Proximity: a Measure to Quantify the Need for Developers' Coordination

Kelly Blincoe
Computer Science Department
Drexel University
Philadelphia, PA, USA
kelly.blincoe@drexel.edu

Giuseppe Valetto
Computer Science Department
Drexel University
Philadelphia, PA, USA
valetto@cs.drexel.edu

Sean Goggins
College of Information, Science and Technology
Drexel University
Philadelphia, PA, USA
sgoggins@drexel.edu

ABSTRACT
We describe a method for determining coordination requirements in collaborative software development. Our method uses "live" data based on developer activity rather than relying on historical data such as source code commits which is prevalent in existing methods. We introduce proximity, a measure of the strength of the work dependencies that lead to coordination requirements among members of a software development organization. Our proximity measure relies on a tool which captures the interactions of a developer with her IDE. It quantifies the similarity between records of interactions of developers as they work on their assigned tasks. We describe an algorithm that measures proximity between pairs of tasks or pairs of developers. Through an empirical study on an open source project that routinely records environment interaction data, we show how proximity accurately determines coordination requirements. The proximity measure thus enables proactive detection of coordination requirements and makes possible real time intervention and coordination facilitation via management-, design- and team-related decisions.

Author Keywords
Awareness, Proximity, Management, Coordination Requirements, Socio-Technical, Task Context, Tools.

ACM Classification Keywords
H5.3. Information interfaces and presentation (e.g., HCI): Group and Organization Interfaces.

1. INTRODUCTION

The coordination of concurrent activities by multiple developers remains problematic for software development organizations [18]. Software engineering pioneers such as Parnas [20] and Brooks [2] recognized the importance of efficiently managing work dependencies and coordination overhead arising within a development team. Such dependencies are becoming more critical as software organizations become larger and more distributed [3,13,14].

Dependencies between tasks often result in Coordination Requirements (CRs) among team members. Cataldo et al. [4,6] introduced a framework to detect and quantify CRs between pairs of software developers by identifying the technical dependencies between software artifacts modified during their assigned tasks. This formalization of CRs led to the definition of Socio-Technical Congruence (STC) in software development. STC is an index that measures the degree to which actual acts of coordination mirror coordination requirements. Empirical studies suggest that high levels of STC are beneficial: when coordination activities focus on the empirically identified CRs, productivity is likely to improve [4,6,25].

Current methods for detecting CRs and calculating socio-technical congruence have two serious drawbacks. First, CRs are identified by mining the source control repository of the project for changes to artifacts committed by a developer. This type of data is typically available only towards the end of the development work for a task. Second, for each file committed to a source code repository, a developer may have consulted several other files. Knowledge of this source code reference behavior is inaccessible from commit records.

Without a "live" view of activities, CRs and STC are not actionable devices for managing coordination in software projects. Several potential applications for such a live view have been discussed. Ehrlich et al. have elaborated a way to rank CRs in a project, enabling prioritization of those whose resolution can improve STC the most [11]. Valetto et al. proposed a set of management-, design-, and team-related decisions that can be used as alternatives to resolve CRs; each with its associated costs and risks [27]. These and other techniques require the ability to detect, analyze and manipulate CRs as they emerge.

CSCW 2012, February 11–15, 2012, Seattle, Washington.

To surface CRs in a timely way, we introduce a measure we call proximity, which characterizes the similarity of working sets of artifacts employed by developers as they work. Proximity can be applied to pairs of tasks as well as pairs of developers who carry out those tasks.

To measure proximity, we need to record developers' actions on artifacts as they occur. Several such recording frameworks exist. Because of its popularity, we focus on the Mylyn framework (formerly Mylar) [16,17]. By means of a field study, we show how our proximity measure detects CRs at least as accurately as other methods such as that described in Cataldo et al. [4,6] We also show that it can detect CRs substantially earlier than other methods. Thus, proximity can provide enhanced support to the management of coordination in software projects.

The rest of the paper is organized as follows: in Section 2 we discuss related work; in Section 3 we present our method, including the data we use and our proximity algorithm; in Section 4, we describe in detail the setting, method and results of our empirical study; in Section 5, we discuss the significance of our contributions; finally, in Section 6, we offer some concluding remarks.

2. RELATED WORK

In the course of any non-trivial software project, it is paramount to be able to manage work dependencies. If such dependencies are not recognized early, they must be reconciled later. This often leads to lower product quality and developer productivity [5,14,25].

The design literature, beginning with Parnas' recognition of the workflow implications of modularization [20], focuses on ways to streamline the technical dependencies between modules as a way to maximize task parallelism [25,26]. In software, such technical dependencies are often derived from a syntactic analysis of the code base. As an alternative to identify technical dependencies, Gall et al. introduced the notion of logical coupling based on the "files changed together" heuristic [12] which aims at identifying semantic relationships that may not manifest in the syntax of the programmatic implementation of the software product. Whereas syntactic dependencies exist a priori with respect to a project and an organization, logical couplings reflect accumulated empirical evidence about how the development work unfolds in the project.

The shift from studying technical dependencies *per se* to exploring the complex interplay they have with organizational structure and project dynamics continues in recent research on the socio-technical aspects of software engineering. For example, several field studies show how information hiding may hamper the ability of development team members to coordinate with one another by decreasing awareness of important work decisions [3,13]. Herbsleb et al. [15] argued for a systematic framework to satisfy, as opposed to try to reduce, work dependencies by explicitly computing and orchestrating the coordination needs of a project. Works on CRs and STC in software development are part of this thread of research [4,5,6].

The study of Coordination Requirements

Cataldo et al. [4,6] describe STC as an index based on the alignment between team interactions and technical dependencies. Conway [7] was the first to describe the possibility of such an alignment. STC measures the extent to which CRs and coordination behavior are aligned in practice. STC is expressed as a simple ratio between CRs that are satisfied by actual acts of coordination and the set of outstanding CRs between developer pairs. Those CRs can be identified according to the following formula:

$$CR = TA \times TD \times TA^t$$

In this formula [6], TA is a people-by-task matrix representing task assignments, and TA^t is its transpose. TD is a task-by-task matrix capturing the work dependencies between tasks. Those are established by considering the technical dependencies occurring between artifacts involved in those tasks. According to this formula, a CR between two developers Alice and Bob can be represented graphically as in Fig. 1. Arc TD_{ab} represents a technical dependency between software artifacts S_a and S_b. These artifacts are involved in tasks to which Alice and Bob, respectively, are assigned (denoted by arcs TA_a, TA_b).

Among the methods for computing technical dependencies, Cataldo et al. offer empirical evidence that logical coupling obtained by tracking files that have been historically checked in together provides a more reliable representation of the technical dependencies relevant for CR detection than syntactic coupling does [5]. However, logical dependencies are computed based on past project history and are only visible after work is completed. Even when syntactic dependencies are chosen, as done by Ehrlich et al. [11], they only become fully known following a commit.

In prevalent CR detection methods, the association of developers to artifacts they work on becomes visible only after code is committed. That limits the potential of CRs as a means to support coordination as the development work unfolds. Our research aims to close that gap.

Applications of Coordination Requirements

Awareness [10] is one of the ways used to conceptualize coordination. A recent work shows that, for a software engineer, the most important form of awareness is locating and keeping up to date with other developers whose work is relevant to her own [1]. Examples of systems that try to achieve that by employing abstractions similar to

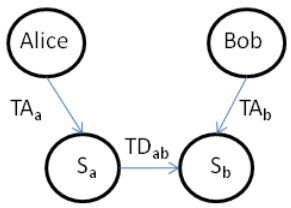

Figure 1. Representation of a Coordination Requirement.

CRs include Ariadne [8], EEL [19], Tesseract [23], and Codebook [1]. Those systems offer various mechanisms including visualization of socio-technical networks (Ariadne and EEL), dashboards (Tesseract), and query and search facilities (Codebook) to try to identify and show important work relationships within a team. All of those systems rely upon the commit logs in the source code repository to establish technical dependencies among artifacts. They use these dependencies to compute transitive relationships between developers. Therefore, these approaches suffer from the same drawback we mentioned earlier, that is, they are hampered in their ability to detect CRs and provide timely awareness support. Among them, the work most similar to our approach is EEL because of the way it combines the concepts of working set and CRs. However, our proximity measure aims at bypassing reliance on any specific conceptualization of technical dependency relationships and removing the reliance on post-mortem data.

There are other awareness approaches in collaborative software development that have tried to leverage live workspace information. For example, Palantír uses notifications to keep a developer abreast with what happens in her colleagues' workspaces [22]. Notifications relate only to changes occurring to the same artifacts a developer has in her own workspace. Palantír thus provides each developer with timely information about any arising same-artifact conflicts which can be seen as a narrow subset of CRs. CollabVS is another awareness system that uses notifications. Compared to Palantír, it has an expanded model of interest. The CollabVS model captures additional conflicts by considering a subset of syntactical dependencies between artifacts [9]. It issues instantaneous warnings to developers as an individual instance of conflict emerges, but it does not offer a model for quantifying the strength of CRs. By introducing the proximity measure, we contribute such a model and can draw a complete and explicit view of coordination across the whole team.

3. TASK CONTEXT AND PROXIMITY

Task Context

The main goal of our research is to identify CRs accurately and early enough to enable decisions on how to best resolve those work dependencies. For example, if Alice and Bob are assigned to tasks that require extensive negotiation and synchronization around interdependent software artifacts, early design decisions to refactor those dependencies and modify modularization can be taken to improve Alice's and Bob's productivity.

We look to capture meaningful data from development work as it happens in order to build an incremental record of the development activities carried out on the working set of artifacts for a task. Several facilities of this kind have been described in the literature [21,24]. Here we focus on Mylyn, a tool that captures the interactions of a developer with her IDE and constructs a data structure called a "task context" [16,17]. Mylyn is an Eclipse plugin whose goal is to provide an individual developer with a task-centric interface. It adapts the Eclipse GUI and focuses its presentation on what is most relevant for that developer in the context of the task she is performing. To achieve that, Mylyn stores significant GUI events, such as software element manipulation, artifact consultation, or the issuing of IDE commands in a *task context event*. Those events are weighted according to a model of interest that takes into account – among other things – action type, frequency of interaction, and a decay factor [16]. Mylyn task contexts therefore characterize a task in terms of its working set, the relative importance of artifacts in the working set, and the nature of the interactions with those artifacts. Since this kind of data is of general significance, Mylyn has started to extend to a number of other prominent development environments besides Eclipse.

A sequence of task context events in Mylyn is therefore a list of IDE interactions by the developer described in an XML dialect. The pertinent information associated to each action includes:

- Kind: type of interaction (selection, edit, etc.)
- Structure Handle: a unique ID that identifies a software artifact.
- Start Date: a timestamp
- End Date a timestamp

We are interested in only selection and edit events. Selection events are captured when a developer opens a file in the Eclipse IDE. Additional selection events can be captured whenever a developer navigates to a class, method or field defined in that file. Edit events are captured as the developer makes changes to those file locations.

The Mylyn structure handle provides more granularity than simply file names. It identifies the file name, class name and even the name of the class element (method or attribute) when available. We can thus choose to consider artifacts at different granularity levels allowing us to determine if two developers were working on the same area of code within a large file. This level of granularity is not available when looking only at commit information.

Note that the original purpose of Mylyn task contexts is to support the work of an individual developer during a single task or to allow a single developer to easily switch between tasks without loss of focus. Instead, we use that same information to support a collaborative goal. We want to detect areas in the project that are relevant for the coordination of concurrent tasks by looking at the amount of overlap between the working sets of the developers assigned to those tasks. To achieve that, we introduce our proximity measure which applies to task contexts and, by extension, to tasks and developers.

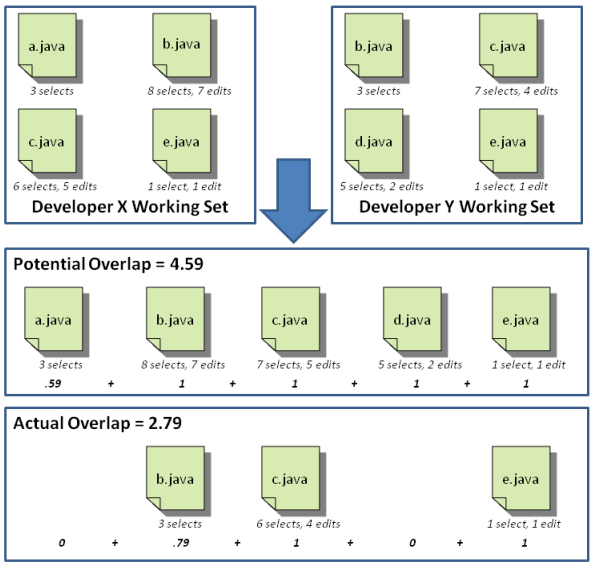

Figure 2. Proximity Algorithm.

Proximity

Proximity is a relative measure that conveys the amount of similarity between the development activities recorded in any two working sets. A working set can be construed at different levels simply by aggregating context events appropriately. To compute proximity between developers, each developer's working set will include all task context events produced for all tasks carried out by that developer during a time frame of interest. To compute the proximity of tasks, the working set of a task includes all context events produced for that task by all developers who worked on that task.

When calculating proximity between two working sets, our algorithm considers all actions recorded for each artifact in each working set in order to apply a weight to that artifact's proximity contribution. There are several types of actions captured by Mylyn. For this study we consider only select and edit actions. Other actions used within Mylyn, such as prediction, propagation and manipulation, were purposely left out of our algorithm. Since these event types are specific to Mylyn, including them would make the replication of our experiments and findings outside of the Mylyn framework difficult. Manipulation actions represent information that developers can explicitly provide to Mylyn to emphasize the importance (or lack thereof) of a given artifact for the task at hand. Prediction and propagation events occur when Mylyn itself "suggests" other artifacts, which are not included in a developer's working set, but appear to be structurally relevant.

We base our weights on the factors that Mylyn itself uses when computing its Degree-Of-Interest (DOI) model for each artifact present in a task context [16]. The DOI provides Mylyn with a way to prioritize the presentation of elements in its task-based interface. The factors used in the Mylyn DOI model are 1 for select events and 0.7 for edit events where higher values indicate higher interest. Those factors have been extensively validated in practice by the Mylyn user community.

Since edit events in the GUI are always preceded by at least one selection event, our algorithm weighs an edit overlap as 1.7. An edit overlap occurs when both developers edit the same artifact. Each developer contributes 1.7 to the overlap score for a total score of 3.4, which is the maximum score for an overlap. In a mixed overlap, one developer edits the artifact while the other developer selects or views the same artifact. In this case, one developer provides a score of 1.7 for the edit event and the other developer provides a score of 1 for the selection event for a total score of 2.7. In a selection overlap, neither developer edits the artifact, but both developers select and view the artifact. In this case, each developer contributes a score of 1 for a total score of 2. We then calculate our weights as percentages of the maximum possible overlap score:

- Edit overlap: 1.7 + 1.7 = 3.4, 3.4/3.4 = 1
- Mixed overlap: 1.7 + 1 = 2.7, 2.7/3.4 = .79
- Selection overlap: 1 + 1 = 2, 2/3.4 = .59

With those weights we compute two scores: an actual overlap and a potential overlap. Actual overlap is the intersection of the two working sets. Potential overlap is the union of the two working sets representing the maximum similarity score had there been perfect overlap between the two sets. Our proximity measure is the ratio between the actual overlap and the potential overlap. Fig. 2 shows the results of our algorithm on two sample working sets. The actual overlap includes files B, C and E since these three files are included in both working sets. B has an overlap score of .79 since working set X edited the file while working set Y only selected the file. C and E have an overlap score of 1 since both working sets edited these files. The Potential overlap includes all five files since we are now taking the union of the two sets. B, C, D and E each have an overlap score of 1 since at least one of the working sets edits each of these files. File A has an overlap score of .59 since only selection events exist for this file.

As a final step of our algorithm, we apply a scaling factor where we consider the total number of events recorded within all working sets. The scaling factor is the ratio between the number of overlapping events for each pair and the average number of overlapping events computed across all pairs in the population. The rationale for this scaling factor is to place greater weight on those pairs engaged in complex development activities, where coordination is likely to be more necessary, than for pairs involved in simple tasks. When performing a complex task, a developer is likely to spend a lot of time on the task and will therefore create many context events. Conversely, a developer completing a task via a trivial change will likely produce a small number of events.

Research Objectives

This study explores the extent to which our proximity measure enables the detection and management of CRs between software developers. Having introduced the semantics of proximity and its measurement technique, we now state our main research objectives:

Research Question RQ1: to what extent are the proximity relationships obtained with task context information (aggregated at the developer level) analogous to CRs determined using information about artifact commits and technical dependency relationships? RQ1 speaks to the feasibility of detecting CRs by leveraging developers' working sets and their overlap, thus bypassing the analysis of technical dependencies between artifacts.

Research Question RQ2: can CR detection based on task contexts and proximity provide more accurate results than current methods? RQ2 speaks to the ability of leveraging the detailed information offered by task contexts to improve precision, recall, or both in detecting CRs.

Research Question RQ3: can the data provided by task contexts support the detection of CRs significantly earlier than current methods? RQ3 speaks to the possibility of supporting timely decisions on how to handle CRs as they form in the project and has immediate consequence on coordination management practices and technologies.

4. CASE STUDY - MYLYN

We carried out an empirical study on the open source project which developed the Mylyn framework itself. Contributors to the Mylyn Eclipse plugin use Mylyn for their work and conventionally publish the task context data in the project's Bugzilla repository (code contributions without attached context data are often rejected). Sixty-nine other projects in the Eclipse community alone report freely available Mylyn task contexts. We selected the Mylyn project because it provides the largest amount of and most complete task context information.

Data Collection and Preparation

We collected development data over eight releases of Mylyn (v2.0 - v3.3). These releases span almost three years, from December 2006 until October 2009. Mylyn data is captured as an attachment file to a Bugzilla entry (which denotes a development task). Our analysis focuses on the 1,970 Bugzilla entries which contain such attachments during the study period. For those tasks, we also collected all code commits recorded in the project's SVN repository and all patch description files. Patch descriptions are attached by those developers who contribute code but do not have commit privileges and report the diff information for all artifacts that were modified as part of a patch. They are thus semantically equivalent to commits. Work on Mylyn during the study period involved 51 distinct developers who attached task context data (context attachers) and 8 distinct developers who committed code (committers). Each release has between 10 and 32 distinct context attachers and 4 to 6 distinct committers.

We compiled three data sets, summarized in Table 1. For the first data set, DS1, we looked at the activity of the group of committers. Each Bugzilla task may have one or more context attachments. We expanded the contexts attached to the 1,970 tasks to enumerate the 588,796 selection and edit events contained within. The majority of these events relate to Java source code files. Some pertain to .class, .jar, and other file types which are by-products, rather than objects, of development work. We thus focused only on the 450,747 context events dealing with Java source code artifacts.

We similarly filtered the SVN commits to include only java files, yielding 27,074 commits of individual artifacts over the same time period. Most of the commits, 25,135 or 92.8%, were linked directly to 1,835 unique tasks via the Bugzilla task IDs conventionally inserted by developers in their commit comments. We then intersected those 1,835 tasks for which we have commit data with the 1,970 tasks which have context data attachments. This resulted in 1,127 tasks which have both context data attached and associated commit records. This set includes 10,647 artifact commits and 450,757 context events.

Upon further examination of DS1, we noticed that a number of file changes reported in commits for a given task were not matched by any edit events in that task's context data. In the time period considered, we identified 6,507 commits out of 10,647 that are not matched by any proof of editing of the same file in the associated task contexts by the developer who committed the change. One reason is that, although it is customary to submit context when committing a change for a task, Mylyn developers do not always abide to this convention. Another reason is that the developer who commits a change is not always the developer who contributed it. We split DS1 in two: DS1-a includes the 4,140 commits for which we have matching events within task contexts; DS1-b includes the other 6,507 commits.

Data Set		Actors	Artifact info	Context info	Matching artifact commit and context info
DS1	DS1-a	8 committers	4,140 SVN commits	450,757 task context events	YES
	DS1-b		6,507 SVN commits	450,757 task context events	NO
DS2		34 patch contributors	1,387 patch file edits	345,521 task context events	YES
DS3		DS1-a and DS2 combined			YES

Table 1. Data sets.

Since looking at just commit data, as in DS1, would limit our analysis to the few developers with commit privileges, we compiled a different data set. DS2 considers the patch description files attached to Bugzilla records. Those files contain data on 7,196 java file changes spanning 936 tasks. Mylyn contexts attached to those tasks yield 345,521 context events for java artifacts contributed by 47 developers. Again, we match the patch description files retrieved from Bugzilla to task context events, leaving 1,387 file changes with associated task context data contributed from 34 different developers.

It is important to notice that DS1 and DS2 are disjoint. There is only a single developer, common to both sets. This ensures there are no overlapping pairs of developers among the two sets. Therefore, DS1 and DS2 represent complementary analyses over the full picture of the project activity. The 1,387 patch file changes in DS2 represent substantially different development work from what is captured in DS1. The one common developer is responsible for 219 changes in DS2, and a manual inspection revealed that only 11 of those 219 changes overlap with commits made by the same developer in DS1 (5%).

Finally, we combined DS1-a and DS2 into a third data set DS3, which incorporates all records of file changes (either via commit traces or patch diff files) and all context events.

Results of the Field Study

RQ1: Is Proximity a Good Proxy for Coordination Requirements?

We started our analysis with DS1-a. We used the 4,140 commits in that set to calculate CRs between committers according to the method described by Cataldo et al. As recommended in [4,6], to define technical dependencies between software artifacts, we used the "files committed together" heuristic introduced by Gall [12] which defines logical coupling between artifacts. We then computed our proximity scores for the same set of committers. Initially, since Cataldo's method calculates CRs at the file level, we considered only the file associated with each context event and ignored the finer-grained information made available by Mylyn at the method and attribute level. Since releases are a logical unit of concurrency for tasks in an open source project, we looked at work in each release separately. DS1-a yielded 70 pairs of committers across all eight releases.

First, we checked that higher values of proximity correlate with the likelihood of a CR by performing a point-biserial correlation with a binary vector denoting the presence of CRs. We then performed a Spearman correlation between the count of CRs for each pair and their proximity scores. We chose to use a Spearman correlation because both the CR counts and proximity scores are not normally distributed. Both tests were statistically significant and provided us with strong positive correlations, as shown in Table 2. Moreover, we observed that 46 of the 70 pairs of committers had some CR, and 43 of those 46 pairs (93.5%) have proximity > 0. Conversely, of the 24 pairs with no CRs, seven present a proximity score of 0.

Test	Unit of Work	p-value	rho
Spearman	File	2.4e-11	0.69
Point-biserial	File	4.9e-07	0.55
Spearman	Granular	6.8e-09	0.62
Point-biserial	Granular	8.8e-06	0.49

Table 2. DS1-A: CR V PROXIMITY CORRELATIONS

We repeated these tests by computing proximity at the finest granularity level for artifacts reported in Mylyn context events. Our expectation was that analysis of contexts at finer granularity would yield less proximity and lower levels of correlation with CRs calculated with Cataldo's method. The results shown in Table 2 are in line with those expectations, but correlations between proximity and CRs are still strong and significant. Of the 46 CR pairs, 42 (91.3%) have proximity > 0. Therefore, the additional granularity removes proximity from one of the 43 pairs with proximity when looking at file level granularity. Eight pairs have a granular proximity score of 0.

These findings seem to confirm that proximity is a valid proxy for CRs. However, DS1-a contains rather homogenous commit and context data: artifacts that are committed are likely to show up prominently in select and edit actions within the task contexts of their respective committer. Therefore, we performed analogous tests on DS1-b. In this data set, the CR and proximity analyses are performed on two completely distinct sets of artifacts since recorded context events do not align with the artifacts that were committed. However, since we are still looking at manifestations of the work done by the same developers within the same set of tasks, we speculated that proximity scores should remain a good indicator of CRs. The results of our tests on DS1-b, in Table 3, show strong and statistically significant positive correlations. Among the 75 developer pairs showing up in this data set, 33 have a CR, and all 33 pairs have a proximity score > 0 at both the file and granular level of analysis. Of the 42 pairs with no CRs, 10 of the pairs also have a proximity score of 0 at the file level and 14 at the more granular level. These results show that the proximity relationship can capture CRs by examining working set information even when that information is partial.

We then moved our analysis of RQ1 from the restricted core group of Mylyn committers of data set DS1 to the

Test	Unit of Work	p-value	rho
Spearman	File	8.7e-09	0.60
Point-biserial	File	1.1e-08	0.59
Spearman	Granular	2.5e-07	0.54
Point-biserial	Granular	1.8e-07	0.55

Table 3. DS1-B :CR V PROXIMITY CORRELATIONS.

Test	Unit of Work	p-value	rho
Spearman	File	< 2.2e-16	0.55
Point-biserial	File	< 2.2e-16	0.54
Spearman	Granular	< 2.2e-16	0.57
Point-biserial	Granular	< 2.2e-16	0.55

Table 4. DS2 CR V PROXIMITY CORRELATIONS

Test	Unit of Work	p-value	rho
Spearman	File	< 2.2e-16	0.68
Point-biserial	File	< 2.2e-16	0.66
Spearman	Granular	< 2.2e-16	0.68
Point-biserial	Granular	< 2.2e-16	0.66

Table 5. DS3 CR V PROXIMITY CORRELATIONS

larger group of project contributors represented in data set DS2. DS2 yielded 277 developer pairs; of which 37 have CRs. Of these, 28 (75.7%) have file-level proximity > 0 and 24 (64.9%) have granular proximity > 0. Of the 240 pairs with no CRs, 206 also have a proximity score of 0 at the file level and 222 at the granular level. Table 4 shows strong positive correlations that are also statistically significant.

Our largest data set, DS3, which combines DS1-a and DS2, includes a total of 347 developer pairs. The correlation results for this data set are shown in Table 5. Using DS3, we further investigated the relationship between CRs and our proximity measure by means of a regression model. We employed a zero-inflated negative binomial regression (zinb) since the CR count is highly skewed and presents many zeroes (264 out of 347 developer pairs have no CRs). The zinb model is statistically significant (χ^2=161.69, df=2, p < 2.2e-16). Results from the regression are shown in Table 6 for both the count and the excess zeroes portions of the model (white and grey rows, respectively). In particular, a one-unit increase in proximity (a large increase in the proximity scale) causes a 2.20-times increase in the log of expected CR count. That is an expected ~9-times increase in CRs for each one-unit increase in proximity. To investigate the influence of the select events in defining our proximity metric, we recomputed proximity including only edit event overlaps. We then ran the same zinb regression and obtained a new model. This new model captures only direct edit conflicts and is, therefore, similar to what could be observed with tools like Palantír. It is still statistically significant (χ^2=157.17, df=2, p < 2.2e-16). We then compared the AIC scores of the new model and our original model of Table 6. We found that our original model has considerably better support since it has a lower AIC. The difference in the AIC scores is 4.51416. Thus, our original proximity model is almost 10 times as likely as the edit event-only model to minimize information loss.

We conclude that the proximity measure is a valid proxy to CRs determined using information about artifact commits and technical dependency relationships.

RQ2: How Accurate is Proximity in Determining Actual Coordination Requirements?

To investigate RQ2, we first reviewed how developer pairs with CRs match against pairs with proximity >0, and conversely how developer pairs with no CRs match against pairs with proximity = 0 in terms of precision and recall. Results at the granular level are reported in Table 7. We assume here that CRs detected with the Cataldo et al. method are the ground truth.

Considering all proximity scores > 0 as indicators of CRs may cast a net that is too wide. A sensitivity analysis to understand the impact and appropriateness of different proximity thresholds will be part of our future work. However, conceptually, a threshold of 0 seems sensible. The lowest possible CR score of 1 indicates that a developer pair worked on only one pair of dependent files. The lowest proximity score of 0.01 also indicates (at least) one artifact overlapping in the developers' working sets. Since CRs computed according to Cataldo et al. are themselves only an approximation of ground truth, we then proceeded to manually examine some of the mismatches. Our goal was to determine if proximity is conducive to identify actual CRs more accurately by weeding out either false positives or false negatives of that method.

For potential false positives, we reviewed the four cases in DS1-a where developer pairs with CRs had granular proximity=0. The record of changes made by each pair of developers held in the relevant task contexts determined that in each case the developers operate on a totally disjoint sets of files and all of the recorded code changes were in areas of those files that appear unrelated to one another. An example is provided by the single CR that exists between developers 6 and 7 in release 3.2. Developer 6 committed BugzillaClient.java, while developer 7 committed BugzillaTaskEditorPage.java. The changes by developer 6 involve a character encoding method that is private to the BugzillaClient class. Developer 7 added a new section to the Mylyn task editor. Although we could ascertain those changes were semantically unrelated, the two involved files had been historically changed together by other developers often enough to cause a logical dependency to be established by the CR detection algorithm. We noticed analogous incidents in the other three cases in DS1-a. Those CRs are therefore false positives of the traditional method that our proximity algorithm correctly eliminates.

	Estimate	Std. Error	Z	p-value
(Intercept)	5.22	0.37	14.17	< 2.2e-16
Proximity	2.20	0.51	4.33	$< 10^{-4}$
Log(theta)	-2.01	0.14	-14.53	< 2.2e-16
(Intercept)	2.32	0.30	7.61	$< 10^{-13}$
Proximity	-106.49	33.18	-3.21	< 0.01

Table 6. ZINB REGRESSION: CR V PROXIMITY

Data Set	# of pairs	Precision	Recall
DS1-a	70	42/58 = 0.72	42/46 = 0.91
DS1-b	75	33/61 = 0.54	33/33 = 1
DS2	277	24/40 = 0.6	24/37 = 0.65
DS3	347	70/100 = 0.7	70/97 = 0.72

Table 7. PRECISION AND RECALL (GRANULAR)

Moving to potential false negatives, we examined the 16 pairs of committers in DS1-a that present some amount of granular proximity but have no CR. We were able to recognize two distinct types of behavior in this set. In one case, involving developers 3 and 7 during release 3.3, proximity contributions came exclusively by selection and mixed overlaps. The pair had seven mixed overlaps and six selection overlaps. Meaning that developers 3 and 7 viewed 13 of the same artifacts, of which seven were edited at some point by either developer 3 or developer 7, but no single artifact was edited by both developer 3 and developer 7. Since there were no overlapping commits, the traditional CR method does not allow for a CR to be detected. However, since we have the advantage of knowing not only what files are edited but also what files are consulted by a developer in the process of completing a task, our algorithm picks up what is likely to be an actual work dependency. Developer 3 and developer 7 repeatedly examined the same area of the software code base and consulted each other's code during their work for release 3.3.

The remaining 15 developer pairs we examined represent an even more interesting case. In each of these cases, the developers edit from 1 to 67 of the same artifacts (17 on average). CRs could not be established in any of these cases because at least one of the developers did not commit her changes. However, task contexts prove that those developer pairs were at one time engaged in development on the very same artifacts - the epitome of a Coordination Requirement. It is likely that in some cases, the two committers became somehow aware of the overlap and decided to avoid conflicts by having one of the two merge all changes and commit on behalf of both. Evidence of such a scenario may be recorded in the archived communications for the Mylyn project, which we intend to mine in the future.

All cases examined in DS1-a turned out to be false positives or negatives of the traditional CR detection method. More importantly, they highlight drawbacks of that method's reliance on post-mortem information and dependency conceptualizations. We conclude that proximity-based CR detection can be more accurate than existing methods.

RQ3: Does Proximity Provide Timely Detection of Coordination Requirements?

Finally, we set out to investigate whether proximity is an early indicator of CRs. The context events we use are, by their nature, antecedents to code commits. It is important to notice that, although the study described in this paper is retrospective, a method for analyzing CRs based on proximity and task contexts does not need to be. All

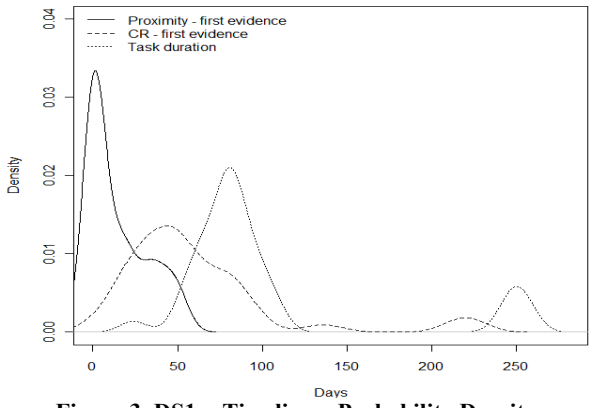

Figure 3. DS1-a Timeliness Probability Density

context events are recorded in real time. Therefore, a tool could gather context data as it is created by all developers and compute proximity on the fly. We discuss such a tool in Section 5. Here, our analysis focuses on how early proximity produces evidence of work dependencies between tasks. The earlier the evidence, the more actionable it is in supporting decisions aimed at resolving CRs as they form. For this analysis, we used the two data sets for which we have task context data associated with file changes (DS1-a and DS2). We considered all pairs of developers who present some CR and have granular proximity >0. There are 36 such pairs in DS1-a and 18 in DS2.

We obtain the time when the first contribution to the proximity score occurs by examining the timestamps for the first overlapping event recorded in all tasks contexts for the two developers for that release. We then compare the first proximity event with the first day of concurrent work by that pair during that release. For perspective, we also considered the day in which the first CR is identified for the same pairs. Fig. 3 shows the probability density functions of proximity detection, CR detection and task duration for data set DS1-a. The difference in the timeliness of recognition of work relationship shown is evident. Similarly distributed probability densities were seen in DS2. We found that in DS1-a the first evidence of proximity is detected on average 14.2 days after parallel work begins. In DS2, it takes 6.2 days. The first CR detection happens in DS1-a 60.7 days on average after the beginning of concurrent work by a pair (a delay of 46.5 days). In DS2, the first CR is detected 17.9 days after the concurrent work begins (11.7 days later). We also compared our findings with the duration of the concurrent work intervals by the same pairs in the various releases. In data set DS1-a, concurrent work intervals last 102 days on average, whereas in DS2, they last 31.4 days on average. The average "advance notice" provided by proximity is 87.8 and 25.2 days, respectively, showing that proximity significantly improves the timeliness of CR detection.

5. DISCUSSION

Our results suggest that CRs can be determined accurately based exclusively on the similarity of task contexts. Our proximity measure adequately models the presence and

intensity of CRs independently of any conceptualization of technical dependencies. Unlike methods which rely on data that is available after work has been completed, we rely on data that is accessible while development is underway.

Implications for Tools

Many tools exist for enhancing developer awareness, and several of them leverage information about CRs [1,8,9,23]. Many of these tools strive to identify all technical dependencies that exist in a software project and provide a comprehensive view of project coordination needs. Since our method provides a novel approach to the identification of CRs, it could be incorporated into those awareness tools and provide them with the benefits outlined above.

To explore our design implications, we developed a prototype that calculates proximity of tasks using the described algorithm. The tool leverages a shared central database which communicates with client components hosted within the individual IDEs and automatically stores context information for the team. Existing IDEs can push context to the database as events occur. This allows proximity relationships to be continuously updated as development is underway with no effort on the part of the developers. An early observation from using this prototype is that the use of selection events can allow for a proximity relationship to appear even before any code modification begins. For example, if a developer starts to consult source files that are likely involved in some task, she can be provided with a list of tasks (and developers) with high proximity. A tool using task context information and proximity is not only timelier, but it can also be richer and more accurate. It captures the entire working set involved in a task and follows the evolution of that working set throughout the task duration, tracking how heavily each artifact is actually used.

The timeliness and comprehensiveness that our proximity measure provides is not currently available in other awareness tools for software engineering. Palantír and CollabVS provide timely CR detection, but only observe a narrow subset of CRs and do not measure their severity. Our measure can alert developers of CRs before the majority of work has been completed. A tool using our measure could facilitate developer coordination much earlier in the development cycle. This could allow developers to negotiate design decisions and code changes to reduce technical dependencies. Early detection of CRs can significantly reduce the amount of rework required when conflicts are determined later in the development process and can even help avoid duplicate work.

The method described can also be incorporated in a decision support tool. For example, it can provide a dynamic view of Socio-Technical Congruence to inform real-time decisions on aspects like task assignment, scheduling, team composition, or design refactoring [27]. A further implication of our work is the ability to rank detected CRs based on the corresponding proximity scores. Previous methods of CR detection support rank only as a simple count of CRs occurring within a pair of developers [27]. Our rank is based instead on two components: the amount of weighted overlap in the working sets of the pair and the number of overlapping context events as compared to the average number across the population. This way of ranking CRs can help de-emphasize trivial development activities and their impact on coordination.

Finally, while the analysis presented in this paper focuses on detecting CRs between pairs of individuals, proximity can easily be applied at other levels such as tasks, projects or teams. This can be accomplished simply by aggregating context events appropriately. For example, to compute the proximity of teams, each team's working set would include all context events produced by all members of that team. This provides the ability for awareness tools to easily display coordination needs at different aggregation levels. This enables the construction of visualizations, dashboards or reports oriented towards a variety of roles including individual developers, team leaders and project managers.

Threats to Validity

A general caveat is that our findings derive from a single project. In this case study, the number of developers involved and the number of CRs are relatively moderate in size. Our findings should be corroborated by additional empirical studies to ensure that our approach works for other projects and projects of different scales. Another limitation can derive from performing our analysis at the release level. When considering concurrent work at finer-grained temporal units, the outlook on CRs and/or proximity may differ. To properly investigate how sensitive our findings are with respect to this issue, a project with a high density of CRs would make for the best follow-up case study. Another limitation, as mentioned earlier, is that we have considered any level of proximity >0 as an indicator of possible CRs. A sensitivity analysis is the next logical step.

Finally, there may be issues of repeatability. Although Mylyn is widely adopted in open source as well as industrial settings, its consistent use by all developers during all of the project activities is not guaranteed. A commercial version of Mylyn (Tasktop Dev) is available for the Visual Studio IDE. Other commercial-grade facilities (such as Cubeon for the Netbeans IDE) offer similar features, but we cannot assume that they record the exact same IDE interactions. However, it is likely that any facility similar to Mylyn can provide the data our proximity algorithm needs which is – at a minimum – data about the developer, task, timestamp, filename and path for artifact selection and edit events.

6. CONCLUSION

We introduce a proximity relationship that can be used to infer CRs between developers as they form. We describe the algorithm for measuring proximity based on task context information. This context information details

activities of developers within their IDEs and is obtained using existing tools. We show that proximity provides an earlier indication of CRs and overcomes known drawbacks in current CR detection methods. Proximity is promising as a foundational measure for building more accurate and useful representations of developer coordination. These advances improve the quality and timeliness of management, design and team coordination decisions.

ACKNOWLEDGEMENT

Special thanks to Dave Berry for his help with the initial data mining, to Gail Murphy for initial discussions, advice and encouragement, and to Patrick Wagstrom for his feedback and suggestions. This work was partially supported by the NSF through grant no. CCF-0916891.

REFERENCES

1. Begel, A., Phang, K.Y., and Zimmerman, T. 2010 Codebook: Discovering and Exploiting Relationships in Software Repositories. Proc. ICSE 2010.
2. Brooks, F.P. 1995. The Mythical Man-Month: Essays on Software Engineering. Addison Wesley. Reading, MA.
3. Cataldo, M., Bass, M., Herbsleb, J., and Bass, L. 2007. On Coordination Mechanisms in Global Software Development. Proc. ICGSE 2007, 71-80.
4. Cataldo, M., Herbsleb, J.D., and Carley, K.M. 2008. Socio-Technical Congruence: A Framework for Assessing the Impact of Technical and Work Dependencies on Software Development Productivity. Proc. ESEM 2008, 2-11.
5. Cataldo, M., Mockus, A., Roberts, J. A., and Herbsleb, J. D. 2009. Software dependencies, work dependencies, and their impact on failures. IEEE Transactions on Software Engineering. 35, 6, 864-878.
6. Cataldo, M., Wagstrom, P.A., Herbsleb, J.D., and Carley, K.M. 2006. Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools. Proc. CSCW 2006.
7. Conway, M.E. 1968. How do committees invent. Datamation. 14, 4, 28-31.
8. de Souza, C.R., Quirk, S., Trainer, E., and Redmiles, D.F. 2007. Supporting collaborative software development through the visualization of socio-technical dependencies. Proc. of the 2007 international ACM conference on Supporting group work. 147-156.
9. Dewan, P. and R. Hegde. 2007. Semi-Synchronous Conflict Detection and Resolution in Asynchronous Software Development. Proc. E-CSCW 2007. p. 159-178.
10. Dourish, P., and Bellotti, V. Awareness and Coordination in Shared Workspaces. Proc. CSCW 1992: p. 107-114.
11. Ehrlich, K., Helander, M., Valetto, G., Davies, S., and Williams, C. 2008. An analysis of congruence gaps and their effect on distributed software development. Proc. STC 2008.
12. Gall, H., Hajek, K., and Jazayeri, M. 1998. Detection of logical coupling based on product release history. Proc. ICSM 1998.
13. Grinter, R.E., Herbsleb, J.D., and Perry, D. E. 1999. The geography of coordination: dealing with distance in R&D work. Proc. of the international ACM SIGGROUP conference on Supporting group work.
14. Herbsleb, J.D. and Grinter, R.E. 1999. Splitting the organization and integrating the code: Conway's law revisited. Proc. ICSE 1999, 85-95.
15. Herbsleb, J.D., Mockus, A., and Roberts, J.A. 2006. Collaboration in software engineering projects: A theory of coordination. Proc. ICIS 2006.
16. Kersten, M. and Murphy, G.C. 2005. Mylar: a degree-of-interest model for IDEs. Proc. AOSE 2005, 159-168.
17. Kersten, M. and Murphy, G.C. 2006. Using task context to improve programmer productivity. Proc. FSE 2006.
18. Kraut, R. and Streeter, L. 1995. Coordination in software development. Communications of the ACM. 38, 3, 69-81.
19. Minto, S. and Murphy, G.C. 2007. Recommending emergent teams. Proc. MSR 2007.
20. Parnas, D.L. 1972. On the criteria to be used in decomposing systems into modules. Communications of the ACM. 15, 12, 1058.
21. Rothlisberger, D., Nierstrasz, O., Ducasse, S., Pollet, D., and Robbes, R. 2009. Supporting task-oriented navigation in IDEs with configurable heatmaps. Proc. ICPC 2009, pp. 253–257
22. Sarma, A., Noroozi, Z., and van der Hoek, A. Palantír: raising awareness among configuration management workspaces. Proc. ICSE 2003.
23. Sarma, A., Maccherone,L., Wagstrom, P., and Herbsleb, J. 2009. Tesseract: Interactive visual exploration of socio-technical relationships in software development. Proc ICSE 2009, 23-33.
24. Singer, J., Elves, R., and Storey, M.-A. 2005. Navtracks: Supporting navigation in software maintenance, Proc. ICSM 2005, pp. 325–335.
25. 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.
26. Sullivan, K.J., Griswold, W.G., Cai, Y., and Hallen, B. 2001. The structure and value of modularity in software design. Proc. FSE 2001, pp. 99-108.
27. Valetto, G., Chulani, S., and Williams, C. 2008. Balancing the value and risk of socio-technical congruence. Proc. STC 2008.