencies among engineering decisions appear to have enormous variety, and seem unlikely to be susceptible, in general, of such "cookbook" solutions. Our approach, in contrast, does not identify and consider particular coordination problems, but rather just the sets of decisions as they constrain decision-makers.

1.1.2 Distributed Cognition

Many complex tasks are best understood as cognitive or problem-solving process that are distributed over individuals and artifacts (e.g., [13, 14]), distributed over time, and partially embedded in the habits, practices, and routines of the people who carry out the cognitive activities (e.g., [3]). While we are not aware of any attempts to formally and explicitly describe the coordination aspects of distributed cognition, the view of coordinated activity as many interdependent tasks, where coordination occurs by means of communication and sharing of artifacts, and is embedded in a social and organizational context has much in common with our view. In fact, our work could be considered a formalization of one key aspect of distributed cognition, i.e., the impact of mutual constraints among decisions.

A related line of work in artificial intelligence, called distributed artificial intelligence, has specified frameworks for coordinating activities among distributed agents for tasks such as regulating traffic signals or factory floor robots (see, e.g, [9]). Distributed artificial intelligence of necessity incorporates an explicit model of coordination policies. The tasks, however, are much smaller and simpler than typical industrial software engineering tasks.

1.1.3 Geographically Distributed Software Engineering

Coordination issues become more apparent when the usual modes of coordinating activity are interrupted. In geographically distributed software engineering, projects are split across sites, with a resultant nearly total absence of informal communication among developers [11]. As one would expect, the reduction in communication appears to lead to longer cycle times [12]. The effect appears to be caused by the involvement of more people in distributed work items that for comparable work items where all work is performed at a single site. In fact, a statistical model of interval for work items shows that the number of people involved in a work item (which is presumably a reasonable indicator of coordination issues) is the most significant predictor of interval, in a model that includes other important variables such as size and complexity of the change, and other factors [12]. As will become evident, the delays associated with geographically distributed software engineering can be seen as a special case of the coordination phenomena described by our theory.

1.1.4 Toward Empirical Theory for Software Engineering

We conclude this section by discussing very briefly the requirements and motivation for empirical theory specifically in

the field of software engineering. First, we note that there is considerable overlap between the subject matter of software engineering and several fields of social science, such as psychology, anthropology, sociology, and organizational science. We have much to learn from these fields, both in empirical methods and substance. Yet we suspect that software engineers and researchers in software engineering, who are generally trained in computer science or an engineering discipline, will never be comfortable with the theories of social science. Such theories are seldom formulated in a way that software engineers would consider sufficiently rigorous and precise. Just as the various social sciences have evolved their own theoretical traditions which differ markedly from each other, and for good reason, we believe software engineering must evolve its own tradition of empirical theorizing and hypothesis testing. We offer our theory, in part, as a modest step in this direction.

Second, while empirical investigation has, over time, assumed increasing importance in software engineering, due to the pioneering efforts of numerous investigators, (e.g., [1, 2, 21, 24, 25]) we believe that an increased emphasis on theory can have a major impact on empirical investigations by providing a mechanism whereby results become more cumulative. Empirical investigations in software engineering often have a one-off flavor, i.e., are aimed at validating a particular method or technique, or discovering the circumstances under which it is effective. We have no quarrel with such studies – validation of claims about specific software engineering innovations is a vital facet of software engineering research, and will continue to be so.

What is relatively rare in software engineering, however, is the sort of theory-driven investigation that dominates the sciences. Theories provide perspectives for viewing large classes of phenomena, understanding the fundamental principles by which they operate, and making predictions about what should happen in specific cases. Much empirical research is typically focused on testing hypotheses derived from the theories. These tests of hypotheses may or may not have immediate practical consequences (in contrast to empirical validations, which one hopes would always have results of immediate practical interest). Results accumulate over time into a more comprehensive, well-established, and nuanced view of the field as theories are supported, falsified, and modified on the basis of empirical results.

This is slightly different from the usual view taken in empirical studies in software engineering. Generality, or external validity, of a study is most often analyzed as a function of "representativeness," e.g., of the task, the subjects, and other characteristics of the study (see, e.g., [27] pp. 72-73; [16] p. 732). The "representativeness" strategy uses essentially the logic of sampling (i.e., the subjects should ideally be a random sample of the population to which one would like to generalize, the task should be a random sample of the population of tasks, etc. [4] to establish legitimate generalizations.

While this view is well established and perfectly sound, it is not the only legitimate basis for generalization in science (see, e.g., [19]). For example, one might create in the laboratory a phenomena (e.g., by using a particle accelerator) that seldom if ever occurs in nature, but that permits one to test a hypothesis derived from a theory that purports to explain many phenomena, whether those phenomena occur naturally or are artificially generated. The experiment should be judged by how clearly it tests hypotheses associated with the theory, and perhaps by how important, clear, and general the theory appears to be. We believe that research in software engineering would benefit from more theory-based research, and the cumulative results that such research provides.

In this section, we have discussed the need for empirical theories, some of the antecedents of our empirical theory of coordination, and specific reasons for theory development and testing in software engineering. In the next section, we present our theory.

2. EMPIRICAL THEORY OF COORDINATION (ETC)

Our focus is to propose a simple and well-defined model that would describe coordination in software engineering, so we can qualitatively express a number of principles, laws, and practices in the field of software engineering. More specifically, we want to represent coordination of engineering decisions in a precise way so that meaningful theoretical statements can be formulated and testable hypotheses generated. We use the word "engineering" in a broad sense, meaning essentially, designing and constructing an artifact with required characteristics.

Before developing the theory itself, let us say a word about the intuitions behind it. We assume that there is a single (very large, but finite) set of engineering decisions that characterize software projects in general. One can then think of any particular software project as defined by the combinations of choices for those decisions that satisfy the requirements for that project. (Not every possible decision will apply to every possible project – we can handle this case simply by assuming that one possible choice for each decision is "does not apply.") We simplify by not considering that different combinations of choices will satisfy the requirements more completely or less completely. We consider that each combination of choices is associated with a binary value, i.e., it does or does not satisfy the requirements.

Software engineering work proceeds by making choices for all of the decisions. As each is made, fewer decisions remain, until all of the decisions are made, resulting in a final product that may or may not satisfy the requirements. There is some degree of concurrency in making the choices and the choices are potentially made by many different people. Information about the choices that have been made at any given time is imperfect. Decisions can be made more than once, i.e., one choice can be retracted and a new choice made.

We make no attempt to enumerate, or even create a taxonomy of engineering decisions. Rather, the theory is concerned with the patterns of dependency among them. A metaphor that has driven some of our thinking about theory development is the kinetic theory of gasses (see, e.g., [22]). By making a few assumptions, e.g., about elasticity and velocity of highly idealized molecules, some important large scale behaviors of gasses (e.g., the relation among pressure, temperature, and volume) can be understood and predicted. Our theoretical view of engineering decisions is also highly idealized, but we believe that it is useful for understanding and making predictions about certain phenomena, i.e., those associated with coordination in software projects. Explanations of other types of phenomena will no doubt require somewhat different views of engineering decisions.

Engineering decisions are often mutually constraining. Making one decision generally limits the alternatives from which one may choose when making some other decisions. If constraints are violated, then one or more decisions must be reconsidered or it will not be possible to meet the requirements. Reconsidering one decision may require reconsideration of additional decisions, again, because of mutual constraints.

The essential problem of coordination that concerns us in this paper[1] is given by this property of mutual constraint among engineering decisions, and by the necessity, in all projects of significant size, of assigning responsibility for decisions to different people. The problem of coordination, then, is the problem of ensuring that these mutual constraints are recognized and correctly acted upon as the engineering work proceeds.

2.1 Key definitions

Denote each engineering decision as a variable X_i that can take values $x_{ij(i)}$, where $j(i)$ indicates that the range of possible values varies with i. Selecting a choice for decision X_i is equivalent to assigning a value to this variable. A design space is the set of all possible assignments of the set of variables representing the engineering decisions that have to be made. The goal space is the subset of the design space, each member of which satisfies the requirements. Each element of the goal space is a solution.

The engineering 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 further assignments to other variables. The constraints are defined implicitly by the feasibility function.

Define a feasibility function

$f(x_{1j(1)},\ldots, x_{nj(n)})$ ={1 iff product satisfies requirements,
0 otherwise}

Such function implicitly defines a set of feasible choices for each decision.[2] Feasible choices for decision X_i denoted as $FC(X_i)$ are defined as a set

$$x_{ij*}:\forall k \neq i, \exists j(k) \text{ such that } f(x_{1j(1)},\ldots x_{ij*},\ldots x_{nj(n)}) = 1$$

Obviously, a particular choice in each decision has effects on feasible choices in other decisions.

Effects of a decision $j(k): X_k = x_{kj(k)}$ on a decision l, denoted $E(X_l \mid X_k = x_{kj(k)})$, is the set difference

$$FC(X_l) - FC(X_l \mid X_k = x_{kj(k)})$$

where $FC(X_l \mid X_k = x_{kj(k)})$ denotes the set of feasible choices given that the set of possible choices in decision X_k has been narrowed to $x_{kj(k)}$.

In other words, the effect of decision k on decision l is the difference between feasible choices in the design space of variable X_l before a variable X_k is assigned a value and the design space after X_k is assigned a value. This definition obviously generalizes to effects of a group of decisions on a single decision.

We can also define maximal effects where

$$ME(X_l \mid X_k) = \cup_{x_{kj(k)} \in FC(X_k)} E(X_l \mid X_k = x_{kj(k)})$$

defines a set of choices for X_l that can be made infeasible by at least one feasible value of X_k.

The state of a project can be defined as the set of decisions that have been taken, which implicitly defines a set of remaining decisions and their feasible choices.

2.2 Common "Laws" of Software Engineering

In this section, we provide two examples of "laws" or generally accepted beliefs about coordination in software engineering in terms of our theory, i.e., by considering various strategies of taking decisions and their effects.

First consider a partition of decisions into non-overlapping module-induced clumps M_p. The clump M_p provides information hiding if decisions made outside the clump do not have effects on the decisions made within the clump.[3]

More specifically:

$$\forall p, i, k : X_i \in M_p, X_k \notin M_p, ME(X_i \mid X_k) = \{\}$$

We call this the principle of modularity because it expresses the ideas of modularity suggested by Parnas [20]. The clumps of decisions M_p can be induced by pieces of architecture, i.e., when decisions are grouped if they pertain to a part of software architecture.

We define the "Parnas" effect for a given decision X_i as the number of decisions in other modules that have nonempty effects on X_i. The Parnas effect for a system is the sum of the Parnas effects for all of the X_i.

Alternatively, the clumps of decisions T_c can be induced by an organizational unit (e.g., teams or individuals) involved in the software project, i.e., when decisions are grouped if they are made by an organizational unit, such as a development team. Conway's Law [5] states that the structure of a system resembles the structure of the organization that designed it.

Conway's law can be formulated as follows:

$$\forall p \exists c : M_p \subset T_c,$$

i.e., no modules are implemented by several organizations. In other words there exists a homomorphic function with the domain of all modules and the range of teams. As Conway [5] and others [11] have noted, even if this function exists for an initial design, the evolution of the design will generally outpace organizational change, and lead to "violations" of Conway's Law. The number of variables that do not fit in this homomorphic relationship could be said to be the "Conway" number of the system.

Finally, we wish to suggest that many more of the basic conceptual tools of software engineering can be expressed in comparable ways. For example, design methods can be seen as ordering and clustering engineering decisions with respect to time, as well as providing various notations that help to make the current state of the design more visible. We will say a bit more about such possible applications in our discussion.

---

[1] There may, of course, be additional coordination problems, such as competition for resources, and so on, that do not concern us in this paper.

[2] Realistic decision functions will include many other considerations besides feasibility, e.g., option value [24].

[3] For simplicity, we ignore decisions concerning module interfaces.

In this section, we have laid out the basic definitions of our theory, and shown how some interesting "wisdom" about coordination in software engineering can be expressed. We have not yet said anything about deriving testable hypotheses, which we address in the next section.

2.3 Additional Assumptions

Ultimately, it must be possible to bring observation to bear in order to test hypotheses derived from any theory. If this is not possible, the theory is either vacuous or tautological, and therefore uninteresting.

In the previous section, we have hinted at some additional assumptions that are necessary in order to render concrete predictions from the theory we propose. In this section, we will develop these assumptions more explicitly. The first set of assumptions concern the effects of making infeasible choices (i.e., assignments for which the feasibility function evaluates to 0).

The possible effects of infeasible choices are as follows:

A1. defects, faults, errors, failure to complete project (if the infeasible choice is never reconsidered)

A2. rework (infeasible choice is identified and changed, and perhaps other decisions dependent on it must also be changed)

A3. longer cycle time (introducing then changing infeasible choices will cause the project to take longer, since rework consumes time)

A4. lower productivity (introducing and changing infeasible choices will lower productivity, since rework consumes resources)

A second set of assumptions is somewhat more speculative, and has to do with factors that make infeasible choices more likely or less likely. The fundamental underlying ideas are 1) that when the person who has responsibility for a decision is aware of the constraints that relate that decision to other decisions, it is more likely that the decision maker will make a feasible choice, and 2) more frequent communication among decisions makers who are making mutually constraining decisions increases the likelihood of making a feasible choice. Each of the assumptions listed below should be read to implicitly include a "ceteris paribus" clause, i.e., in each case, we assume that all other relevant factors are held equal.

Feasible choices for mutually constraining decisions are more likely to be made when:

A5. the decisions are made by a single person, or fewer people (rather than more people);

A6. they are made by people in frequent (rather than infrequent) communication with each other; and

A7. the constraints that bear on a decision are highly visible to the decision maker.

While we treat these assertions as assumptions for our present purposes, it would be desirable to test many of them in future research. Such a dual role of an assertion as both assumption and testable hypotheses is not uncommon in scientific contexts. For the purposes of one experiment, for example, one might simply assume the validity of the readings of a particular measuring instrument or procedure. If any doubt arises as to these assumptions, however, it is reasonable to perform additional experiments for the purpose of testing these assumptions. It is seldom possible to test the entire interrelated complex of assumptions and hypotheses involved in a line of experiments at once.

One way to test our formulation, e.g., of Conway's Law, would be to show that higher Conway numbers are associated with more rework, longer cycle time, or other indicators of the presence of infeasible decisions. Yet it is very difficult to actually determine the Conway number of a development, since it is generally not possible to completely specify the set of engineering decisions and the constraints among them for any project of significant size. Given our definition of a development's Conway number, however, this would be required in order to directly measure a Conway number.

While such a complete enumeration of decisions and constraints is generally not feasible, it may often be feasible to establish that one system, or one set of changes to a system, has a higher Conway number than another system or set of changes. If it is possible to eliminate or account for other possible sources of variation, it may be possible to make relevant measurements, e.g., of rework, cycle time, or productivity, and test hypotheses about such things as Conway numbers on this indirect basis. Supporting such indirect tests was, of course, a primary motivation in introducing the assumptions in this section.

In this section, we have considered various assumptions that we find necessary in order to derive testable hypotheses from the theory, and to perform observations relevant to those hypotheses. In the next section, we describe the methods we employed in a preliminary test of the Conway and Parnas hypotheses.

3. EMPIRICAL METHODS AND RESULTS

We have two aims in designing empirical procedures to test hypotheses derived from our theory. First, and most importantly, we intend this to be a proof of concept, in the sense that we illustrate that such hypotheses are testable. Second, we generate evidence that is relevant to the substance of the theory itself, although we do not claim that our results are final or definitive.

In broad strokes, our method is as follows. We use data from a modification request (MR) system from a development project at Avaya Technologies. Using this MR data, we construct two graphs, one that shows the flow of work among individuals, and one that shows the organization of files in terms of which files tend to get changed together. We extract scalars from these graphs to represent theoretically relevant properties, and use these scalars in regression analyses to predict important quantities such as productivity and cycle time. We use assumptions from section 2 and the results of these regression models to test two ETC hypotheses. We spell out the details of this method in the remainder of this section.

3.1 Project and Site

We study embedded software for a communications device with a user interface, running on a popular embedded operating system. The product had significant changes in hardware and substantial increase in functionality leading to extensive software development over the range of two years. At the time of the writing the product is approaching its fourth major release.

More than 30 active developers participated over more than two and a half years modifying approximately 5,000 files. In all, the changes included more than 10,000 delta adding more than 3 million lines of code for this system. Most of the developers were located in one site in the eastern United States and a very small group of developers were located in Australia.

Most of the code was written in C language, some also in Java and C++, and assembly language. One release, for example, contains approximately 1 million lines of code (LOC) of C and C++ and 200,000 LOC of assembly language, and 100,000 LOC of Java.

3.2 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 version of an MR process follows. The developers are assigned (or, more often, assign themselves) a new feature or a defect to work on. In case of defects, they investigate the problem, make necessary changes and submit an MR for integration. In 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 in the analysis used Sablime configuration management system that uses SCCS version control system to manage code changes. We extracted workflow relationships by processing Sablime MR history file for each change. This file contains the history 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.

3.3 Graph Construction and Hypotheses

We used the MR data to construct two types of graphs that would allow us to generate measures of the properties of interest. Our theory of engineering decisions focuses on how the conditions under which engineering work is done influences important outcomes. The outcomes (e.g., cycle time, productivity, quality, rework) can often be measured for individual MRs.

On the other hand, the factors that influence these variables operate at the larger scale of how work relates to coordination among individuals and is distributed across software modules. In order to generate measures of these independent variables, we constructed two types of graphs that allow us to capture the important conditions under which the MR work was done.

3.3.1 Empirical Workflow Graph

From a theoretical point of view, one would expect, from assumption A5, that the more people with whom one must coordinate one's mutually-constraining engineering decisions, the

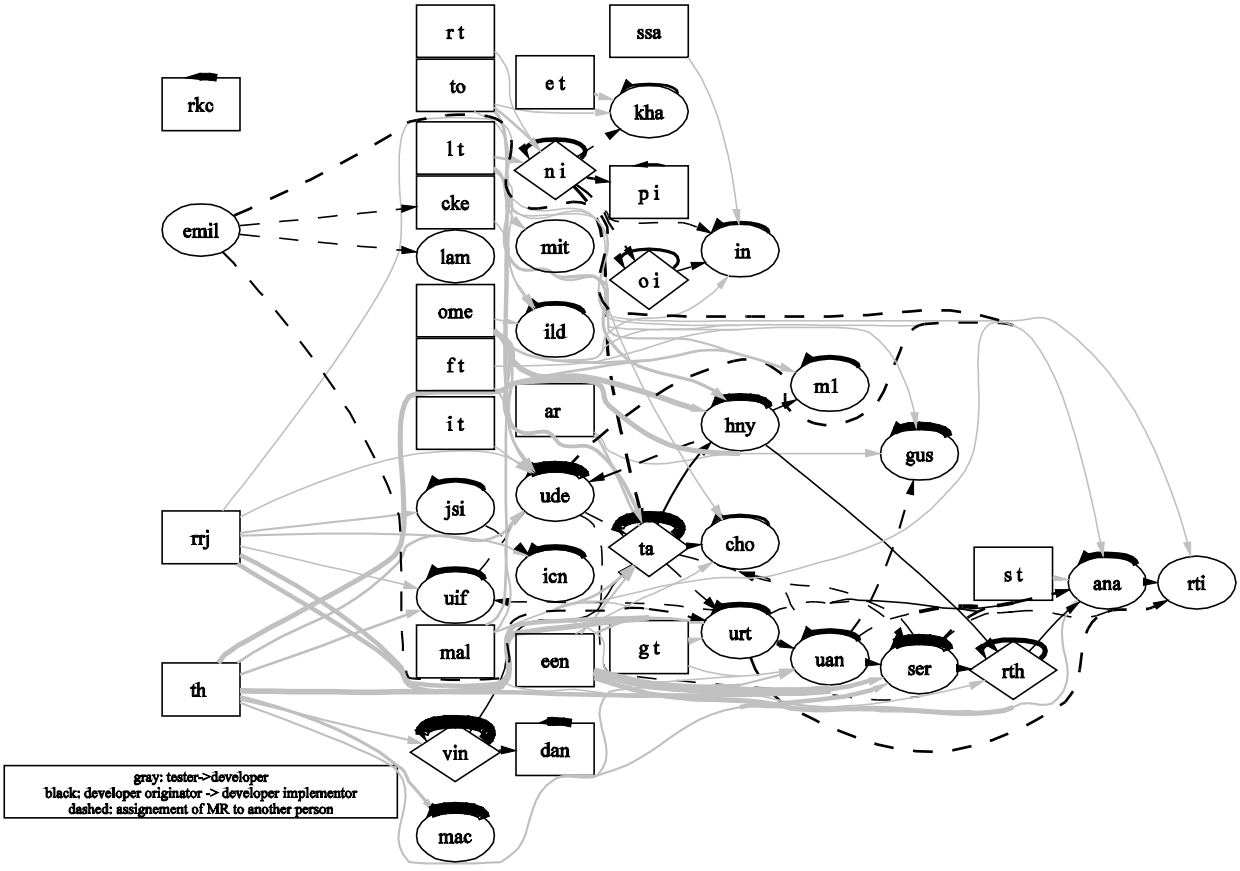

Figure 1. Graph of work flow constructed from modification request data.

more infeasible decisions one is likely to make, hence by assumption A4, the less productive one is likely to be.

In order to investigate this prediction, we constructed an empirical workflow graph. In this directed graph, each person involved in the project is represented as a node. Arcs represent instances of workflow, i.e., when some type of work on the MR (e.g., MR is created, assigned, or some code is committed in order to resolve the MR) by one person is followed by work on the same MR by another person.

So, for example, if person A creates an MR, and the next recorded activity is that person A assigns the MR to person B, and person B subsequently contributes code to the MR, we record an arc from A to B. If person B assigns the MR to person C, we record an arc from B to C. The workflow graph is shown in Figure 1. Each person is shown as a node: square nodes represent testers, oval nodes represent developers, and diamond nodes represent developers who reassign significant portion of their MRs to other developers, i.e., have worked as change coordinators deciding whom to assign unassigned MRs. The gray links indicate a connection from a tester who raises an MR to a developer who works on it. The dashed links indicate MR reassignments among developers, and the black links indicate MRs created by developers and assigned to another (or the same) developer. The thickness of links is proportional to the square root of the number of MRs, and direction of the link is in the direction of workflow.

While the appearance of the graph is cluttered, one can see, for example, that testers typically assign MRs to a particular group of developers. Most MRs are raised and solved by the same developer. Some, but not all, developers work on assignments.

From this empirical workflow graph, we derive several relevant measures. For each node (i.e., person) in the graph, we count the number of MRs the person assigned to him/herself (self), the number of MRs assigned to a person by others (in), the number of MRs assigned by a person to others (out), the total number of individuals who assigned at least one MR to a person (inDegree), and the total number of others to whom a person assigned at least one MR (outDegree).

The inDegree variable is the best indicator of the number of people whose work constrains a given developer's decisions. If I receive work handoffs from many different people, I will increase the likelihood that I will make infeasible decisions. (Handing work to additional people should not have a similar effect on *my* productivity, although my involvement will presumably generate more constraints on the decisions of those to whom I make the handoff).

We are able to estimate productivity from our data (details below), which gives us hypothesis 1:

H1: Developers with higher inDegree (more people assigning work to them) will have lower productivity.

3.3.2 Work Modularity Graph

If engineering decisions concerning one module have no effects on engineering decisions in other modules, then all relevant constraints are revealed by looking only at a portion of the system (the module) rather than the whole system. The constraints should therefore be much more visible to the developers, which (by assumption A7) will lead to fewer infeasible decisions. By assumption A3, fewer infeasible decisions should lead to shorter cycle times.

In order to investigate this prediction, we constructed a work modularity graph which partitions the code into empirically-derived modules (see [18]) to investigate the properties of MRs where all of the work occurs within a single module and those that require work in more than one module. In this graph, files are the nodes, and edges are drawn between nodes whenever those files are both modified in order to perform the work for a single MR. Files are clustered into two modules (one is shown with a gray background) based on an algorithm that minimizes the number of edges between files in different modules.

Thickness of the link represents the square root of the number of MRs. Figure 2 shows only links and files that these links touch where links contain more than 8 MRs within a module and more than 3 MRs for links that cross module boundaries. These constraints are needed to produce a small figure that could be read. The file names are converted to random three letter combinations.

We construct modules on this empirical basis rather than some other basis, such as directory structure, in order to perform a more meaningful test of modularity in the Parnas sense [20], i.e., modules where changes or work-items are contained within a module. Modules defined around directory structure often do not have this property, therefore using the modules defined by directory structure would not bear in any clear way on the theory.

The reason for this difference, of course, is the information-hiding property of modules.

Given that we can measure the cycle time, or interval, for MRs (see below), we have hypothesis 2:

H2: Modification requests that require work in different modules will have longer cycle times than modification requests that require work in only a single module.

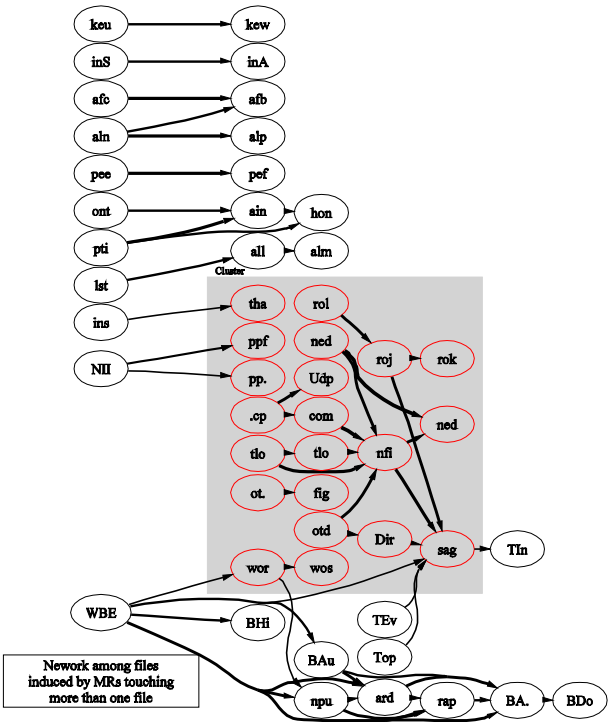

Figure 2. Graph of file and module structure.

3.4 Statistical Models

3.4.1 Productivity and Coordination

The predictors in the model are shown in Table 1 below. We have transformed variables by square root transformation to obtain more Gaussian distributions of the variables.

In the regression model below, the response is productivity, which we measured as the number of MRs divided by the total time participating in the project. Other measures of productivity are possible (e.g., lines of code added, number of commits or deltas). However, many MRs that do not result in changes to the code involve considerable investigative work, e.g., to try to reproduce an error, or to understand how the code works or is supposed to work). For this reason, we decided that measures based solely on changes to the code could not adequately reflect these sorts of productive work.

Table 1. Variables used in regression model for productivity.

self	Square root of the number of MRs initiated and assigned by a person to him/herself
in	Square root of the number of MRs assigned to the person
out	Square root of the number of MRs assigned by the person
inDegree	Square root of the number of individuals who assigned MRs to the person
outDegree	Square root of the number of individuals who were assigned MRs by the person

The first factor is MRs a developer creates and completes and second is the number of MRs assigned to a developer. These factors represent aspects of individual productivity. The number of MRs that the developer assigns to other people is the next factor. This covariate might decrease productivity, since developer has to spend time to assign these MRs but they are not counted in the individual productivity number. The last two factors are the number of individuals assigning the MR to a person (i.e., the variable our hypothesis is concerned with) and the number of individuals that the person assigns MRs to. The results are in the Table 2 below.

The results indicate that productivity of an individual developer significantly increases (unsurprisingly) with the number of MRs they resolve (selfMr and inMR), and significantly decreases, as our hypothesis *H1* predicts, with the number of people who assign MRs to them (inPeople). This appears to indicate that the more people whose work has an effect on a given developer's work, the lower the productivity of the given developer. (The inspection of multicolinearity, normality, and residuals showed nothing unusual. The multiple R-squared of .69 indicates a good fit of the model to the data.)

Table 2. Regression performed on productivity, using variables from the empirical workflow graph.

Variable	Coefficient	Std. Error	t value	Pr>|t|
Intercept	6.4	0.96	6.7	<0.001
self	0.47	0.18	2.7	0.01
in	1.16	0.32	3.6	0.001
out	0.42	0.82	0.5	0.6
inDegree	-2.1	0.68	-3.0	0.006
outDegree	-1.1	1.4	-0.8	0.41

3.4.2 Cycle Time and Modularity

Here we are looking at the interval for the MRs that involve changes to code. We have selected a large module using globalization techniques [18]. The module contains 257 files out of total 872 files in the system and about four percent of MRs touching the files in the module also modify files outside the module.

Table 3. Predictors used in regression model for cycle time.

Other	indicator if the MR was created not by developer
Nreleases	number of releases the MR is included
Nfiles	Logarithm of the number of files MR touches
Developer	developer resolving MR
Multi-mod	indicator of MR crossing boundary of a module

The interval is modeled by first including factors that are likely to affect the interval and then adding the factor that we would like to test. The first covariate identifies if another person has created the MR (this factor is likely to increase the interval because of the need to communicate the issue to another person and has been observed previously [12]), how many releases MR was included in (indicating a problem serious enough to be fixed in older or newer releases and the dependency issues associated with changing code in multiple releases), number of files touched that indicates the complexity of the issue, the identity of developer that may affect the time it takes to solve the problem, and, finally, an indication if the MR crossed the module boundary. In general, one would expect changes that cross module boundaries (in comparison to those that do not) to require understanding of the module internal structure as well as its surroundings.

Table 4. Regression performed on variables from the work modularity graph.

Variable	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	11.3	0.24	46.7	< 0.001
Other	2.46	0.098	24.9	< 0.001
nReleases	1.04	0.11	9.3	< 0.001
NFiles	0.18	0.05	3.3	0.001
Multi-mod	0.41	0.19	2.2	0.027

(The inspection of multicolinearity, normality, and residuals showed nothing unusual, and the multiple R-squared of .631

indicates a good fit. The 27 regression coefficients, one for each developer, are excluded because of space considerations. They ranged from -1.2 to 1, with mean 0 and median -0.1.)

This result supports *H2*.

4. DISCUSSION

Our empirical results illustrate initial steps toward validating our theory. Before we have a high degree of confidence that our formulation of the Parnas and Conway effects is really supported, we will need to conduct additional studies, both replications and extensions. We regard this as a promising start that merits further consideration of how other ways of clustering, sequencing, visualizing, and partitioning engineering decisions influence outcomes.

As we mentioned in the introduction, we suspect that the bare bones ETC theory presented here can be extended in various ways to represent other important coordination issues. For example, one could represent coordination effects of different ways of ordering design decisions by examining the cardinality of the effects of the decisions. Under some circumstances, for example when engineers fear that the effects of early decisions are difficult to discern, one might want to make choices early on that have minimal effects. This represents a strategic choice not to foreclose choices for future decisions more than necessary. In other circumstances, one might want to make early choices with maximal effects in order to reduce the design space as much as possible and simplify future decisions. Different design methods can be seen as prescribing standard orderings of engineering decisions that embody these sorts of tradeoffs.

For purposes of illustration, consider Michael Jackson's "problem frames" [15]. By adopting, say, a high level decomposition consisting of "simple workpieces" and "display" problem frames, there are several effects relevant to coordination among decisions. First, such "high level" decisions (i.e., decomposition into these two known frames) can be seen as making many individual decisions at once, and doing so in a way that (one hopes) introduces no infeasible choices. Second, the state of the engineering design space after the "high level" decision, i.e., the set of all decisions not yet made and the constraints on those decisions, is partitioned such that decisions concerning one problem frame have few effects (all of which should be explicit and highly visible -- see below) on decisions concerning the other problem frame. With appropriate assignment to teams, it permits one to take advantage of Conway's Law.

Another area we believe is ripe for future research is the "visibility" of the project's current state, where current state is the set of decisions not yet made and their remaining feasible choices. If one considers a graph where the decisions are nodes and constraints are edges, one might speculate that visibility for any given engineer is a function of a) the number of nodes reachable from the nodes representing the decisions assigned to a developer (larger number of reachable nodes means more one needs to be aware of), b) the predictability of the choices for decisions that are reachable but not assigned to the given engineer (high predictability means guesses or assumptions about what other engineers will decide are likely to be correct), and c) the form, content, and clarity of information about current project state.

While we regard ETC as promising, these results are clearly in need of further test. Since many projects use modification requests in a similar way, the techniques we employ can be applied to data from many settings. We would also like to see tests of many more hypotheses derived from our theory, in order to determine if in fact it provides a useful, empirically valid view of coordination in software engineering. We also note that while we consider our assumptions to be very plausible, they require independent empirical tests in order for the theory to be validated. The assumptions provide a critical bridge between theory and observation, and they should be subjected to rigorous tests. As we said in the introduction, empirical tests of theories, as opposed to individual hypotheses, require a program of research, on which we have now just embarked.

5. CONCLUSION

This paper makes three contributions. First, we formulate an empirical theory of coordination that is precisely specified, and which can account for many important phenomena in software engineering. Second, we show how this theory, when used in conjunction with several explicit and plausible assumptions about the possible effects of infeasible choices and the circumstances likely to lead to infeasible choices, generates testable hypotheses. Finally, we performed empirical investigation of two hypotheses derived from the theory.

We believe that coordination is an enormously important aspect of software engineering, where development and testing of empirical theory is vitally important in order for research to progress. We are convinced that many areas of software engineering would benefit from development and testing of empirical theory. We offer our theory as an example that we hope will stimulate discussion, and lead to further development and testing of empirical theory in coordination and other areas where we need to achieve a better understanding of the contingent properties of important phenomena in software engineering.

6. ACKNOWLEDGEMENT

The authors wish to acknowledge support through the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298.

7. REFERENCES

1. Basili, V.R., The Role of Experimentation in Software Engineering: Past, Current, and Future. in *18th International Conference on Software Engineering (ICSE 18)*, (Berlin, Germany, 1996), IEEE Computer Society Press, 442-449.
2. Basili, V.R., McGarry, F.E., Pajerski, R. and Zelkowitz, M.V., Lessons learned from 25 years of process improvement: The Rise and Fall of the NASA Software Engineering Laboratory. in *International Conference on Software Engineering*, (Orlando, FL, 2002), ACM Press, 69-79.
3. Brown, J.S. and Duguid, P. Knowledge and organization: A social-practice perspective. *Organization Science*, *12* (2). 198-213.
4. Campbell, D.T. and Stanley, J.C. *Experimental and Quasi-Experimental Designs for Research*. Houghton Mifflin, Boston, MA, 1963.
5. Conway, M.E. How Do Committees Invent? *Datamation*, *14* (4). 28-31.
6. Crowston, K. A taxonomy of organizational dependencies and coordination mechanisms. in Malone, T.W., Crowston, K. and Herman, G. eds. *Tools for Organizing Business Knowledge: The MIT Process Handbook*, MIT Press, Cambridge, MA, in press.
7. Curtis, B., Krasner, H. and Iscoe, N. A Field Study of the Software Design Process for Large Systems. *Communications of the ACM*, *31* (11). 1268-1287.
8. Dellarocas, C. A Coordination Perspective on Software Architecture: Towards a Design Handbook for Integrating

Software Components *Center for Coordination Science*, Massachusetts Institute of Technology, Cambridge, MA, 1996.
9. Durfee, E.H. Organisations, Plans, and Schedules: An Interdisciplinary Perspective on Coordinating AI Systems. *Journal of Intelligent Systems*, *3* (2-4). 157-187.
10. Gelernter, D. and Carriero, N. Coordination languages and their significance. *Communications of the ACM*, *35* (2). 97-107.
11. 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, 1999), ACM Press, 85-95.
12. Herbsleb, J.D. and Mockus, A. An Empirical Study of Speed and Communication in Globally-Distributed Software Development. *IEEE Transactions on Software Engineering*, *To appear*.
13. 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). 174-196.
14. Hutchins, E. The Technology of Team Navigation. in Galegher, J., Kraut, R.E. and Egido, C. eds. *Intellectual Teamwork*, Lawrence Erlbaum, Hillsdale, NJ, 1990, 191-220.
15. Jackson, M. *Problem Frames*. Addison-Wesley, Boston, MA, 2001.
16. Kitchenham, B.A., Pfleeger, S.L., Pickard, L.M., Jones, P.W., Hoaglin, D.C., Emam, K.E. and Rosenberg, J. Preliminary Guidelines for Empirical Research in Software Engineering. *IEEE Transactions on Software Engineering*, *28* (8). 721-734.
17. Malone, T.W. and Crowston, K. The interdisciplinary theory of coordination. *ACM Computing Surveys*, *26* (1). 87-119.
18. Mockus, A. and Weiss, D.M. Globalization by Chunking: A Quantitative Approach. *IEEE Software*, *January - March*.
19. Mook, D.G. In Defense of External Invalidity. *American Psychologist*, *April*. 379-387.
20. Parnas, D.L. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, *15* (12). 1053-1058.
21. Perry, D.E., Staudenmayer, N.A. and Votta, L.G. People, Organizations, and Process Improvement. *IEEE Software*, *11* (4). 36-45.
22. Reif, F. *Fundamentals of Statistical and Thermal Physics*. McGraw-Hill, New York, 1965.
23. Simon, H.A. The structure of ill structured problems. *Artificial intelligence*, *4*. 145-180.
24. Tichy, W.F. Should Computer Scientists Experiment More? *IEEE Computer*, *31* (5). 32-40.
25. Votta, L.G. and Porter, A., Experimental Software Engineering: A Report on the State of the Art. in *17th International Conference on Software Engineering (ICSE 17)*, (Seattle, Washington, 1995), ACM Press, 277-279.
26. 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). 62-77.
27. Wohlin, C., Runeson, P., Host, M., Ohlsson, M.C., Regnell, B. and Wesslen, A. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Boston, 2000.