Socio-Technical Congruence: A Framework for Assessing the Impact of Technical and Work Dependencies on Software Development Productivity

Marcelo Cataldo
Research and Technology Center
Bosch Corporate Research
Pittsburgh, PA 15212, USA
marcelo.cataldo@us.bosch.com

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

Kathleen M. Carley
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA
Kathleen.carley@cmu.edu

ABSTRACT
The identification and management of work dependencies is a fundamental challenge in software development organizations. This paper argues that modularization, the traditional technique intended to reduce interdependencies among components of a system, has serious limitations in the context of software development. We build on the idea of congruence, proposed in our prior work, to examine the relationship between the structure of technical and work dependencies and the impact of dependencies on software development productivity. Our empirical evaluation of the congruence framework showed that when developers' coordination patterns are congruent with their coordination needs, the resolution time of modification requests was significantly reduced. Furthermore, our analysis highlights the importance of identifying the "right" set of technical dependencies that drive the coordination requirements among software developers. Call and data dependencies appear to have far less impact than logical dependencies.

Categories and Subject Descriptors
D.2.9 [**Software Engineering**]: Management – *Productivity, Programming Teams.* K.6.1 [**Management of computing and information Systems**]: Project and People Management – *Software development.*

General Terms
Management, Measurement, Human Factors.

Keywords
Collaborative software development, coordination, software dependencies.

ESEM'08, October 9–10, 2008, Kaiserslautern, Germany.

1. INTRODUCTION
A growing body of research shows that work dependencies – i.e., engineering decisions constraining other engineering decisions – is a fundamental challenge in software development organizations, particularly in those that are geographically distributed (e.g., [11][16][25][28]). The modular product design literature has extensively examined issues associated with dependencies. Design structure matrices, for example, have been used to find alternative structures that reduce dependencies among the components of a system [19][43]. These research streams can also inform the design of development organizations so they are better able to identify and manage work dependencies. However, we first need to understand the assumptions of the different theoretical views and how those assumptions relate to the characteristics of software development tasks.

This study argues that modularization is a necessary but not a sufficient mechanism for handling the work dependencies that emerge in the process of developing software systems. We build on the concept of congruence introduced by Cataldo et al [10] to examine how different types of technical dependencies relate to work dependencies among software developers and, ultimately, how those work dependencies impact development productivity. Our empirical evaluation of the congruence framework illustrates the importance of understanding the dynamic nature of software development. Identifying the "right" set of technical dependencies that determine the relevant work dependencies and coordinating accordingly has significant impact on reducing the resolution time of software modification requests. The analyses showed traditional software dependencies, such as syntactic relationships, tend to capture a relatively narrow view of product dependencies that is not fully representative of the important product dependencies that drive the need to coordinate. On the other hand, logical dependencies provide a more accurate representation of the product dependencies affecting the development effort.

The rest of this paper is organized as follows. We first discuss the theoretical background concerning the relationship between technical and work dependencies in software development projects. Next, we present the socio-technical congruence framework followed by a description of data, measures and models used in the empirical analysis. Finally, we discuss the results, their implications and future work.

2. THE NATURE OF SOFTWARE DEVELOPMENT AND MODULAR DESIGN

The idea of dividing a complex task into smaller manageable units is consistent with the reductionist view [41][44] which is well developed in the product development literature [19]. Projects, typically, have a general description of the system's components and their relationships or a more detailed report such as architectural or high-level design document. Managers use the information in those documents to divide the development effort into work items that are assigned to specific development teams, minimizing the interdependencies among those teams [13][19][43]. In the system design literature, it has long been speculated that the structure of a product inevitably resembles the structure of the organization that designs it [13]. In Conway's original formulation, he reasoned that coordinating product design decisions requires communication among the engineers making those decisions. If everyone needs to talk to everyone, the communication overhead does not scale well for projects of any size. Therefore, products must be split into components, with limited technical dependencies[1] among them, and each component assigned to a single team. Conway [13] proposed that the component structure and organizational structure stand in a homomorphic relation, in that more than one component can be assigned to a team, but a component must be assigned to a single team.

A similar argument has been proposed in the strategic management literature. Baldwin and Clark [2] argued that modularization makes complexity manageable, enables parallel work and tolerates uncertainty. The design decisions are hidden within the modules, which communicate through standard interfaces. Modularization adds value by allowing independent experimentation of modules and substitution [2]. Moreover, Baldwin and Clark [2] argued that a modular design structure leads to an equivalent modular task structure. Thus, their view aligns with Conway's idea that one or more modules can be assigned to one organizational unit and work can be conducted almost independently of others. In the context of software engineering, a similar approach was first articulated by Parnas [39] as modular software design. Parnas [39] argued that modules ought to be considered work items instead of just a collection of subprograms. Development work can continue independently and in parallel across different modules. Parnas' views also coincide with the theoretical arguments from product design and strategic management literatures.

All three theoretical views rely on two interrelated assumptions. The authors assumed a simple and obvious relationship between product modularization and task (or work) modularization. Hence, by reducing the technical interdependencies among the modules of a system, the modularization theories argue, work interdependencies are reduced, thereby reducing the need for communication among work groups. Unfortunately, there are several problems with these assumptions when applied in the context of software development. One problem is that existing software modularization approaches use only a subset of the technical dependencies, typically syntactic relationships, of a software system [23]. Other potentially relevant work dependencies might be ignored. Moreover, recent empirical evidence indicates that the relationship between product structure and task structure is not as simple as previously assumed, and diminishes over time [11].

Promoting minimal communication between teams responsible for interdependent modules is also problematic. Recent studies suggest that minimal communication between teams, collocated or distributed, is detrimental to the success of projects. The product development literature argues that information hiding, which leads to minimal communication between teams, causes variability in the evolution of projects, frequently resulting in integration problems [46]. In context of software development, de Souza and colleagues [17] found that information hiding led development teams to be unaware of others teams' work resulting in coordination problems. Grinter and colleagues [25] reported similar findings for geographically distributed software development projects. The authors highlighted that the main consequence of reducing the teams' communication was increased costs because problems tended to be discovered later in the development process. Those findings do not suggest that modularization is not useful. They highlight the need to supplement it with coordination mechanisms to allow developers to deal correctly with the assumptions that are not captured in the specification of the dependencies.

Finally, another important problem associated with the assumptions of modular design is the role of change which can be characterized along three interrelated dimensions: the evolution of requirements, the stability of the interfaces between software modules and the dynamic nature of technical dependencies that arise as design and implementation decisions are made.

Evolution of requirements. It is widely accepted among software engineering researchers and practitioners that the requirements of the system become known over time or those requirements change as time progresses [35]. In some cases, the changes in the requirements result in minor alterations of specific development tasks. In other instances, new features have to be added or features under development are eliminated. These events introduce a certain level of dynamism in software development that challenges the determinism and stability assumptions of the modularization approach.

Nature and stability of interfaces. The interfaces between software modules might differ in complexity and little is known about the impact of instability on coordination among development teams. However, recent research has started to examine those issues. Cataldo et al [11] presented case studies where even simple interfaces between modules developed by remote teams create coordination breakdown and integration problems. The authors reported that semantic dependencies were often problematic and they argued that the developers' ability to identify and manage dependencies was hindered by several inter-related factors such as development processes, organizational attributes (e.g. structure, management style) and uncertainty of the interfaces. In relation to the stability of interfaces, de Souza [16] found, in a field study of a large software project, that interfaces tended to change often and their design details tended to be incomplete, increasing the likelihood of future changes to them and leading to serious integration problems. This lack of stability represents a constant challenge for software development organizations in terms of coordination and, ultimately, productivity and quality.

[1] The terms "technical dependency" and "product dependency" are used interchangeably through out this paper. Examples of such relationships are syntactic or semantic dependencies between architectural components or modules.

Dynamic nature of dependencies. Dependencies arise dynamically as part of the development of a piece of code. The act of developing a software system consists of a collection of design decisions, either at the architectural level or at the implementation level. Those design decisions introduce constraints that might establish new dependencies among the various parts of the system, modify existing ones or even eliminate dependencies. The changes in dependencies can generate new coordination requirements that are quite difficult to identify *a priori*, particularly when the dependencies are not obvious, or as a project matures over time [27][42]. Failure to discover the changes in coordination needs might have a profound impact on the quality of the product [15], on productivity [28] and even on the projects' overall design [4]. In addition, little is known about the specific impact of the various types of dependencies that arise among parts of a software system such as explicit versus implicit dependencies or syntactic versus logical dependencies.

The previous paragraphs highlight the limitations of the product modularization approach, which does not necessarily yield an equivalent task modularization structure. The nature of software development such as the attributes and stability of interfaces among modules and the dynamics of technical dependencies, are a constant challenge for software development organizations, particularly for those that are geographically distributed. Mechanisms to complement the modular design approach are required to maintain appropriate levels of coordination among development groups. This leads us to the following research questions:

RQ1: ***How can relevant task dependencies be identified from technical dependencies?***

RQ2: ***What is the impact of those task dependencies on development productivity?***

3. SOCIO-TECHNICAL CONGRUENCE

Product development endeavors involve two fundamental elements: a technical and a social component. The technical properties of the product to develop, the processes, the tasks, and the technology employed in the development effort constitute the technical component. The second element consists of the organization and the individuals involved in the development process, their attitudes and behaviors. In other words, a product development project can be thought of as a socio-technical system where the two components, the technical and the social elements, need to be aligned in order to have a successful project. A key issue is to understand how we can examine the relationship between those two dimensions.

Two lines of work are particularly relevant in this context. First, the idea of "fit" from the organizational theory literature provides the conceptual framework. Fit is defined as the match between a particular organizational design and the organization's ability to carry out a task [6]. This line of research has, traditionally, focused on two factors: the temporal dependencies among tasks that are assigned to organizational groups and the formal organizational structure as a means of communication and coordination [9][39]. The second relevant line of work is the research on dynamic analysis of social networks which provides an innovative approach, called the meta-matrix, to examine the dynamic co-evolution of relationships among multiple types of entities such as resources, tasks, and individuals [8][33]. Building on those two streams of research, we define socio-technical congruence as the match between the coordination requirements established by the dependencies among tasks and the actual coordination activities carried out by the engineers. In other words, the concept of congruence has two components. First, the coordination needs determined by the technical dimension of the socio-technical system and, secondly, the coordination activities carried out by the organization representing the social dimension. The following paragraphs discuss in detail the mathematical framework to measure the two components of congruence originally introduced by Cataldo and colleagues [10].

3.1 Identification of Coordination Requirements

In order to identify which set of individuals should be coordinating their activities, we need to represent two sets of relationships. One set is given by which individuals are working on which tasks. The relationships or dependencies among tasks represent the second element. In the rest of this section, we will use the example depicted in Figure 1 to describe the sets of relationships involved in determining coordination requirements. In the framework introduced by Cataldo and colleagues [10], assignments of individuals to particular work items is represented by a people by task matrix where a one in cell *ij* indicates that worker *i* is assigned to task *j*. We refer to such matrix as *Task Assignments* (T_A). In our example (figure 1), we can think of the matrix T_A as representing the set of files modified by each developer that worked on a modification request. The set of dependencies among tasks can also be represented as a square matrix where a cell *ij* (or cell *ji*) indicates that task *i* and task *j* are interdependent. We refer to such matrix as *Task Dependencies* (T_D). Figure 1 shows an example where T_D captures the syntactic dependencies among the source code files of a system. For instance, file 1 (row 1) has three dependencies (e.g. function calls) into file 3 (column 3). Multiplying the T_A matrix and the T_D matrix results in a people by task matrix that represents the extent to which a particular worker should be aware of tasks that are interdependent with those that he or she is responsible for. However, we are interested in a people to people relationship of coordination needs. Such a representation of the coordination requirements among the different workers is obtained by multiplying the $T_A * T_D$ product by the transpose of T_A. This product results in a people by people matrix where a cell *ij* (or cell *ji*) indicates the extent to which person *i* works on tasks that share dependencies with the tasks worked on by person *j*. In other words, the resulting matrix represents the *Coordination Requirements* (C_R) or the extent to which each pair of people needs to coordinate their work. In the context of the example depicted by figure 1, the *Coordination Requirements* matrix represents the extent to which the developers that work on a particular modification request need to coordinate their work given the set of syntactic relationships that exits among a system's modules. More formally, the C_R matrix is defined by the following product:

$$C_R = T_A * T_D * T_A^T \qquad \text{(Eq. 1)}$$

where, T_A is the Task Assignments matrix, T_D is the Task Dependencies matrix and T_A^T is the transpose of the Task Assignments matrix. This framework provides alternative ways of thinking about coordination requirements among workers depending on what type of data is used to populate the Task Dependencies

matrix. Past work had focused on temporal relationships between tasks, for instance, task A needs to be done before task B (e.g. [36]). In the context of software development, such way of thinking about task dependencies is quite common. Alternative views could be based on high level roles in the development organizations (e.g. integration and testing depends on development) or task dependencies based on product dependencies in the actual software code (e.g. function calls between modules). The focus in this paper is on the relationship between the structure of work dependencies and the structure of product dependencies because, as discussed earlier, the difficulty of identifying and managing certain types of product dependencies is a critical factor in coordination success and ultimately in productivity and quality.

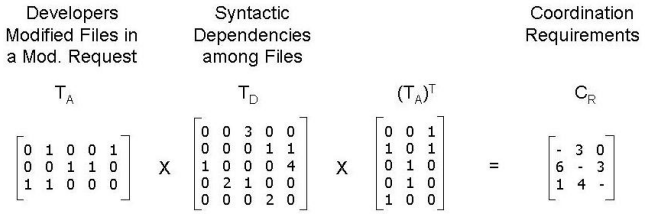

Figure 1: Example on Computing Coordination Requirements

3.2 Computing Congruence

Given a particular C_R matrix constructed from relating product dependencies to work dependencies, we can compare it to an *Actual Coordination* (C_A) matrix that represents the coordination activities of interactions software engineers. Then, congruence is defined as the proportion of coordination activities that actually occurred (represented in the C_A matrix) relative to the total number of coordination activities that should have taken place (represented by the C_R matrix). For instance, if the C_R matrix shows that 10 pairs should coordinate, and of these, 5 show coordination activities in the C_A matrix, then the congruence is 0.5. Formally, congruence is defined as follows:

$$Diff\ (C_R, C_A) = card\ \{\ diff_{ij} \mid cr_{ij} > 0\ \&\ ca_{ij} > 0\ \}$$

$$|C_R| = card\ \{\ cr_{ij} > 0\ \}$$

Then, we have:

$$Congruence\ (C_R, C_A) = Diff\ (C_R, C_A) / |C_R| \qquad \text{(Eq. 2)}$$

In sum, the value of congruence belongs to the [0,1] interval that represents the proportion of coordination requirements that were satisfied through some type of coordination activity or mechanism. This measure of socio-technical congruence provides a new way of thinking about coordination by providing a fine-grain level of analysis of different types of technical dependencies and allowing us to examine how coordination needs are impacted by them.

3.3 Two Approaches to Identify Technical Dependencies in Software Systems

The measure of congruence presented in the previous section relies on a representation of dependency that drives the engineers' coordination needs. In this section, we discuss two approaches to identify technical dependencies from a software system.

The traditional view of software dependency has its origins in compiler optimizations and they focus on control and dataflow relationships [30]. This approach extracts relational information between specific units of analysis such as statements, functions or methods, as well as modules, typically, from the source code of a system or from an intermediate representation of the software code such as bytecodes or abstract syntax trees. These relationships can represent either a data-related dependency (e.g. a particular data structure modified by a function and used in another function) or a functional dependency (e.g. method A calls method B). This type of dependency analysis technique has been widely used in a research context to examine the relationship between coupling and quality of a software system (e.g. [31][40]). Syntactic dependency analyses are also used by software developers to improve their understanding of programs and the linkages among the various parts of those programs [37].

One characteristic of these relational structures such as a call-graph, and for that matter other graphs such as inheritance and data dependencies graphs, is that they provide a particular view of the system-wide structure. Moreover, the accuracy of the information represented in these graphs depends on the ability of the tool used to identify all the appropriate types of syntactic relationships allowed by the underlying programming language [37].

An alternative mechanism of identifying dependencies consists of examining the set of source code files that are modified together as part of a modification request. This approach is equivalent to the approach proposed by Gall and colleagues [22] in the software evolution literature to identify logical dependencies between modules. A source code file can be viewed as representing a "bundle" of technical decisions. If a modification request can be implemented by changing only one file, it provides no evidence of any dependencies among files. However, when a modification request requires changes to more than one file, it can be assumed that decisions about the change to one file in a modification request depend in some way on the decisions made about changes to the other files involved in implementing the modification request. Dependencies could range from syntactic, for instance a function call between files, to more complex semantic dependencies where the computations done in one file affects the behavior of another file. This approach would represent a better estimate for semantic dependencies relative to call graphs or data graphs because it does not rely on language constructs to establish the dependency relationship between source code files. The remainder of this paper refers to this approach to identify dependencies as the "Files Changed Together" (FCT) method. We will refer to the method to identify dependencies based on syntactic functional and data relationships described earlier as the CGRAPH method.

4. METHOD

4.1 Description of the Data

In this study, we used data from the first four releases of a company's main product which was a large distributed system. Overall, the data covered a period of 39 months of development activity. A hundred and fourteen developers grouped into eight development teams distributed across three development locations worked full time on the project during the time period covered by the data. Software developers communicated and coordinated using various means. Opportunities for interaction existed when the developers worked in the same formal team or when they were located in the same development site. Developers also used tools such as Internet Relay Chat (IRC) and a MR tracking system

(similar to Bugzilla) to interact and coordinate their work. For instance, the MR tracking system kept track of the progress of the task, comments and observations made by developers as well as additional material used in the development process. We collected communication and coordination information from those two systems. Finally, we also collected demographic data about the developers such as their programming and domain experience and level of formal education.

The unit of analysis is the modification request which corresponds to a development work item associated with a defect or a new feature. A total of 2375 multi-team modification requests were identified. Those modification requests belonged to the first four releases of the product and involved more than one software development team. The decision to focus on such modification requests is based on a growing body of research which shows that difficulties in communication and coordination breakdowns are recurring problems in software development [15][28][34], particularly when the work items are geographically distributed [28] and the task involves more than one organizational team [15][20][34].

4.2 Descriptions of Measures

The literature has identified a number of factors that affect development time and, consequently, the resolution of modification requests. Some of those factors are related to characteristics of the task such as the amount of code to be written and the priority of the task, whereas other factors capture relevant attributes of the individual developers and the teams that participate in the development task. In the following paragraphs, we first describe our dependent variable, resolution time of modification requests. Secondly, the procedures used to construct the measures of congruence are described. Finally, we describe a number of control measures that were also included in the statistical models.

Productivity Measure: Our measure of development productivity is *Resolution Time* which is defined as the time, in days, it took to resolve a particular MR. We recognize that some modification requests may have longer resolution times because people are working on multiple MRs simultaneously, or because a MR was temporarily suspended to address other higher priority work. We addressed these concerns in two different ways. First, we collected several control variables that impact resolution time and they described later in this section. Secondly, our measure of productivity accounts only for the time that the MR was assigned to a particular developer. We were able to accurately determine such time periods because the company had a process in which modification requests were assigned to developers only when they were actively working on them. Otherwise, MRs would be assigned to a generic team identifier. Inspection of a random sample of modification requests suggested that this process rarely was not followed.

Congruence Measures: The data for building the *Coordination Requirements* matrix (equation 1) was extracted from several data sources such as the modification request reports, the version control system as well as the software code itself. A modification request provides the "developer *i* modified file *j*" relationship that constitutes our *Task Assignment* matrix. Since two different methods for identifying dependencies were used, FCT and CGRAPH, we constructed two different *Task Dependency* matrices. In the case of the FCT method, the cell c_{ij} of the *Task Dependency* matrix represents the number of times a particular pair of source code files changed together as part of the work associated with a modification request. A moving window of 19 months was used to capture a representative set of logical dependencies among the software modules. The resolution date of the modification request was paired with the end of the time window used to collect the task dependency information. In the case of the CGRAPH method, the cell c_{ij} of the *Task Dependency* matrix represents the number of data/function/method references from file *i* into file *j*. The syntactic relationships were extracted from the system's source code using the C-REX tool [26]. We constructed quarterly call-graphs of the entire system. The data from the quarter associated with the resolution date of the modification request was used to collect the task dependency information. Given the *Task Assignments* and *Task Dependencies* matrices just described, we computed as described in equation 1, two *Coordination Requirement* matrices, one based on the FCT and a second based on the CGRAPH method.

Software engineers can coordinate and exchange information through numerous communication means. Therefore, we constructed four *Actual Coordination* matrices which represent coordination activities that took place through different communication paths during the work associated with a modification request. *Structural Coordination* captures the potential communication and coordination activity that individuals that belong to the same organizational team might have through mechanisms such as periodic team meetings. We built the actual coordination matrix where a coordination activity between developers *i* and *j* exists if they belong to the same formal team. An extension of the structural coordination measure is to consider all the developers that are collocated in a same development facility as one group of individuals that have opportunities to communicate and coordinate given their physical proximity [1][38]. We refer to this measure as *Geographical coordination* and in terms of the matrix of coordination activities, engineers *i* and *j* have a linkage if they work in the same location.

The two *Actual Coordination* measures described so far are proxies for expected coordination within teams and within locations. We also constructed two additional measures that represent coordination activity that was measured more directly. *MR communication* considers an interaction between engineers *i* and *j* only when both *i* and *j* explicitly commented in the modification request report. Multiple modification requests might refer to the same problem and later be marked as duplicates of a particular modification request. All duplicates of the focal MR were also used to capture the interactions among developers. We focused on interactions among developers that explicitly commented on the MR report because the MR-tracking system notified through emails to all the individuals registered in a CC list every time an MR is updated. Therefore the recipients of updates could be significantly larger than the set of people actually providing information to the MR. In other words, our approach creates a network composed of only the engineers that contributed information to the MR report. Finally, *IRC communication* was computed based on interaction between developers from the IRC logs. Identifying which IRC messages related to which modification request posed a particular problem. The work on a MR could extend over days, weeks or even months. The IRC logs must be examined through out that period of time to identify interactions among engineers that are relevant to that MR. Furthermore, developers could refer to the MR id number (e.g. "<developer01> developer02: have you

looked at bug 12345") or to the problem the MR represents without any explicit reference to the MR (e.g. "<developer01> does anyone know why would RPC call 123 return the error code 12345?"). Three raters, blind to the research questions, were trained to examined the IRC logs corresponding to the period of time associated with each MR and to established an interaction between engineers *i* and *j* if they made reference to the bug ID or to the task or problem represented by the MR in their conversations. In order to assess the reliability of the raters' work, 10% of the MRs where coded by all raters. Comparisons of the obtained networks showed that 98.2% of the networks had the same set of nodes and edges. All four *Actual Coordination* matrices were symmetric.

Using the congruence computation described in *Eq. 2*, we constructed eight congruence measures, using all combinations of the two *Coordination Requirements* matrices (FCT and CGRAPH) and the four *Actual Coordination* matrices (Structural, Geographic, MR, and IRC). These congruence measures were used in the regression models described in sections 4.3 and 5.

Control Measures: As described in Cataldo et al [10], numerous factors impact the resolution time of modification requests. In this paper, we used the same set of control measures utilized in the baseline model reported by Cataldo et al [10]. The following paragraphs describe the details of those measures that capture attributes of the modification requests, the software engineers and the teams associated with the development work. Several task-specific factors such as the temporal dependency among MRs, task priority and task re-assignments could have an important effect on development time. *Temporal Dependency* was measured as the number of modification requests that the focal MR depends on in order for the task to be performed. Management prioritized the activities of the developer by using a scale from 1 to 5 in the modification request report where level 5 as the highest priority and level 1 as the lowest priority. This rating constituted our measure of *priority* of the MR. *Task re-assignment* was measured as the number of times an MR was re-assigned to a different engineer or team. Re-assignment impacts resolution time because each new developer needs to build up contextual information about the task. In addition, MRs opened by customers could represent work items with higher importance consequently affecting the resolution time. A dummy variable was used to indicate if the MR is associated with the service request from a customer. *Multiple Locations* is a binary variable that indicates whether the all the developers that worked on a particular MR were in the same geographical location (a value of 0) or were distributed across the development labs (a value of 1). Finally, the *release* variable identifies the release of the product that the modification request is associated with. This variable could also be considered as a proxy for time to control for efficiencies that might develop over time and, consequently, affect the resolution time of the modification requests. The *change size* measure is a proxy for the actual amount of development work done and it was computed as the number of files that were modified as part of the change for the focal MR. Although past work [20] has used lines of code changed as a measure of the size of the modification, our analysis of both measures showed equivalent results in the statistical models used in this study.

It is well established that experience, along different dimensions such as tools, programming languages and product domain, is critical to software development productivity [5][14][15]. We constructed several experience measures using archival information. *Programming experience* was computed as the average number of years of programming experience prior to joining the company of all the engineers involved in the modification request. *Tenure* was measured as the average number of months in the company of all the engineers that worked in the modification request at the time the work associated with the MR was completed. *Component experience* was computed as the average number of times that the engineers responsible for the modification request have worked on the same files affected by the focal modification request. This measure was also log-transformed to satisfy normality requirements. Finally, *Team load* is a measure of the average work load of the teams responsible for the components associated with the modification request. This control variable was computed as the ratio of the average number of modification requests in *open* or *assigned* state over the total number of engineers in the groups involved in the focal modification request during the period of time the MR was in *assigned* state.

4.3 Description of the Model

Past research has found that linear [20][29] and hierarchical linear [20][34] models are appropriate techniques for examining the effects of different factors on development productivity. In this study, we examined the effect of congruence on resolution time using the following linear regression model:

$$ResolutionTime = \sum_i \beta_i * CongruenceMeasure_i + \sum_j \delta_j * ControlVariable_j + \varepsilon$$

An examination of descriptive statistics and Q-Q plot indicated that several of the variables (*Resolution Time*, *Change Size* and *Component Experience*) were highly skewed to the left. The log transformation provided the best approximation to a normal distribution. The analysis of the pair-wise correlations amongst the variables in the model suggested no relevant collinearity problems. Only a small set of correlations were statistically significant but their levels did not exceed +/- 0.343.

The measures of structural and geographical congruence could be affected by personnel turnover and mobility across teams. We examined archival data collected from the company and we determined a yearly turnover rate of only 3% and an inter-group mobility rate of less than 1%. The modification requests that involved individuals that left the company or changed group membership were eliminated from the analysis. However, an analysis including those modification requests showed results consistent with those reported in section 5.

5. RESULTS

We performed several linear regression analyses to assess the effect of the congruence measures on the resolution time of modification requests. As discussed in section 4, two different methods, FCT and CGRAPH, were used to identifying technical dependencies which resulted in two sets of congruence measures. We first discuss the results of the analyses done using the congruence measures based on the FCT method. Table 1 shows the results from the OLS regressions. Model I is a baseline regression model which only considers the control factors. Consistent with

previous empirical work in software engineering, factors such as the size of the modification to the code, familiarity with the software components, and general programming experience are significant elements that affect resolution time of MRs [20][28]. Task-specific characteristics such as temporal dependencies with other modification requests and the priority of the task are associated with an increase in development time. As it has been reported in previous research [20][28], the results also show that when developers are geographically distributed, the amount of time required to resolve modification requests is likely to increase. The coefficients from model I also suggest that time, captured by the variable *Release*, had no statistically significant effect. Since the *Release* measure is in fact a categorical variable, we also examined its impact using two dichotomous variables to represent the four possible values. The results were identical to defining *Release* as an integer from 1 to 4 to represent the four releases of the product.

Table 1: Effects on Resolution Time (FCT method)

	Model I	Model II	Model III
(Intercept)	4.81**	4.63**	4.48**
Temporal Dependency	0.59**	0.59**	0.59**
Priority	-0.40**	-0.41**	-0.40**
Re-assignment	0.01	0.01	0.01
Customer MR	0.09	0.10	0.09
Release	-0.02	-0.02	-0.03
Change Size (log)	0.31**	0.31**	0.31**
Team Load	-0.01	-0.01	-0.01
Multiple Locations	0.13**	0.13**	0.13**
Programming Experience	-0.17**	-0.17**	-0.17**
Tenure	-0.01^{+}	-0.01^{+}	-0.01^{+}
Component Experience (log)	-0.07**	-0.07**	-0.07**
Structural Congruence		-0.18*	-0.14*
Geographical Congruence		-0.02*	-0.04*
MR Congruence		-0.06*	-0.05*
IRC Congruence		-0.21*	-0.21*
Multiple Locations X MR Congruence			0.13
Multiple Locations X IRC Congruence			-0.27*
N	2375	2375	2375
Adjusted R^2	0.718	0.819	0.831

($^{+}$p < 0.10, *p < 0.05, **p < 0.01)

Model II introduces the measures of congruence into the analysis. The results show statistically significant effects on all the congruence measures computed using the FCT method. The estimated coefficients of the congruence measures have negative values which are associated with a reduction in resolution time. The results highlight the important role of congruence on task performance as well as the complementary nature of all communication paths. Structural congruence is associated with shorter development times suggesting that when coordination requirements are contained within a formal team and appropriate communication paths exists, task performance increases. Geographical congruence had a positive effect on resolution time, consistent with past research that argued distance has detrimental effects on communication (see [28] and [38] for reviews). Communication congruence based on the interactions amongst engineers through the MR reports as well as IRC were also statistically significant suggesting the usefulness of these tools in facilitating coordination among individuals that belong to different teams and could potentially be geographically distributed.

Finally, model III includes several interaction factors to assess whether the role of congruence changes when the groups involved in a particular MR are geographically distributed. The results show a statistically significance impact only for the Multiple Locations X IRC term. The negative coefficient suggests that when developers are geographically distributed the impact of IRC congruence on resolution time is higher above and beyond the direct effect.

Table 2 shows the results of our analysis obtained when the congruence measures are computed using the CGRAPH method for identifying technical dependencies. Model I is the same model reported in table 1. We observe in model IV that only geographical congruence is statistically significant and its coefficient is negative indicating a reduction in the resolution time as congruence increases. Structural congruence was marginally significant. Finally, Model V shows that interaction terms were not statistically significant. In sum, these results suggest that the two dependency identification methods, FCT and CGRAPH, are capturing different sets of technical dependencies that impact the development tasks differently.

Table 2: Effects on Resolution Time (CGRAPH method)

	Model I	Model IV	Model V
(Intercept)	4.81**	4.88**	4.81**
Temporal Dependency	0.59**	0.59**	0.59**
Priority	-0.40**	-0.40**	-0.40**
Re-assignment	0.01	0.03	0.01
Customer MR	0.09	0.19	0.09
Release	-0.02	-0.02	-0.02
Change Size (log)	0.31**	0.31**	0.31**
Team Load	-0.01	-0.01	-0.01
Multiple Locations	0.13**	0.12**	0.13**
Programming Experience	-0.17**	-0.17**	-0.17**
Tenure	-0.01^{+}	-0.01^{+}	-0.01^{+}
Component Experience (log)	-0.07**	-0.07**	-0.07**
Structural Congruence		-0.21^{+}	-0.23^{+}
Geographical Congruence		-0.11*	-0.03*
MR Congruence		0.41	0.48
IRC Congruence		-0.01	-0.02
Multiple Locations X MR Congruence			0.05
Multiple Locations X IRC Congruence			-0.41
N	2375	2375	2375
Adjusted R^2	0.718	0.731	0.722

($^{+}$p < 0.10, *p < 0.05, **p < 0.01)

6. DISCUSSION

This study has significant contributions to the software engineering and management of product development organizations literatures. First, the empirical evaluation of the congruence framework showed the importance of understanding the dynamic nature of software development. Identifying the "right" set of product de-

pendencies that determine the relevant work dependencies and coordinating accordingly has significant impact on reducing the resolution time of modification requests. The analyses showed traditional software dependencies, such as syntactic relationships, tend to capture a relatively narrow view of product dependencies that is not fully representative of the important product dependencies that drive the need to coordinate. On the other hand, logical dependencies provide a more accurate representation of the most relevant product dependencies in software development projects. The statistical analyses showed that when developers' coordination patterns are congruent with their coordination needs, the resolution time of modification requests was, on average, reduced by 32% when considering the collective effect of all four measures of congruence. Generalizing, the empirical examination of the congruence framework and coordination patterns showed the tight relationship between team design, coordination and performance providing an important contribution to the organizational literature.

The congruence framework extends traditional conceptualizations of coordination by taking a fine-grain level of analysis to better examine the mismatches between dependencies and coordination activities. Those gaps could have major implications for the productivity and the quality of the output of product development organizations [15][20][28][42]. Our empirical results suggest that our measure of socio-technical congruence represents a useful framework to examine how coordination needs that are not satisfied impact software development productivity. When the developers coordinate their task with the relevant set of workers, productivity increases. Individuals have difficulties identifying task interdependencies that are not obvious or explicit [42] and the developers' ability to recognize dependencies diminish as coordination requirements change over time [21]. For these reasons, changes in the coordination requirements represent an important obstacle for product development organizations, particularly, when work groups are geographically distributed. Collaborative tools and managerial techniques that utilize the congruence framework could play an important role in reducing the gap between recognized and actual interdependencies.

7. LIMITATIONS

It is also important to highlight some of the limitations of the work reported in this paper. First, the measures proposed by the congruence framework are contingent on assumptions about the software development processes used in the development organization as well as usage patterns of tools that assist the development effort such as defect tracking and version control systems. One key assumption is the possibility to identify (1) the set of source code files that were changed as part of a modification request and (2) the developers that made those changes. For instance, a policy of source code file ownership by particular developers could potentially bias the congruence measures. Developers that own a particular source code might appear as participants in the development effort associated with a modification request, however, that might not be the case. In other cases, such as open source projects, the nature of the work in certain project is such that the information about which files changed together as part of a MR is not easily reconstructible in a reliable way.

The alternative approach of computing coordination requirements based on syntactic relationships also has its limitations. The method relies on tools that can reliably extract the dependency information among software modules for a specific programming language. More importantly, projects that use multiple programming languages will represent a challenge, particularly, in terms of determining syntactic dependencies that involve modules written in different programming languages.

Another limitation of the work presented in this paper is a potential concern for external validity. Our analysis examined only one system with particular technical properties that might be conducive to support the results found by the analysis. However, the processes and tools used by the development organization are commonplace in the software industry. Moreover, the general technical characteristics of the system are similar to other types of distributed systems developed into products in the software industry. Hence, we think the results are generalizable, particularly, in the context of development organizations responsible for delivering complex software systems. Finally, the study does not consider all forms of coordination, such as telephone and e-mail communication. Data on such communication were not available.

8. FUTURE WORK

8.1 Enhancing coordination needs awareness

Collaboration, coordination, and task awareness tools are a natural application for the coordination requirements measure presented in this paper. Part of the research effort of the CSCW community has been on improving traditional tools, such as email and instant messaging, which have become an integral part of work in the vast majority of organizations [3][45]. For instance, the coordination requirements measure could provide a way of identifying the email exchanges that are more relevant given the task interdependencies among individuals. This information would enable tools to present an enhanced task management experience by, for instance, prioritizing to-do lists and generating reminders to respond to task-specific emails based on the coordination requirements. This email sorting approach could be thought as a task-specific alternative to other social-based sorting techniques such as the one proposed by Fisher and colleagues [21]. A more recent set of tools, such as sidebars [7] and productivity assistants [24], would also benefit from the congruence framework. These types of tools focus on activity-centric collaboration and, as argued by Geyer and colleagues [24], the majority of the tools assume user intervention in terms of deciding what type of information to make part of the sidebar. The congruence framework would provide an automatic mechanism to identify people of interest giving a particular set of task dependencies among the workers.

In the context of large software development projects, identifying the appropriate person to interact with and coordinate interdependent activities is not a straightforward task. In fact, it is well established that software developers have serious difficulties identifying the right set of individuals to coordinate with [17][25]. The coordination requirement measure provides a mechanism to augment awareness tools that provide real-time information regarding the likely set of workers that a particular individual might need to communicate with. For instance, integrated development environments, such as Eclipse [18] or Jazz [32], could use the coordination requirement information to recommend a dynamic "coordination buddy list" every time particular parts of the software are modified. In this way, the developer becomes aware of the set of engineers that modified parts of the system that are interdependent with the one the developer is working on. The concept of the

"buddy list" in communication and collaboration tools is not a new idea. However, the novel contribution is to construct the "buddy list" from accurate estimates of the set of individuals more likely to be relevant to a particular developer in relations to the work dependencies, information which is captured by the coordination requirements measure.

8.2 Identification of coordination requirements in early stages of software projects

The empirical examination of the congruence framework showed the relevance of matching coordination activity with the fine-grained coordination needs that emerge in the development of software systems. However, the measure of congruence, as computed in the study, relies on archival data to capture the information about product dependencies, task assignments as well as coordination activity carried out by the development organization. Our promising results highlight the importance of identifying potential coordination needs as early as possible in the development process in order to provide the development organization with the appropriate communication and coordination mechanisms. Certainly such a task is a challenging one.

In early stages of a project, only architectural or high level design specifications of a system are available. Those documents by definition abstract a significant portion of the technical details of software systems in order to understand the overall attributes and relationships among the main components of a system. A higher level of abstraction could potentially hinder the identification of relevant technical dependencies and consequently, important coordination requirements. However, the use of standardized design and modeling languages, such as UML, might represent a way of overcoming these challenges. Researchers have proposed standard graphical representations of software architectures that capture different technical aspects of a software system [12]. Examples of those graphical representations are the *module* view and the *components-and-connectors* view.

We envision a *coordination* view of the architecture that combines the product's technical dependencies with relationships among the organizational units responsible for carrying out the development work. In order to generate such representations, methods of identifying relevant dependencies from the technically focused views of the architecture are to be devised. One potentially promising approach is to synthesize the dependencies represented in the various types of UML diagrams (e.g. class diagrams, sequence diagrams, etc.) into a single set of technical relationships among modules. Such a method could be able to identify logical relationships among parts of the systems which, as shown in this paper, are an important factor driving the work dependencies in software development organizations.

9. ACKNOWLEDGMENTS

We gratefully acknowledge support by the National Science Foundation under Grants No. IIS-0414698, IIS-0534656 and IGERT 9972762; the Software Industry Center at Carnegie Mellon University and its sponsors, especially the Alfred P. Sloan Foundation; and a Software Engineering Institute IRD grant.

10. REFERENCES

[1] Allen, T.J. 1977. Managing the Flow of Technology. MIT Press.

[2] Baldwin, C.Y. and Clark, K.B. 2000. Design Rules: The Power of Modularity. MIT Press.

[3] Bellotti, V. et al. 2003. Taking email to task: the design and evaluation of a task management centered email tool. In Proceedings International Conference on Human Factors in Computing Systems (CHI'03), Ft. Lauderdale, FL.

[4] Bass, M., Bass, L., Herbsleb, J.D. and Cataldo, M. 2006. Architectural Misalignment: an Experience Report. To appear in the Proceedings of the 6th International Conference on Software Architectures (WICSA '07).

[5] Brooks, F. 1995. The Mythical Man-Month: Essays on Software Engineering. Addison Wesley.

[6] Burton, R.M. and Obel, B. 1998. Strategic Organizational Diagnosis and Design. Kluwer Academic Publishers.

[7] Cadiz, J.J. et al. 2002. Designing and deploying an information awareness interface. In Proceedings of the Conference on Computer Supported Cooperative Work (CSCW'02), New York, NY.

[8] Carley, K.M. 2002. Smart Agents and Organizations of the Future. In Handbook of New Media. Edited by Lievrouw, L. and Livingstone, S., Sage, Thousand Oaks, CA.

[9] Carley, K.M and Ren, Y. 2001. Tradeoffs between Performance and Adaptability for C^3I Architectures. In Proceedings of the 6th International Command and Control Research and Technology Symposium, Annapolis, Maryland.

[10] Cataldo, M. et al. Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools. In Proceedings of the Conference on Computer Supported Cooperative Work (CSCW'06), Banff, Alberta, Canada.

[11] Cataldo, M. et al. 2007. On Coordination Mechanism in Global Software Development. In Proceedings of the International Conference on Global Software Engineering (ICGSE'07), Munich, Germany.

[12] Clements, P. et al. 2002. Documenting Software Architectures: Views and Beyond. Addison-Wesley.

[13] Conway, M.E. 1968. How do committees invent? Datamation, 14, 5, 28-31.

[14] Curtis, B. 1981. Human Factors in Software Development. Ed. by Curtis, B., IEEE Computer Society.

[15] Curtis, B., Kransner, H. and Iscoe, N. 1988. A field study of software design process for large systems. Communications of ACM, 31, 11, 1268-1287.

[16] de Souza, C.R.B. 2005. On the Relationship between Software Dependencies and Coordination: Field Studies and Tool Support. Ph.D. dissertation, Donald Bren School of Information and Computer Sciences, University of California, Irvine.

[17] de Souza, C.R.B. et al. 2004. How a Good Software Practice Thwarts Collaboration – The multiple roles of APIs in Software Development. In Proceedings of the 12th Conference on Foundations of Software Engineering (FSE '04), Newport Beach, CA.

[18] Eclipse Project. 2008. http://www.eclipse.org. URL accessed on February 28th, 2008.

[19] Eppinger, S.D. et al. A Model-Based Method for Organizing Tasks in Product Development. Research in Eng, Design, 6, 1-13.

[20] Espinosa, J.A. 2002. Shared Mental Models and Coordination in Large-Scale, Distributed Software Development. Unpublished Ph.D. Dissertation, Graduate School of Industrial Administration, Carnegie Mellon University.

[21] Fisher, D., Brush, A.J., Gleave, E. and Smith M.A. 2006. Revisiting Whittaker and Sidner's "Email Overload": Ten Years Later. In Proceedings of the Conference on Computer Supported Cooperative Work (CSCW'06), Banff, Alberta, Canada.

[22] Gall, H. Hajek, K. and Jazayeri, M. 1998. Detection of Logical Coupling Based on Product Release History. In Proceedings of the International Conference on Software Maintenance (ICSM '98), Bethesda, Maryland.

[23] Garcia, A., et al. 2007. Assessment of Contemporary Modularization Techniques, ACOM'07 Workshop Report. ACM SIGSOFT Software Engineering Notes, 35, 5, 31-37.

[24] Geyer, W. et al. 2007. Malibu Personal Productivity Assistant. In Proceedings International Conference on Human Factors in Computing Systems (CHI'07) – Work in Progress Section, San Jose, CA.

[25] Grinter, R.E., Herbsleb, J.D. and Perry, D.E. 1999. The Geography of Coordination Dealing with Distance in R&D Work. In Proceedings of the Conference on Supporting Group Work (GROUP'99), Phoenix, Arizona.

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

[27] Henderson, R.M. and Clarck, K.B. 1990. Architectural Innovation: The Reconfiguration of Existing Product Technologies and the Failure of Established Firms. Adm. Sci. Quarterly, 35, 9-30.

[28] Herbsleb, J.D. and Mockus, A. 2003. An Empirical Study of

[29] Speed and Communication in Globally Distributed Software Development. IEEE Trans. on Software Engineering, 29, 6.

[30] Herbsleb, J.D., Mockus, A. and Roberts, J.A. 2006. Collaboration in Software Engineering Projects: A Theory of Coordination. In Proceedings of the International Conference on Information Systems (ICIS'06), Milwaukee, Wisconsin.

[31] Horwitz, S., Reps, T., and Binkley, D. 1990. Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems, 22, 1, 26-60.

[32] Hutchens, D.H. and Basili, V.R. 1985. System Structure Analysis: Clustering with Data Bindings. IEEE Transactions on Software Engineering, 11, 8, 749-757.

[33] Jazz Project. 2008. http://jazz.net/pub/index.jsp. URL accessed on February 28th, 2008.

[34] Krackhardt, D. and Carley, K.M. 1998. A PCANS Model of Structure in Organization. In Proceedings of the 1998 International Symposium on Command and Control Research and Technology.

[35] Kraut, R.E. and Streeter, L.A. 1995. Coordination in Software Development. Communications of ACM, 38, 3, 69-81.

[36] Leffingwell, D. and Widrig, D. 2003. Managing Software Requirements: A Use Case Approach, 2nd Edition. Addison-Wesley.

[37] Levchuk, G.M. et al. 2004. Normative Design of Project-Based Organizations – Part III: Modeling Congruent, Robust and Adaptive Organizations. IEEE Trans. on Systems, Man & Cybernetics, 34, 3, 337-350.

[38] Murphy, G.C. et al. 1998. An empirical study of call graph extractors. ACM Trans. on Software Engineering Methodology, 7, 2, 158-191.

[39] Olson, G.M. and Olson, J.S. 2000. Distance Matters. Human-Computer Interaction, 15, 2 & 3, 139-178,

[40] Parnas, D.L. (1972). On the criteria to be used in decomposing systems into modules. Communications of ACM, 15, 12, 1053-1058.

[41] Selby, R.W. and Basili, V.R. (1991). Analyzing Error-Prone System Structure. IEEE Transactions on Software Engineering, 17, 2, 141-152.

[42] Simon, H.A. 1962. The Architecture of Complexity. In Proceedings of the American Philosophical Society, 106, 6, 467-482.

[43] Sosa, M.E., Eppinger, S.D., and Rowles, C.M. 2004. The Misalignment of Product Architecture and Organizational Structure in Complex Product Development. Management Science, 50, 12, 1674-1689

[44] Sullivan, K.J. et al. 2001. The Structure and Value of Modularity in Software Design. In Proceedings of the International Conference on Foundations of Software Engineering (FSE '01), Vienna, Austria.

[45] Von Hippel, E. 1990. Task Partitioning: An Innovation Process Variable. Research Policy, 19, 407-418.

[46] Wattenberg, M. et al. 2005. E-Mail Research: Targeting the Enterprise. J. of Human-Computer Interaction, 20, 139-162.

[47] Yassine, A. et al. 2003. Information Hiding in Product Development: The Design Churn Effect. Research in Eng. Design, 14, 145-161.