Software Dependencies, Work Dependencies, and Their Impact on Failures

Marcelo Cataldo, Audris Mockus, Jeffrey A. Roberts, and James D. Herbsleb

Abstract—Prior research has shown that customer-reported software faults are often the result of violated dependencies that are not recognized by developers implementing software. Many types of dependencies and corresponding measures have been proposed to help address this problem. The objective of this research is to compare the relative performance of several of these dependency measures as they relate to customer-reported defects. Our analysis is based on data collected from two projects from two independent companies. Combined, our data set encompasses eight years of development activity involving 154 developers. The principal contribution of this study is the examination of the relative impact that syntactic, logical, and work dependencies have on the failure proneness of a software system. While all dependencies increase the fault proneness, the logical dependencies explained most of the variance in fault proneness, while workflow dependencies had more impact than syntactic dependencies. These results suggest that practices such as rearchitecting, guided by the network structure of logical dependencies, hold promise for reducing defects.

Index Terms—Distribution/maintenance/enhancement, metrics/measurement, organizational management and coordination, quality analysis and evaluation.

1 Introduction

It has long been established that many software faults are caused by violated dependencies that are not recognized by developers designing and implementing a software system [12], [26]. The failure to recognize these dependencies could stem from technical properties of the dependencies themselves as well as from the way development work is organized. In other words, two dimensions are at play—technical and organizational.

On the technical side, the software engineering literature has long recognized call and dataflow syntactic relationships as an important source of error [4], [29], [40]. Research in the software evolution literature has introduced a new view on technical dependencies among software modules. Gall et al. [21] introduced the idea of "logical" coupling (or dependencies) by showing that source-code files that are changed together can uncover dependencies among those files that are not explicitly identified by traditional syntactic approaches. Past work has also examined aspects of the relationship between logical dependencies and failures in software systems. Eick et al. [15] used increases of such logical coupling as an indicator of "code decay." Graves et al. [23] showed that past changes are good predictors of future faults, and Mockus and Weiss [32] found that the spread of a change over subsystems and files is a strong indicator that the change will contain a defect.

Human and organizational factors can also strongly affect how dependencies are handled, potentially affecting the quality of a software system. Research has shown that the level of interdependency between tasks tends to increase the level of communication and coordination activities among workers [20], [46]. Recent studies suggest, however, that the identification and management of technical dependencies is a challenge in software development organizations, particularly when those dependencies are semantic rather than syntactic [7], [12], [24], [27]. Appropriate levels of communication and coordination may not occur, potentially decreasing the quality of a system [11], [26]. Consequently, it is important to understand how work dependencies (i.e., the way dependencies are manifested in development tasks) impact failure proneness.

In contrast with research on fault prediction models [35], [36], [48], our work focuses on evaluating several potential causes of defects, rather than formulating a predictive model. The principal contribution of this study is the examination of the *relative* impact that syntactic, logical, and work dependencies have on the failure proneness of software systems. While all of these factors are shown to be related to failures, the strength of the relationships varies dramatically. Understanding the relative impact is critical for determining where to focus research, tools, and process improvement. In addition, we also sought to improve the external validity of the study by replicating the analysis over multiple releases of two distinct projects from two unrelated companies.

The remainder of the paper is organized as follows: The next two sections elaborate on how syntactic, logical, and work-related dependencies relate to a software system's failure proneness. Sections 4, 5, and 6 describe the study

- *M. Cataldo is with the Research and Technology Center, Robert Bosch LLC, 2 North Shore Center, #320 Pittsburgh, PA 15212. E-mail: marcelo.cataldo@us.bosch.com.*
- *A. Mockus is with Avaya Labs Research, 233 Mt. Airy Road, Basking Ridge, NJ 07920. E-mail: audris@avaya.com.*
- *J.A. Roberts is with the Palumbo Donahue School of Business, Duquesne University, 600 Forbes Avenue, Pittsburgh, PA 15282. E-mail: robertsj503@duq.edu.*
- *J.D. Herbsleb is with the School of Computer Science, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213. E-mail: jdh@cs.cmu.edu.*

Manuscript received 3 Nov. 2008; revised 20 Apr. 2009; accepted 22 Apr. 2009; published online 9 July 2009.
Recommended for acceptance by P. Devanbu.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2008-11-0361.
Digital Object Identifier no. 10.1109/TSE.2009.42.

methodology, preliminary analyses, and the results, respectively. We conclude the paper with a discussion of the contributions, limitations, and future work.

2 Software Dependencies and Failure Proneness

The traditional syntactic view of software dependency had its origins in compiler optimizations, and focused on control and dataflow relationships [28]. This approach extracts relational information between specific units of analysis such as statements, functions or methods, and source-code files. Dependencies are discovered, typically, by analysis of source code or from an intermediate representation such as bytecodes or abstract syntax trees. These relationships can be represented either by a data-related dependency (e.g., a particular data structure modified by a function and used in another function) or by a functional dependency (e.g., method A calls method B).

The work by Hutchens and Basili [29] and Selby and Basili [40] represents the first use of dependency data in the context of a system's propensity for failure. Building on the concepts of coupling and cohesion proposed by Stevens et al. [43], Hutchens and Basili [29] presented metrics to assess the structure of a system in terms of data and functional relationships, which were called bindings. The authors used clustering methods to evaluate the modularization of a particular system. Selby and Basili [40] used the data binding measure to relate system structure to errors and failures. They found that routines and subsystems with lower coupling were less likely to exhibit defects than those with higher levels of coupling. Similar results have been reported in object-oriented systems. Chidamber and Kemerer [9] proposed a set of measures that captures different aspects of the system of relationships between classes. Briand et al. [4] found that the measures of coupling proposed by Chidamber and Kemerer were positively associated with failure proneness of classes of objects.

More recently, models focused on the prediction of failure proneness have been explored using various concepts to organize (or group) software artifacts into various units of analysis. These organizing concepts include architectural, graph-theoretic, and "concerns" perspectives. Measures such as network, syntactic dependency, and complexity metrics are used to explore the association between the artifact groups and postrelease defects. Eaddy et al. [14] explored defects using concerns (i.e., features or requirements) to organize software artifacts for analysis. Here, the authors found that dispersion of a concern's implementation ("scatter") was associated with software defects. Nagappan and Ball [35] explored software failures using two architectural levels within Microsoft Windows to establish their unit of analysis. The authors found that syntactic dependencies and source-code change metrics ("churn") calculated within and between components (binaries or DLLs) and higher level application areas (e.g., the Internet Explorer area) were predictive of postrelease failures. Zimmerman and Nagappan [48] applied a graph theoretic lens to classify and calculate network measures for Windows binaries. In this work, the authors demonstrated that orthogonal linear combinations of network, syntactic dependency, and complexity metrics could be used to predict postrelease defects.

In contrast to the previously discussed research, an alternative view of dependency has been developed in the software evolution literature. This approach focuses on deducing dependencies between the source-code files of a system that are changed together as part of the software development effort and it was first discussed in the literature as "logical coupling" by Gall et al. [21]. Unlike traditional syntactic dependencies, this approach identifies indirect or semantic relationships between files that are not explicitly deducible from the programming language constructs [21]. There are several cases where logical dependencies provide more valuable information than syntactic dependencies. Remote procedure calls (RPCs) represent a simple example. Although the syntactic dependency approach would provide the necessary information to relate a pair of modules, such information would be embedded in a long path of connections from the RPC caller through the RPC stubs all the way to the RPC server module. On the other hand, when the module invoking the RPC and the module implementing the RPC server are changed together, a logical dependency is created, showing a direct dependency between the affected source-code files. The logical dependency approach is even more valuable in cases such as publisher-subscriber or event-based systems where the call-graph approach would fail to relate the interdependent modules since no syntactically visible dependency would exist between, for instance, a module that generates an event and a module that registers to receive such an event.

Not only does the logical dependency approach have the potential to identify important dependencies not visible in syntactic code analyses, it may also filter out syntactic dependencies that are unlikely to lead to failures. For example, in the case of basic libraries (e.g., memory management, printing functionality, etc.) the syntactic dependencies approach would highlight these highly coupled files. Yet, they tend to be very stable and unlikely to fail despite a high level of coupling. The logical dependency approach eliminates these problems as the likelihood of change in files that implement these basic functions is very low, hence, a logical dependency would not be established.

It is difficult to know if the logical dependency approach actually realizes these potential advantages. Only limited work has focused on the relationship between logical dependencies and failure proneness of a system. Mockus and Weiss [32] found that in a large switching software system, the number of subsystems modified by a change is an excellent predictor of whether the change contains a fault. Nagappan and Ball [35] found that architecturally based logical coupling metrics are correlated with post-release failure proneness of programs. However, the authors computed metrics at the level of component and program areas, a coarse-grain approach resulting in measures too highly correlated to allow the authors to assess each metric's relative impact on failure proneness.

In sum, the extant research exploring the relationship between failure proneness of software with regard to dependencies has focused on a single dependency type (syntactic or logical) and has not examined the relative contribution of each of these types. One implication of this limitation is that decisions regarding the focus of quality improvement efforts may be misplaced. Additionally, research in this area has examined only a single project limiting the external validity of results. This leads to our first research question:

RQ 1. *What is the relative impact of syntactic and logical dependencies on the failure proneness of a software system?*

3 Work Dependencies and Failure Proneness

The literature on failure proneness has only recently begun to look at the impact of human and organizational factors on the quality of such systems. The work on coordination in software development suggests that identification and management of work dependencies is a major challenge in software development organizations [12], [24], [27]. Modularization is the traditional approach used to cope with dependencies in product development. In software engineering, Parnas [37] was the first to articulate the idea of modular software design introducing the concept of information hiding. Parnas argued that modules be considered work items, not just a collection of subprograms. The idea being that development on one module can proceed independently of the development of another. Baldwin and Clark [2], in the product development literature, argued that modularization makes complexity manageable, enables parallel work, and tolerates uncertainty. Like Parnas, Baldwin and Clark argued that a modular design structure leads to an equivalent modular work structure.

The modularization argument assumes a simple and obvious relationship between product modularization and task modularization—reducing the technical interdependencies among modules also reduces the interdependencies among the tasks involved in producing those modules. In addition, the modular design approach assumes that reducing dependencies reduces the need for work groups to communicate. Unfortunately, there are several problems with these assumptions. Recent empirical evidence indicates that the relationship between product structure and task structure is not as simple as previously assumed [6]. Moreover, promoting minimal communication between teams responsible for related modules is problematic because it significantly increases the likelihood of integration problems [13], [24]. Herbsleb et al. [26] theorized that the irreducible interdependence among software development tasks can be thought of as a distributed constrain satisfaction problem (DSCP) where coordination is a solution to the DSCP. Within that framework, the authors argued that the patterns of task interdependence among the developers as well as the density of the dependencies in the constraint landscape are important factors affecting coordination success and, by extension, the quality of a software system and the productivity of the software development organization.

More recently, Nagappan et al. [36], Pinzger et al. [38], and Mockus and Weiss [32] investigated a series of organizational metrics as predictors of failure proneness in Windows components and other software. All of the above studies share important limitations with respect to understanding the impact of organizational and social factors in failure proneness. First, they focus on failure prediction models and contain no analysis of the relative importance of the measures in predicting software defects. Furthermore, the proposed measures do not specifically capture work dependencies per se but rather they are proxies for numerous phenomena not necessarily related to the issue of work dependencies. For instance, the measure "number of unique engineers who have touched a binary" in [36, p. 524] could be capturing different sources of failures such as difficulties stemming from disparities in engineers' experience and organizational processes rather than capturing issues of coordination [36]. In sum, there is a need to better understand how the quality of a software system is affected by the ability of the developers to identify and manage work dependencies. This leads to our second research question:

RQ 2. *Do higher levels of work dependencies lead to higher levels of failure proneness of a software system?*

4 Methods

We examined our research questions using two large software development projects. One project was a complex distributed system produced by a company operating in the computer storage industry. The data covered a period of approximately three years of development activity and the first four releases of the product. The company had 114 developers grouped into eight development teams distributed across three development locations. All of the developers worked full time on the project during the time period covered by our data. The system was composed of approximately 5 million lines of code distributed in 7,737 source-code files in C language with a small portion, of 117 files, in C++ language.

The second project was an embedded software system for a communications device developed by a major telecommunications company. Forty developers participated in the project over a period of five years covering six releases of the product. All but one developer worked in the same location. The system had more than 1.2 million lines of C and C++ code in 1,224 files with 427 files written using in C++. We will refer to the distributed system as "project A" and to the embedded system as "project B."

In both development organizations, every change to the source code was controlled by modification requests. A modification request (MR) is a development task that represents a conceptual change to the software that involves modifications to one or more source-code files by one or more developers [33]. The changes could represent the development of new functionality or the resolution of a defect encountered by a developer, the quality assurance organization, or reported by a customer. We refer to the latter type of defects as "field" defects. A similar process was associated with each modification request in both projects. Upon creation, the MR is in *new* state; it is then assigned to a particular development team by a group of managers performing the role of a change control board. Commits to the version control systems were not allowed without modification request identifier. This characteristic of the process allowed the organizations to have a reliable mechanism of associating the modification request reports with the actual changes to the software code. As soon as all of the changes associated with a modification request are completed, the MR is set to *review required* state and a reviewer is assigned. Once the review is passed and the changes are integrated and tested, the modification request is set to *closed* state. In project A, we collected data corresponding to a total of 8,257 resolved MRs belonging to the first four releases of the product. We collected the data associated with more than 3,372 MRs in project B. In the remainder of this section, we describe the measures and the statistical models used in this research.

4.1 Descriptions of the Data and Measures

We used three main sources of data in both projects A and B. First, the MR-tracking system data were used to collect the modification requests included in our analysis. Second, the version control systems provided the data that captured the changes made to the system's source code. Finally, the source code itself. Using the above data sources, we constructed our dependent and independent measures that are described in the following paragraphs.

4.1.1 Measuring Failure

We chose to investigate failure proneness at the file level. Our dependent variable, *File Buggyness*, is a binary measure indicating whether a file has been modified in the course of resolving a field defect. For each file, we determined if it was associated with a field defect in any release of the product covered by our data. We used the logistic regression model shown in (1) in order to model the binary dependent variable and assess the effect of syntactic, logical, and work dependencies.

$$\begin{aligned} FileBuggyness = &\sum_i \beta_i * SyntacticDependenciesMeasure_i + \\ &\sum_j \chi_j * LogicalDependenciesMeasure_j + \\ &\sum_n \delta_n * WorkDependenciesMeasure_n + \\ &\sum_k \varphi_k * AdditionalMeasure_k + \varepsilon. \end{aligned} \quad (1)$$

4.1.2 Syntactic Dependencies

We obtained syntactic dependency information using a modified version of the C-REX tool [25] to identify programming language tokens and references in each entity of each source-code file.[1] For all revisions of both systems, a separate syntactic dependency analysis was performed for a snapshot of all source code associated with that revision. Each source-code snapshot was created at the end of the quarter in which the release took place. Using the resulting data, we computed syntactic dependencies between source-code files by identifying data, function, and method references crossing the boundary of each source-code file. Let D_{ij} represent the number of data/function/method references that exist from file i to file j. We refer to data references as *data dependencies* and function/method references as *functional dependencies*.

Arguably, data and functional syntactic dependencies could impact failure proneness differently. Functional dependencies provide explicit information about the relationship between a caller and a callee. On the other hand, data relationships are not quite as obvious, particularly in terms of understanding the modification sequences of data objects such as global variables. Such understanding typically requires the usage of a tool such as a debugger. Consequently, we collected four syntactic dependencies measures: inflow and outflow data relationships and inflow and outflow functional dependencies. Each of those four measures capture the number of syntactic dependencies of such type exhibited by each file i.

4.1.3 Logical Dependencies

Logical dependencies relate source-code files that are modified together as part of an MR. If an MR can be implemented by changing only one file, it provides no evidence of any dependencies among files. However, when an MR requires changes to more than one file, we assume that decisions about the change to one file depend in some way on the decisions made about changes to the other files involved in the MR. The concept of logical dependencies is equivalent to Gall et al.'s [21] idea of logical coupling.

In both projects, modification requests contained information about the commits made in the version control system. As described earlier, such information was reliably generated as part of the submission procedures established in the development organizations. Such data allowed us to identify the relationship between development tasks and the changes in the source code associated with such tasks. Using this information, we constructed a logical dependency matrix. The logical dependency matrix is a symmetric matrix of source-code files where C_{ij} represents the sum, across all releases, of the number of times files i and j were changed together as part of an MR. We accumulate the data across releases as files that are changed together in an MR provide mounting evidence of the existence of a logical dependency. The longer the period of time considered, the more changes take place, increasing accuracy of the identified logical dependencies.

Although the association between MRs and changes in the code was enforced by processes and tools, there are other sources of potential errors that might impact the quality of the data represented in the logical dependency matrix. For instance, a developer could commit a single change to two files where one contained a fix to one MR and the second file had an unrelated change to a second MR. We performed a number of analyses to assess the quality of our MR-related data and minimize measurement error. We compared the revisions of the changes associated with the modification requests and we did not find evidence of such type of behavior. We also grouped version control commits that might have been associated with modification requests that were marked as duplicates under a single MR. Finally, we examined random samples of modification requests to determine if developers have work patterns that could impact the quality of our data such as the example described above. For instance, during the data collection

[1] We were not able to utilize common object-oriented coupling measures as both systems are predominantly written using the C programming language.

process of project A, one of the authors and a senior developer from the project examined a random sample of 90 modification requests. None of the commits contained changes to the code that were not associated with the task represented in the modification requests.

Two file-level measures were extracted from the logical dependency matrix—*Number of Logical Dependencies* and *Clustering of Logical Dependencies*. The *Number of Logical Dependencies* measure for file i was computed as the number of nonzero cells on column i of the matrix.[2] Since the logical dependencies matrix is symmetric, this measure is equivalent to the degree of a node in undirected graph, excluding self-loops. The difference in the nature of the technical dependencies captured by the syntactic and logical approaches is evidenced by the limited overlap between those two types of dependencies. In project A, 74.3 percent of the syntactic dependencies were not identified as logical relationships between a pair of source of files while in project B such difference was 97.3 percent.

Herbsleb et al. [26] argued that the density of dependencies increases the likelihood of coordination breakdowns. Building on that argument, we constructed a second measure from the logical dependency matrix that we called *Clustering of Logical Dependencies*. Unlike the *Number of Logical Dependencies*, this measure captures the degree to which the files that have logical dependencies to the focal file have logical interdependencies among themselves. Formally and in graph theoretic terms, the *Clustering of Logical Dependencies* measure for file i is computed as the density of connections among the direct neighbors of file i. This measure is equivalent to Watts's [47] local clustering measure and it is mathematically represented by (2) where k_i is the number of files or "neighbors" that a particular file i is connected to through logical dependencies and e_{jk} is a link between files j and k which are neighbors of file i. The values of this measure range from 0 to 1.

$$CLD(f_i) = \frac{2|\{e_{jk}\}|}{k_i(k_i - 1)}. \tag{2}$$

4.1.4 Work Dependencies

We constructed two different measures of work dependencies—*Workflow Dependencies* and *Coordination Requirements. Workflow Dependencies* capture the temporal aspects of the development effort while *Coordination Requirements* capture the intradeveloper coordination requirements.

Workflow Dependencies. As described previously, both projects used MR-tracking systems to assess the progress of development tasks. Each modification request followed a set of states from creation until closure. Those transitions represent an MR workflow where particular members of the development organization had work-related responsibilities associated with such MR at some point in time during its lifecycle. Such a workflow constitutes the traditional view of work dependencies were individuals are sequentially interdependent on a temporal basis [45]. More specifically, two developers i and j are said to be interdependent if the MR was transferred from developer i to developer j at some point between the creation and closure of the MR. For instance, suppose an MR requires changes to two subsystems, with the changes to the second relying on changes to the first. Developer i completes the work on subsystem one and then he/she transfers the development task to developer j to finish the work on the subsystem two.

Grouping the workflow information of all of the MRs associated with a particular release of the products, we constructed a developer-to-developer matrix where a cell c_{ij} represents the number of work dependencies developer i has on developer j. The information in such a matrix captures the web of workflow-related dependencies in which each developer was embedded during a particular release of the product. Such developer-to-developer relationships can be examined through the lenses of social network analysis which provides the relevant theoretical background and methodological framework [30], [46]. A traditional result in the social network literature is that individuals centrally located in the network (i.e., have, on average, a larger number of relationships to other individuals) tend to be more influential because they control the flow of information [5], [30]. On the negative side, a high number of linkages requires a significant effort on the part of those individuals in order to maintain the relationships [5], [30]. This latter point is particularly important in the context of the workflow dependencies because it argues that centrally located developers are more likely to be overloaded because of the effort associated with managing the work dependencies, increasing the likelihood for communication break downs and thus the quality of software produced could be expected to diminish.

Degree centrality [19] is a traditional measure used in the social network literature to identify central individuals based on the number of ties to other actors in the network. Formally, degree centrality is defined as $DC(n_i, M) = d(n_i)$, where $d(n_i)$ is the number of connections of node n_i in matrix M. The values of this measure range from 0 to $n - 1$, where 0 indicates the node is an isolate (i.e., not connected to any other node) and $n - 1$ indicates that the node i has ties to all other $n - 1$ nodes. Building on the theoretical argument outlined in the previous paragraph and on the concept of degree centrality, the *Workflow Dependencies* measure was constructed as follows: For each file i, we identified the developer j that worked on the file and was linked to the greatest number of individuals in the developer-to-developer workflow network for each release. That is, the developer exhibiting the highest degree centrality. As discussed earlier, such individuals are the more likely to introduce an error due to higher levels of effort they face in managing a higher number of work dependencies. Equation (3) formally describes the *Workflow Dependencies* measure. We also considered the average of the number of linked developers over the set of developers that worked on each file. However, this measure was highly correlated with our other independent measures and thus excluded from further analysis.

$$\begin{aligned} WD(f_i) = \max\{&DC(dev_j, WD)| \\ &j \in \{developers\ that\ changed\ f_i\}\}. \end{aligned} \tag{3}$$

Coordination Requirements. Workflow dependencies relate developers through the temporal evolution of

[2] The diagonal of the matrix indicates the number of times a single file was modified and can be disregarded from further analysis.

modification requests and the developers' involvement in those MR. There are additional work-related dependencies that emerge as development work is done in different parts of a system. For instance, two developers could work on two different modification requests involving files that are syntactically or logically interdependent. In this case, modifications made by each developer could impact the other's work. These types of work-related dependencies are more subtle in nature and require more effort on the part of the developers to identify and manage. Cataldo et al. [6] proposed a framework for examining the relationship between the technical dependencies of a software system and the structure of the development work to construct such system. Coordination requirements, an outcome of that framework, represent a developer-by-developer matrix (C_R) where each cell $C_{R\,ij}$ represents the extent to which developer i needs to coordinate with developer j given the assignments of development tasks and technical dependencies of the software system. More formally, Cataldo et al. [6] defined the C_R matrix as follows:

$$C_R = T_A\,^{*}T_D\,^{*}T_A^T, \tag{4}$$

where $\mathrm{T_A}$ is the *Task Assignments* matrix, $\mathrm{T_D}$ is the *Task Dependencies* matrix, and $\mathrm{T_A^T}$ is the transpose of the *Task Assignments* matrix. In the context of our study, the $\mathrm{T_A}$ and $\mathrm{T_D}$ matrices were constructed using data from the MR reports and the version control system in the following way: An MR report provides the "developer i modified file j" relationship. We grouped such information across all modification requests in a particular release to construct the *Task Assignment* matrix which is a developer-to-file matrix. The *Task Dependency* matrix was a file-to-file matrix and it was constructed using the same approach described in the computation of the logical dependencies measures. In other words, each 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 the MRs. Following the theoretical argument and the process presented in the previous section (description of workflow dependencies), the *Coordination Requirements* measure captures for each file i, the degree centrality of the most central developer in the C_R matrix (a developer-to-developer matrix) that worked on the file i. Equation (5) formally describes the *Coordination Requirements* measure.

$$\begin{aligned} CR(f_i) = \max\{&DC(dev_j, CR)| \\ &j \in \{developers\ that\ changed\ f_i\}\}. \end{aligned} \tag{5}$$

4.1.5 Additional Control Factors

The objective of this study is to examine the relative impact that important conceptual factors such as technical and work dependencies have on failure. In order to account for the effects of potentially confounding influences, however, our analysis must include factors that past research has found to be associated with failures. Numerous measures have been used to predict failures [14], [18], [23], [35], [36], [48]. As suggested by Graves et al. [23], such measures can be classified as either process or product measures. Process measures such as number of changes, number of deltas, and age of the code (i.e., churn metrics) have been shown to be very good predictors of failures [23], [35]. Accordingly, we control for the *Number of MRs*, which is the number of times the file was changed as part of a past defect or feature development. We also control for the *Average Number of Lines Changed* in a file as part of MRs.

In contrast, product measures such as code size and complexity measures have produced somewhat contradictory results as predictors of software failures. Some researchers have found a positive relationship between lines of code and failures [4], [23], while others have found a negative relationship [3]. Our collective experience regarding the relationship between product measures and software defects has been that such measures are associated with increased software failure. Thus, we expect that product measures will be positively associated with software defects. We measure the size of the file (*LOC*) as the number of nonblank noncomment lines of code.

5 Preliminary Analysis

Our four dependency measures (syntactic, logical, workflow, and coordination requirements) capture different characteristics of the technical and work-related dependencies that emerge in the development of software systems. Table 1 presents a comparative summary of our dependency measures. Syntactic and Workflow dependencies are explicit in nature and are therefore easier to identify and manage by developers or other relevant stakeholders in software development projects. On the other hand, the Logical and Coordination Requirement dependency measures capture less explicit, more subtle relationships among software artifacts and developers, respectively. The implicit nature of those dependencies makes identification and management of such relationship more challenging. In sum, our measures assess explicit and implicit dependencies that emerge in the technical and work-related dimensions of software projects.

Table 2 summarizes the descriptive statistics of all the measures described in the previous sections. Due to a moderate degree of skewness, we applied a log transformation to each of the independent variables. Table 3 reports the pairwise correlations of all our measures. Overall, the pairwise correlations are relatively similar across projects indicating that the phenomena reflected by these measures may be common in both projects. There are, however, several high correlations that deserve attention. For instance, the *Number of MRs* (past changes) variable is highly correlated with *LOC*, *Average Lines Changed*, and our measure of logical dependencies, particularly in project B. In addition, the syntactic dependencies measures are also highly correlated among themselves and with other measures such as *LOC* and *Number of MRs*. We computed variance inflation factors (VIFs) and tolerances to further examine potential issues due to multicollinearity among our independent variables. A tolerance close to 1 indicates little multicollinearity, whereas a value close to 0 suggests that multicollinearity may be a significant threat. VIF is defined as the reciprocal of the tolerance.

Table 4 reports the variance inflation factor and tolerance associated with each of our measures. We start our multicollinearity diagnostic with model I that contains all our independent measures. We observe that, for both

TABLE 1
Comparative Summary of Dependency Measures

	Dimension	**Identifiability**	**Manageability**
Syntactic Dependencies	Technical	Captures explicit relationships between source code files.	A host of tools can aid developers in the management of this type of dependencies.
Logical Dependencies	Technical	Captures semantic or implicit relationships between source code files, in addition to some explicit relationships.	Dependence on historical data, attributes of the tools (e.g. version control system) and consistent processes over time limits the developers' ability to manage this type of dependency.
Workflow Dependencies	Work / Social	Captures explicit relationships among project members based on workflows and/or processes	Traditional tools (e.g. ClearQuest or Bugzilla) facilitate significantly the management of these dependencies.
Coordination Requirement Dependencies.	Work / Social	Captures less explicit relationships among project members based on their past contributions to the development effort and the technical dependencies of the system under development.	Dependence on historical data, attributes of the tools (e.g. version control system) and consistent processes over time limit the developers' ability to manage this type of dependency.

projects A and B, the measures *Out-Data Syntactic Dependencies* and *Out-Functional Syntactic Dependencies* have a VIF significantly higher (or a tolerance significantly lower) than the other measures. We removed those two variables and the recomputed VIF and tolerances values for the remaining measures are reported in model II in Table 4. We observe that *Number of MRs* has a lower tolerance than the rest of the measures, particularly in project B's data. Consequently, we removed it and the resulting VIFs and tolerances are reported in model III. In this case, the data for project A do not show signs of multicollinearity, with the tolerances of all measures above 0.70.

On the other hand, in project B, the low tolerance values for the two measures of work dependencies suggest some potential multicollinearity problems. Removing the *Coordination Requirement Dependencies* measure from model III results in an improvement of the VIF associated with Workflow dependencies down to 1.20 ($\text{tolerance} = 0.8304$). In addition, the tolerances of all remaining variables increased with the minimum value being 0.7028 for the

TABLE 2
Descriptive Statistics

Project A: Distributed System						
	Mean	**SD**	**Min**	**Max**	**Skew**	**Kurtosis**
File Buggyness	0.49	0.500	0	1	0.011	1.001
LOC	481.9	836.1	0	17853	4.931	47.24
Avg. Lines Changed	10.85	32.67	0	738	8.512	108.9
In-Data Syntactic Dep.	4.57	58.94	0	1741	24.40	647.6
Out-Data Syntactic Dep.	8.90	9.243	0	53	0.792	3.050
In-Functional Syntactic Dep.	20.36	71.49	0	951	5.701	42.78
Out-Functional Syntactic Dep.	25.96	68.42	0	543	5.241	32.57
Num. Logical Dep.	87.27	99.54	0	836	1.856	7.584
Clustering Logical Dep.	0.72	0.316	0	1	-1.024	3.011
Workflow Dep.	22.53	12.76	0	44	-0.013	1.878
Coordination Req.	0.14	0.121	0	0.62	2.655	11.91
Project B: Embedded System						
	Mean	**SD**	**Min**	**Max**	**Skew**	**Kurtosis**
File Buggyness	0.14	0.35	0	1	2.026	5.105
LOC	750.8	2874.3	0	65542	18.24	389.6
Avg. Lines Changed	19.18	52.53	0	987	9.617	135.7
In-Data Syntactic Dep.	10.61	85.60	0	1805	16.18	287.1
Out-Data Syntactic Dep.	7.85	14.41	0	173	207.9	27.07
In-Functional Syntactic Dep.	9.17	29.09	0	612	11.11	180.4
Out-Functional Syntactic Dep.	15.84	29.08	0	238	3.396	18.01
Num. Logical Dep.	38.61	41.61	0	370	3.152	18.61
Clustering Logical Dep.	0.52	0.19	0	0.69	-1.241	4.010
Workflow Dep.	28.41	15.60	1	72	0.253	2.461
Coordination Req.	0.85	0.14	0	1	-2.956	15.29

TABLE 3
Pairwise Correlations ($*p < 0.01$) for Last Release in Each Data Set

Project A: Distributed System						
	1	**2**	**3**	**4**	**5**	**6**
1.FileBugyness	-					
2.LOC (log)	0.28*	-				
3.Number MRs (log)	0.37*	0.24*	-			
4.Avg. Lines Changed (log)	0.18*	0.27*	0.30*	-		
5.In-Data Dep. (log)	0.06*	0.01	0.08*	0.03	-	
6.Out-Data Dep. (log)	0.18*	0.47*	0.19*	0.19*	-0.26*	-
7.In-Functional Dep. (log)	0.04*	0.27*	0.09*	0.09*	-0.10*	0.37*
8.Out-Functional Dep. (log)	0.11*	0.43*	0.15*	0.16*	-0.24*	0.78*
9.Num Logical Dep. (log)	0.49*	0.33*	0.45*	0.16*	0.04*	0.23*
10.Clustering Logical Dep. (log)	-0.32*	-0.21*	-0.29*	-0.13*	-0.06*	-0.17*
11.Workflow Dep. (log)	0.33*	0.06*	0.33*	0.12*	0.02	0.07*
12.Coordination Req. Dep. (log)	0.04*	-0.06*	-0.15*	-0.06*	-0.01	-0.03
	7	**8**	**9**	**10**	**11**	**12**
8.Out-Functional Dep. (log)	0.44*	-				
9.Num Logical Dep. (log)	0.06*	0.19*	-			
10.Clustering Logical Dep. (log)	-0.10*	-0.14*	-0.05*	-		
11.Workflow Dep. (log)	-0.07*	-0.03	0.31*	-0.12*	-	
12.Coordination Req. Dep. (log)	-0.07*	-0.05*	0.02	0.12*	0.15*	-
Project B: Embedded System						
	1	**2**	**3**	**4**	**5**	**6**
1.FileBugyness	-					
2.LOC (log)	0.28*	-				
3.Number MRs (log)	0.55*	0.41*	-			
4.Avg. Lines Changed (log)	0.19*	0.42*	0.35*	-		
5.In-Data Dep. (log)	0.22*	0.33*	0.26*	0.19*	-	
6.Out-Data Dep. (log)	0.26*	0.60*	0.34*	0.35*	0.49*	-
7.In-Functional Dep. (log)	0.19*	0.36*	0.25*	0.19*	0.47*	0.54*
8.Out-Functional Dep. (log)	0.28*	0.59*	0.38*	0.39*	0.43*	0.88*
9.Num Logical Dep. (log)	0.29*	0.26*	0.62*	0.25*	0.13*	0.20*
10.Clustering Logical Dep. (log)	-0.28*	-0.15*	-0.34*	-0.10*	-0.17*	-0.21*
11.Workflow Dep. (log)	0.26*	0.09*	0.38*	0.01	0.19*	0.10*
12.Coordination Req. Dep. (log)	0.17*	-0.03	0.26*	-0.05	0.14*	0.02
	7	**8**	**9**	**10**	**11**	**12**
8.Out-Functional Dep. (log)	0.52*	-				
9.Num Logical Dep. (log)	0.12*	0.22*	-			
10.Clustering Logical Dep. (log)	-0.19*	-0.20*	0.17*	-		
11.Workflow Dep. (log)	0.08	0.10*	0.29*	-0.18*	-	
12.Coordination Req. Dep. (log)	0.07	0.04	0.24*	-0.12*	0.75*	-

LOC measure. In Section 6, we revisit this issue when discussing the results from our regression analyses.

6 Results

We approached the analysis in two stages. In the first stage, we focused on examining the relative impact of each dependency type on failure proneness of source-code files. The data corresponding to the last release from each project were used in this analysis. In the second stage, we verified the consistency of the initial results by conducting a number of confirmatory analyses for each project. These analyses included reestimating our logistic regression models for each release as well as estimating a single longitudinal model comprising all releases. The detailed results of each stage are discussed in turn.

6.1 The Impact of Dependencies

We constructed several logistic regression models to examine the relative impact of each class of independent variable on the failure proneness of a software system using the data from the last release of each project. Following a

TABLE 4
Collinearity Diagnostics

Project A: Distributed System			
	Model I **VIF (Tolerance)**	**Model II** **VIF (Tolerance)**	**Model III** **VIF (Tolerance)**
Number of MRs (log)	1.59 (0.6289)	1.59 (0.6297)	---
LOC (log)	1.53 (0.6530)	1.32 (0.7564)	1.32 (0.7564)
Avg. Lines Changed (log)	1.16 (0.8596)	1.16 (0.8625)	1.11 (0.9035)
In-Data Dep. (log)	1.13 (0.8867)	1.02 (0.9793)	1.02 (0.9825)
Out-Data Dep. (log)	2.85 (0.3503)	---	---
In-Functional Dep. (log)	1.26 (0.7916)	1.11 (0.9007)	1.11 (0.9031)
Out-Functional Dep. (log)	2.79 (0.3587)	---	---
Num Logical Dep. (log)	1.47 (0.6825)	1.45 (0.6880)	1.26 (0.7950)
Clustering Logical Dep. (log)	1.16 (0.8584)	1.16 (0.8628)	1.09 (0.9152)
Workflow Dep. (log)	1.26 (0.7921)	1.24 (0.8040)	1.18 (0.8487)
Coordination Req. Dep. (log)	1.09 (0.9213)	1.08 (0.9218)	1.05 (0.9523)
Project B: Embedded System			
	Model I **VIF (Tolerance)**	**Model II** **VIF (Tolerance)**	**Model III** **VIF (Tolerance)**
Number of MRs (log)	2.82 (0.3547)	2.80 (0.3573)	---
LOC (log)	1.83 (0.5467)	1.49 (0.6689)	1.45 (0.6897)
Avg. Lines Changed (log)	1.34 (0.7469)	1.30 (0.7687)	1.28 (0.7826)
In-Data Dep. (log)	1.47 (0.6787)	1.38 (0.7244)	1.38 (0.7260)
Out-Data Dep. (log)	4.91 (0.2038)	---	---
In-Functional Dep. (log)	1.58 (0.6344)	1.39 (0.7181)	1.39 (0.7184)
Out-Functional Dep. (log)	4.75 (0.2105)	---	---
Num Logical Dep. (log)	2.32 (0.4316)	2.31 (0.4321)	1.33 (0.7528)
Clustering Logical Dep. (log)	1.61 (0.6223)	1.60 (0.6251)	1.19 (0.8435)
Workflow Dep. (log)	2.56 (0.3913)	2.55 (0.3927)	2.50 (0.4003)
Coordination Req. Dep. (log)	2.38 (0.4201)	2.37 (0.4228)	2.36 (0.4230)

standard hierarchical modeling approach, we started our analysis with a baseline model that contains only the traditional predictors. In subsequent models, we added the measures for syntactic, logical, and work dependencies described in the previous sections. We assessed the goodness of fit of the model to evaluate the impact of each class of dependency measures on failure. For each statistical model, we report the χ^2 of the model, the percentage of deviance explained by the model as well as the statistical significance of the difference between a model that adds new factors and the previous model without the new measures. Deviance is defined as -2 times the log likelihood of the model. The percentage of the deviance explained is a ratio of the deviance of the null model (containing only the intercept) and the deviance of the final model. Model parameters were estimated, as is customary in logistic regression, using a maximum-likelihood method. In order to simplify the interpretation of the results, we report the odds ratios associated with each measure instead of reporting the regression coefficients. Odds ratios larger than 1 indicate a positive relationship between the independent and dependent variables whereas an odds ratio less than 1 indicates a negative relationship. For example, an odds ratio of two for a binary factor doubles the probability of a file having a customer-reported defect when the remaining factors in the model are at their lowest values. The presented odds ratio is the exponent of the logistic regression coefficient.

Tables 5 and 6 report the odds ratios of the various logistic regression models using the data from project A and project B, respectively. In both tables, model I includes the *LOC* and *Avg. Lines Changed* measures. As discussed in Section 5, the *Number of MRs* measure (a proxy for past changes) was not included in the analyses due to multicollinearity concerns. Model I, in Tables 5 and 6, shows that *LOC* is positively associated with failure proneness. These results agree with those found by Briand et al. [4], in contrast with earlier findings [3], [34]. *Avg. Lines Changed* is also positively related to failure proneness in both projects, indicating that the more modifications to a file, the higher the likelihood of encountering a field defect associated with that file. Specifically, a unit change in the log-transformed *Avg. Lines Changed* measure (or a change from 1 to 2.7 lines per MR in untransformed units) increases the odds of a field defect by 20 percent for project A (Table 5—Model I) and 25 percent in the case of project B (Table 6—Model I).

Model II introduces the syntactic dependency measures *Inflow Data* and *Inflow Functional.* The results of the logistic regression show that the impact of data syntactic dependencies are only marginally significant, which can be seen more clearly as the other factors are included in the regression model (see models III-V in Tables 5 and 6). In the case of project A, data syntactic dependencies are statistically significant across the various models and with the expected direction in their impact on failure proneness. On the other hand, the impact of the functional syntactic dependencies measure, unexpectedly, has the opposite direction. However, once the models include logical and work dependencies, the functional syntactic dependency measure no longer has statistical significance, indicating

TABLE 5
Odds Ratios from Logistic Regression on Project A (Distributed System) Data

	Model I	Model II	Model III	Model IV	Model V
LOC (log)	1.392**	1.418**	1.119**	1.142**	1.150**
Avg. Lines Changed (log)	1.203**	1.200**	1.138**	1.114**	1.126**
In-Data Dep. (log)		1.166**	1.103*	1.105*	1.112*
In-Functional Dep. (log)		0.949*	0.953+	0.982	0.989
Num Logical Dep. (log)			2.277**	2.079**	2.108**
Clustering Logical Dep. (log)			0.009**	0.012**	0.009**
Workflow Dep. (log)				2.011**	1.905**
Coordination Req. Dep. (log)					2.801**
Model χ2 (p-value)	388.87 (p < 0.01)	412.21 (p < 0.01)	1621.31 (p < 0.01)	1737.52 (p < 0.01)	1763.18 (p < 0.01)
Deviance Explained	7.1%	7.5%	29.5%	31.6%	32.1%
Model Comparison χ2 (p-value)	--	23.34 (p < 0.01)	1209.10 (p < 0.01)	116.21 (p < 0.01)	25.67 (p < 0.01)

(+ p < 0.10; * p < 0.05; ** p < 0.01)

that this type of syntactic relationship does not impact failure proneness. This latter pattern is also reflected in the data for project B, where both syntactic dependency measures become irrelevant once the logical and work dependency measures enter the models (see Table 6—models III-V). Given the limited impact of the syntactic dependencies on failure proneness it is not surprising to see a relatively modest improvement in the explanatory power of model II over model I (e.g., in project A, deviance improves from 7.1 percent to 7.5 percent). We do note, however, that, while improvement in the explanatory power is modest, the addition of the syntactic dependency measures does provide a statistically significant improvement in model fit as indicated by the model comparison χ^2 (project A: 23.34—$p < 0.01$; project B: 14.41—$p < 0.01$).

Model III also considers the logical dependency measures. As Tables 5 and 6 show, the odds ratios associated with each of the logical dependency measures in the logistic regression are greater than one, indicating that higher numbers of logical dependencies are related to an increase in the likelihood of failure. In particular, a unit increase in the log-transformed *Number of Logical Dependencies* measure, increases the odds of a failure 2.272 times higher for project A (Table 5—Model III) and 2.277 times higher for project B (Table 6—Model III). The analyses reported in Section 5 showed relatively low levels of correlation between syntactic and logical dependency measures. Thus, the results reported in Tables 5 and 6 suggest the effect of logical dependencies on failure proneness is complementary and significantly more important than the impact of syntactic dependencies. In addition, the levels of explained deviance for model III in both projects clearly shows that the contribution of the logical dependencies measures to the explanatory power of the model is much higher than the impact of the syntactic dependencies measure.

The results reported in Model III in Tables 5 and 6 also indicate that increases in the C*lustering of Logical Dependencies* significantly reduce the likelihood of failures. This

TABLE 6
Odds Ratios from Logistic Regression on Project B (Embedded System) Data

	Model I	Model II	Model III	Model IV	Model V
LOC (log)	1.800**	1.638**	1.497**	1.493**	1.499**
Avg. Lines Changed (log)	1.247**	1.253**	1.115	1.178	1.184
In-Data Dep. (log)		1.207*	1.124	1.046	1.142
In-Functional Dep. (log)		1.131	1.013	1.002	0.996
Num Logical Dep. (log)			2.303**	1.822**	1.803**
Clustering Logical Dep. (log)			0.005**	0.013**	0.014**
Workflow Dep. (log)				6.527**	4.899**
Coordination Req. Dep. (log)					37.616
Model χ2 (p-value)	86.01 (p < 0.01)	100.42 (p < 0.01)	218.13 (p < 0.01)	239.27 (p < 0.01)	240.02 (p < 0.01)
Deviance Explained	11.8%	13.8%	30.1%	32.9%	33.0%
Model Comparison χ2 (p-value)	--	14.41 (p < 0.01)	117.71 (p < 0.01)	21.14 (p < 0.01)	0.75 (p=0.387)

(+ p < 0.10; * p < 0.05; ** p < 0.01)

TABLE 7
Impact of Dependencies across Releases in Project A

	Release 1	**Release 2**	**Release 3**
LOC (log)	1.211**	1.087**	1.201**
Avg. Lines Changed (log)	1.122**	1.083*	1.048
In-Data Dep. (log)	1.243**	1.207*	1.125*
In-Functional Dep. (log)	0.985	1.041	1.013
Num Logical Dep. (log)	1.411**	1.949**	1.806**
Clustering Logical Dep. (log)	0.064**	0.023**	0.017**
Workflow Dep. (log)	1.287**	0.850**	1.448**
Coordination Req. Dep. (log)	1.007	10.852**	3.901**
Model $\chi 2$ (p-value)	514.53 (p < 0.01)	821.61 (p < 0.01)	1121.96 (p < 0.01)
Deviance Explained	13.7%	19.1%	22.2%

(+ p < 0.10; * p < 0.05; ** p < 0.01)

result may suggest that the clustering is a symptom of good, consistent modular design. Alternatively, it may be that as clusters of consistently interrelated files emerge, developers become more cognizant of such relationships and know where to look to make sure that changes to one part of the system do not introduce problems elsewhere.

In both Tables 5 and 6, model IV includes the first of our work dependency measures—workflow dependencies. The results are consistent across both projects. Higher numbers of workflow dependencies increase the likelihood that source-code files contain field defects. In particular, a unit increase in the log-transformed number of *Workflow Dependencies* measure increases the odds of a failure 2.011 times higher for project A (Table 5—Model IV) and 6.527 times higher for project B (Table 6—Model IV). Model V shows the impact of the second work dependency measure—coordination requirements. In project A, the impact of the *Coordination Requirement* measure is statistically significant and with an odds ratio of 2.801, its impact is higher than the impact of the Workflow Dependencies. On the other hand, in project B, its effect is not statistically significant. As discussed in Section 5, there is high collinearity between the two work dependency measures in project B's data (Table 3: correlation is 0.75; Table 4: VIFs > 2); consequently, the regression results were expected.

In this paper, we set out to examine the relative impact of syntactic, logical, and work-related classes of dependencies on failure proneness. The results presented in this section showed that all types of dependencies affect failures in a software system. More importantly, their role is complementary, suggesting the various types of dependencies capture different relevant aspects of the technical properties of a software system as well as elements of the software development process. Logical and work dependencies have a significantly higher impact on failure proneness as their associated odds ratios indicate. For instance, a unit increase in the log-transformed measures of *Number of Logical Dependencies* and *Workflow dependencies* increases the odds of postrelease defects two times more than syntactic dependencies in the case of project A and two times and six times, respectively, for the case of project B.

6.2 Stability Analysis

In the previous section, we showed that the different types of dependencies affected failure proneness in the last release of each project. It is also critical to examine whether our results are robust across the various releases of the products covered by our data. Accordingly, we ran the same logistic regression models on the data from the first three releases from project A and the additional five releases from project B. Table 7 reports the odd ratios for all the measures from the logistic regression using the data from project A. Table 8 reports the odd ratios for the measures from the logistic regression using the data from project B. As discussed in the previous section, we did not include the *Coordination Requirement Dependencies* measures in the analysis of project B because of the high correlation of that measure with the *Workflow Dependencies* measure. We observe that the results are mostly consistent with those reported in the previous section for both project A and B. However, there is one exception. The results for the measure of *Workflow Dependencies* are not consistent across releases in the data from project A. One possible explanation is the changing nature of the development work associated with each release. For instance, release 1 of project A was in fact the first release of the product. The development effort associated with subsequent releases involved an increasing amount of work related to fixing defects reported against previous releases and a decreasing amount of development effort on new features. In the case of project B, the impact of the *Workflow Dependencies* measure is consistent across all five releases. However, the coefficient for release 1 is not statistically significant.

The results reported in Tables 7 and 8 showed overall consistent effects of our predictors across the different releases covered by our data. However, the development effort associated with each release might have a temporal relationship. For instance, the technical or work dependencies from release 2 could influence the measures from release 3. More formally, the various measures associated with each of the releases could exhibit autocorrelation. Therefore, we ran an additional confirmatory analysis using a longitudinal (random effects) model that considers the

TABLE 8
Impact of Dependencies across Releases in Project B

	Release 1	Release 2	Release 3	Release 4	Release 5
LOC (log)	1.642*	1.823*	1.713*	1.447**	1.477**
Avg. Lines Changed (log)	0.984	0.816	0.892	1.116	1.171
In-Data Dep. (log)	1.126	0.905	0.948	0.981	1.057
In-Functional Dep. (log)	0.619	1.153	0.978	1.016	1.001
Num Logical Dep. (log)	3.964**	3.187**	2.166**	1.771**	1.865**
Clustering Logical Dep. (log)	0.001**	0.007**	0.008**	0.012**	0.013**
Workflow Dep. (log)	1.101	1.870*	1.711*	3.936**	3.904**
Coordination Req. Dep. (log)	---	---	---	---	---
Model $\chi 2$ (p-value)	103.44 (p < 0.01)	150.09 (p < 0.01)	159.31 (p < 0.01)	201.63 (p < 0.01)	213.99 (p < 0.01)
Deviance Explained	42.1%	40.5%	29.4%	30.6%	30.8%

(+ p < 0.10; * p < 0.05; ** p < 0.01)

data from all releases in each project simultaneously. Using this procedure, we accounted for any potential temporal factors that might affect the estimation of the coefficients that represent the impact of our measures on failure proneness. Overall, the results of the random effects model were consistent with those reported in Tables 5, 6, 7, and 8.

7 Discussion

The observed relative contributions of different types of dependencies on failure proneness in two unrelated projects have consequences of both theoretical and practical interest. All three types of dependencies are relevant and their impact is complementary, showing their independent and important role in the development process. These results suggest that quality improvement efforts could be tailored to ameliorate the negative effects of particular types of dependencies with emphasis on areas that have the largest impact on project quality.

Past research [4], [29], [40] has shown that source-code files with a higher number of syntactic dependencies were more prone to failure. Our analyses indicate that such impact is limited. On the other hand, our results suggest logical dependencies and work dependencies are significantly more important factors impacting the likelihood of source-code files to exhibit field defects. In addition, this study is the first analysis that highlights the importance of the structure of the logical relationships—source-code files with logical dependencies to other files that are also highly interdependent among themselves were less likely to exhibit customer-reported defects. We can view these groups of files as a unit where the structure of the technical dependencies in the unit influences its quality. These results suggest a new view of product dependencies with significant implications regarding how we think about modularizing the system and how development work is organized. The effect of the structure of the network of product dependencies elevates the idea of modularity in a system to the level of "clusters" of source-code files. These highly interrelated sets of files become the relevant unit to consider when development tasks and responsibilities are assigned to organizational groups.

The second significant contribution of this study is the recognition and the assessment of the impact the engineers' social network has on the software development process. Nagappan et al. [36] have examined the impact on failure proneness of structural properties of the formal organization (e.g., organizational chart). However, the informal organization which emerges as part of personal relationships is significantly more important for performing tasks in organizations [30]. Similarly, Meneely et al. [31] looked at the relationship among developers based on a file-touched network that may to some extent reflect social relationships among the developers that are more directly captured using workflow measures. Our measures of work dependencies capture the important elements of the informal organization in the context of software development tasks. Our results showed that individuals that exhibited a higher number of workflow dependencies and coordination requirements were more likely to have defects in the files they worked on. These findings suggest the difficulty of needing to receive work from or coordinate with multiple people and manage those relationships appropriately in order to perform the tasks.

This study has an additional characteristic worthy of note. The empirical analyses were replicated across two distinct projects from two unrelated companies obtaining consistent results. This replication provides us with unusually good external validity that is not easily achieved given proprietary concerns, etc.[3] We believe this study provides a proof of concept that such analyses are possible, and given the improved external validity, we think such an approach should be adopted (wherever logistics permit) as a standard of validity for industry studies.

7.1 Threats to Validity and Limitations

First, it is important to highlight some potential concerns for construct validity, particularly regarding work dependencies. Over the years, there have been many efforts to measure task interdependencies in the context of software

[3] In our case, it required a strategy in which data extraction was performed on machines inside company firewalls, to ensure that only anonymized data are provided for statistical modeling.

development. However, most of the approaches have focused on stylized representations of work dependencies, particularly in organizational studies (e.g., [10], [42]). Our study proposed two measures that capture the fine-grained dependencies that exist in software development and emerge over time as technical decisions are implemented. Certainly, there might be other potentially superior measures of development work dependencies; however, little is known about how to develop such measures.

Operationalization of software dependency measures is fraught with difficulties as projects produce products for different domains, using different tools and disparate practices, making it difficult to design measures that capture aspects of the same phenomena across unrelated projects. Therefore, we felt it was important to replicate the entire measurement and analysis process on two unrelated projects each using different sets of tools and practices. Furthermore, we investigated the stability of the results by analyzing individual releases and using random effects models to account for potential autocorrelation.

The work reported in this study has several limitations. First, our analysis cannot claim causal effects. For example, even though dependencies in workflow are related to customer-reported defects, it may be possible that the defects somehow increase the dependencies in the workflow. Second, our results on the role of syntactic dependencies are based on two projects where the software was developed in two programming languages (C and C++) that are somewhat similar in terms of how technical dependencies are represented. Projects that involve programming languages with very distinct technical properties might exhibit a different impact of syntactic dependencies on failure proneness.

7.2 Applications

7.2.1 Enhancing Dependency Awareness

We observed that logical dependencies were considerably more relevant than syntactic dependencies in relation to the failure proneness of a software system. They may also be less apparent to developers since they are not as easily discovered by tracing function calls, value assignments, or other things locally visible in the code.

Tools such as TUKAN [41], Palantir [39], and Ariadne [44] provide visualization and awareness mechanisms to aid developers in coordinating their work. Those tools achieve their goal by monitoring concurrent access to software artifacts, such as source-code files, and by identifying syntactic relationships among source-code files. This information is visualized to assist the developers in resolving potential conflicts in their development tasks. Using the measures proposed in this paper, new tools or extensions to those tools could be developed to provide an additional view of product dependencies using logical dependencies. These new tools would then be in a position to provide complementary product dependency information to the developers which could be more valuable in terms of raising awareness among developers about the potential impact of their changes in the software system. Moreover, since logical dependencies might be of different types such as implicit relationships (e.g., events), cascading function calls, or time-related relationships, tools could leverage such a categorization to provide more selective awareness information for particular user needs or work contexts. Second, these new tools could also provide a more precise view of coordination needs among developers using the work dependencies measures presented in this paper. For instance, the coordination requirements measure goes beyond identifying such dependencies, allowing developers to identify those files that have dependencies among themselves when those dependencies are not explicitly determined. It is important to also highlight that the development of future tools that use logical and coordination requirements dependencies is faced with important challenges such as the identification of the most relevant subset of dependencies for a particular work context and the presentation of such information to improve awareness and limit “play the system” behavior. There are also some minor but quite relevant process-related issues that require attention such as difficulty of maintaining consistent data about modification requests and version control changes over time and automation of the collection and processing of the data.

7.2.2 Reducing and Coping with Dependencies

Once developers, architects, or other relevant stakeholders become aware of particular patterns of technical dependencies, they could be in a position to utilize specific techniques to reduce those dependencies, in particular logical relationships. For instance, system rearchitecting is a promising technique to reduce logical dependencies and in a large system it was demonstrated to relate to quality improvements [22]. Other code reorganization techniques that make the structure of the systems more suitable for geographically distributed software development organizations could also focus their attention on logical dependencies. Such is the case of the globalization by chunking approach [33] that provides a way to select tightly clustered groups of source-code files (in terms of logical dependencies) that exhibit few logical dependencies with the rest of the system. Alternatively, methods to make logical dependencies more explicit by, for example, introducing syntactic dependencies where only logical dependencies exist could be explored given the important difference between the role of logical and syntactic dependencies suggested by our results.

In recent years, a number of tools that either implement some of the code reorganization approaches described in the previous paragraph or provide new mechanisms for coping with technical dependencies have been proposed. For instance, tools that highlight and filter changes from different releases helping to cope with interdependencies between changes in subsequent releases have been shown to improve productivity [1]. The results of this study provide valuable information to allow this type of tool to focus on those dependencies that are most relevant.

7.2.3 Guiding Future Research

While it seems clear that logical dependencies play a major role in software failures, we do not yet have a clear idea of the precise nature of these dependencies. Research and practices focused on syntactic dependencies, as found in strongly typed languages, for example, are likely responsible for weakening the relationship between such dependencies and fault proneness. We suggest that an emphasis on understanding the precise nature of logical dependencies is a fertile area for future research. Such research could, for example, examine the code that is changed together to understand if it represents cascading function calls or semantic dependencies, platform evolution, or other types of relationships. A more detailed understanding of the bases of logical dependencies is an important future direction with implications in research areas such as software quality and development tools.

ACKNOWLEDGMENTS

The authors gratefully acknowledge support by the US National Science Foundation under Grants 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 the Software Engineering Institute grant for "Improving Architectural Design through Organizational Considerations." The authors also gratefully thank A. Hassan and R. Holt for providing the source code for their C-REX tool.

REFERENCES

[1] D. Atkins, T. Ball, T. Graves, and A. Mockus, "Using Version Control Data to Evaluate the Impact of Software Tools: A Case Study of the Version Editor," *IEEE Trans. Software Eng.*, vol. 28, no. 7, pp. 625-637, July 2002.

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

[3] V.R. Basili and B.T. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Comm. ACM*, vol. 12, pp. 42-52, 1984.

[4] L.C. Briand, J. Wust, J.W. Daly, and D.V. Porter, "Exploring the Relationships between Design Measures and Software Quality in Object-Oriented Systems," *J. Systems and Software*, vol. 51, pp. 245-273, 2000.

[5] R.S. Burt, *Structural Holes: The Social Structure of Competition.* Harvard Univ. Press, 1992.

[6] M. Cataldo, P. Wagstrom, J.D. Herbsleb, and K.M. Carley, "Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools," *Proc. Conf. Computer Supported Cooperative Work*, pp. 353-362, 2006.

[7] M. Cataldo, "Dependencies in *Geographically Distributed Software Development: Overcoming the Limits of Modularity*," PhD dissertation, Inst. for Software Research, School of Computer Sciences, Carnegie Mellon Univ., 2007.

[8] M. Cataldo, M. Bass, J.D. Herbsleb, and L. Bass, "On Coordination Mechanism in Global Software Development," *Proc. Int'l Conf. Global Software Eng.*, pp. 71-80, 2007.

[9] S.R. Chidamber and C.F. Kemerer, "A Metrics Suite for Object-Oriented Design," *IEEE Trans. Software Eng.*, vol. 20, no. 6, pp. 476-493, June 1994.

[10] K.C. Crowston, "Toward a Coordination Cookbook: Recipes for Multi-Agent Action," PhD dissertation, Sloan School of Management, Masschusetss Inst. of Technology, 1991.

[11] B. Curtis, H. Kransner, and N. Iscoe, "A Field Study of Software Design Process for Large Systems," *Comm. ACM*, vol. 31, pp. 1268-1287, 1988.

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

[13] C.R.B. de Souza, D. Redmiles, L. Cheng, D. Millen, and J. Patterson, "How a Good Software Practice Thwarts Collaboration —The Multiple Roles of APIs in Software Development," *Proc. Conf. Foundations of Software Eng.*, pp. 221-230, 2004.

[14] M. Eaddy, T. Zimmermann, K.D. Sherwood, V. Garg, G.C. Murphy, N. Nagappan, and A.V. Aho, "Do Crosscutting Concerns Cause Defects?" *IEEE Trans. Software Eng.*, vol. 34, no. 4, pp. 497-515, July/Aug. 2008.

[15] S.G. Eick, T.L. Graves, A.F. Karr, A. Mockus, and P. Schuster, "Visualizing Software Changes," *IEEE Trans. Software Eng.*, vol. 28, no. 4, pp. 396-412, Apr. 2002.

[16] S.D. Eppinger, D.E. Whitney, R.P. Smith, and D.A. Gebala, "A Model-Based Method for Organizing Tasks in Product Development," *Research in Eng. Design*, vol. 6, pp. 1-13, 1994.

[17] S. Faraj and Y. Xiao, "Coordination in Fast-Response Organization," *Management Science*, vol. 52, no. 8, pp. 1155-1169, 2006.

[18] N.E. Fenton and M. Neil, "A Critique of Software Defect Prediction Models," *IEEE Trans. Software Eng.*, vol. 25, no. 5, pp. 675-689, Sept./Oct. 1999.

[19] L.C. Freeman, "Centrality in Social Networks: I. Conceptual Clarification," *Social Networks*, vol. 1, pp. 215-239, 1979.

[20] J.R. Galbraith, *Designing Complex Organizations.* Addison-Wesley Publishing, 1973.

[21] H. Gall, K. Hajek, and M. Jazayeri, "Detection of Logical Coupling Based on Product Release History," *Proc. Int'l Conf. Software Maintenance*, pp. 190-198, 1998.

[22] B. Geppert, A. Mockus, and F. Rößler, "Refactoring for Changeability: A Way to Go?" *Proc. 11th Int'l Symp. Software Metrics*, pp. 35-48, 2005.

[23] T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy, "Predicting Fault Incidence Using Software Change History," *IEEE Trans. Software Eng.*, vol. 26, no. 7, pp. 653-661, July 2000.

[24] R.E. Grinter, J.D. Herbsleb, and D.E. Perry, "The Geography of Coordination Dealing with Distance in R&D Work," *Proc. Conf. Supporting Group Work*, pp. 306-315, 1999.

[25] A.E. Hassan and R.C. Holt, "C-REX: An Evolutionary Code Extractor for C," *Proc. Conf. Systems Eng. Research Meeting,* 2004.

[26] J.D. Herbsleb, A. Mockus, and J.A. Roberts, "Collaboration in Software Engineering Projects: A Theory of Coordination," *Proc. Int'l Conf. Information Systems*, 2006.

[27] J.D. Herbsleb and A. Mockus, "An Empirical Study of Speed and Communication in Globally Distributed Software Development," *IEEE Trans. Software Eng.*, vol. 29, no. 6, pp. 481-494, June 2003.

[28] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Trans. Programming Languages and Systems*, vol. 22, pp. 26-60, 1990.

[29] D.H. Hutchens and V.R. Basili, "System Structure Analysis: Clustering with Data Bindings," *IEEE Trans. Software Eng.*, vol. 11, no. 8, pp. 749-757, Aug. 1985.

[30] D. Krackhardt and J.D. Brass, "Intra-Organizational Networks: The Micro Side," *Social Network Analysis: Research in the Social and Behavioral Sciences*, pp. 207-229, Sage, 1994.

[31] A. Meneely, L. Williams, W. Snipes, and J. Osborn, "Predicting Failures with Developer Networks and Social Network Analysis," *Proc. Foundations of Software Eng.*, 2008.

[32] A. Mockus and D. Weiss, "Predicting Risk of Software Changes," *Bell Labs Technical J.*, vol. 5, pp. 169-180, 2000.

[33] A. Mockus and D. Weiss, "Globalization by Chunking: A Quantitative Approach," *IEEE Software*, vol. 18, no. 2, pp. 30-37, Mar./Apr. 2001.

[34] K.H. Moeller and D. Paulish, "An Empirical Investigation of Software Fault Distribution," *Proc. Int'l Software Metrics Symp.*, pp. 82-90, 1993.

[35] N. Nagappan and T. Ball, "Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study," *Proc. First Int'l Symp. Empirical Software Eng. and Measurement*, pp. 363-373, 2007.

[36] N. Nagappan, B. Murphy, and V.R. Basili, "The Influence of Organizational Structure on Software Quality: An Empirical Case Study," *Proc. Int'l Conf. Software Eng.*, pp. 521-530, 2008.

[37] D.L. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules," *Comm. ACM*, vol. 15, pp. 1053-1058, 1972.

[38] M. Pinzger, N. Nagappan, and B. Murphy, "Can Developer-Module Networks Predict Failures?" *Proc. Foundations of Software Eng.*, 2008.

[39] A. Sarma, Z. Noroozi, and A. Van Der Hoek, "Palantir: Raising Awareness among Configuration Management Workspaces," *Proc. Int'l Conf. Software Eng.*, pp. 444-453, 2003.

[40] R.W. Selby and V.R. Basili, "Analyzing Error-Prone System Structure," *IEEE Trans. Software Eng.*, vol. 17, no. 2, pp. 141-152, Feb. 1991.

[41] T. Schummer and J.M. Haake, "Supporting Distributed Software Development by Modes of Collaboration," *Proc. European Conf. Computer-Supported Collaborative Work,* pp. 79-89, 2001.

[42] N. Staudenmayer, "Managing Multiple Interdependencies in Large Scale Software Development Projects," unpublished PhD dissertation, Sloan School of Management, Massachusetts Inst. of Technology, 1997.

[43] W.P. Stevens, G.J. Myers, and L.L. Constantine, "Structure Design," *IBM Systems J.,* vol. 13, pp. 231-256, 1974.

[44] E. Trainer, S. Quirk, C. de Souza, and D. Redmiles, "Bridging the Gap between Technical and Social Dependencies with Ariadne," *Proc. Workshop the Eclipse Technology Exchange,* pp. 26-30, 2005.

[45] J.D. Thompson, *Organizations in Action: Social Science Bases of Administrative Theory.* McGraw-Hill, 1967.

[46] E. von Hippel, "Task Partitioning: An Innovation Process Variable," *Research Policy,* vol. 19, pp. 407-418, 1990.

[47] D.J. Watts, *Small Worlds: The Dynamics of Networks between Order and Randomness.* Princeton Univ. Press, 1994.

[48] T. Zimmermannn and N. Nagappan, "The Predicting Defects Using Network Analysis on Dependency Graphs," *Proc. Int'l Conf. Software Eng.,* pp. 531-540, 2008.

Marcelo Cataldo received the MS and PhD degrees in computation, organizations, and society from Carnegie Mellon University in 2007. He also received the BS degree in information systems from the Universidad Tecnologica Nacional (Argentina) in 1996 and the MS degree in information networking from Carnegie Mellon University in 2000. His research interests are geographically distributed software development with special focus on the relationship between the software architecture and the organizational structure in large-scale software development projects. He is a senior research engineer at Robert Bosch's Research and Technology Center.

Audris Mockus received the BS and MS degrees in applied mathematics from the Moscow Institute of Physics and Technology in 1988. In 1991, he received the MS degree and in 1994 he received the PhD degree in statistics from Carnegie Mellon University. He is interested in quantifying, modeling, and improving software development. He designs data mining methods to summarize and augment software change data, interactive visualization techniques to inspect, present, and control the development process, and statistical models and optimization techniques to understand the relationships among people, organizations, and characteristics of a software product. He works in the Software Technology Research Department of Avaya Labs. Previously, he worked in the Software Production Research Department of Bell Labs.

Jeffrey A. Roberts received the MS and PhD degrees in information systems from Carnegie Mellon University and the MBA degree from the University of Texas at Austin. He is an assistant professor of information systems management at the Palumbo Donahue School of Business at Duquesne University. His research interests include software development methodology, open source software, and e-enabled business process improvement. He is a member of the Association for Information Systems.

James D. Herbsleb received the JD degree (1980) and the PhD degree in psychology (1984) from the University of Nebraska, and the MS degree in computer science (1991) from the University of Michigan. He is a professor of computer science and director of the Software Industry Center at Carnegie Mellon University. His research interests lie primarily in the intersection of software engineering and computer-supported cooperative work, focusing on such areas as geographically distributed development teams, open source software development, and, more generally, on coordination in software engineering.