This is a post-peer-review, pre-copyedit version of this article. The final authenticated version is available online at: https://doi.org/10.1145/2491411.2491440

Do All Task Dependencies Require Coordination? The Role of Task Properties in Identifying Critical Coordination Needs in Software Projects

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

Daniela Damian
Software Engineering Global Interaction Lab
University of Victoria
Victoria, BC, Canada
danielad@cs.uvic.ca

ABSTRACT
Several methods exist to detect the coordination needs within software teams. Evidence exists that developers' awareness about coordination needs improves work performance. Distinguishing with certainty between critical and trivial coordination needs and identifying and prioritizing which specific tasks a pair of developers should coordinate about remains an open problem. We investigate what work dependencies should be considered when establishing coordination needs within a development team. We use our conceptualization of work dependencies named Proximity and leverage machine learning techniques to analyze what additional task properties are indicative of coordination needs. In a case study of the Mylyn project, we were able to identify from all potential coordination requirements a subset of 17% that are most critical. We define critical coordination requirements as those that can cause the most disruption to task duration when left unmanaged. These results imply that coordination awareness tools could be enhanced to make developers aware of only the coordination needs that can bring about the highest performance benefit.

Categories and Subject Descriptors
D.2.9 [**Software Engineering**]: Management – *productivity*, *programming teams*

General Terms
Management, Performance, Human Factors.

Keywords
Task Dependencies; Proximity; Coordination Requirements; Awareness; Collaborative Software Development; Machine Learning

1. INTRODUCTION
In large software projects, developers work on tasks in parallel or on interdependent tasks. This often results in work dependencies and, consequently, coordination needs. When developers remain unaware or do not obtain timely awareness of the coordination that is required to manage work dependencies, there is potential for software productivity or quality problems [6, 22]. Existing techniques to support coordination awareness assume all dependencies may require coordination, and they simply enumerate the universe of those potential coordination needs in a project. This can lead to an overwhelming number of recommendations and alert developers of even trivial coordination needs. This is especially problematic in large projects or when coordination requirement detection occurs at fine granularity, for example, at the level of individual tasks.

The main premise of our work, instead, is that not all coordination requirements are created equal. We explore what properties of development tasks and corresponding software code may indicate a critical coordination need. We define a critical coordination requirement as one that can cause the most disruption and inefficiency to the development process if not properly and timely managed. An understanding of the characteristics of the critical coordination needs can lead to better awareness tools and more focused coordination efforts in software development teams.

In this paper, we report on an exploratory case study of the Mylyn open source project with a large set of potential coordination requirements. We first evaluated a quantitative conceptualization of work dependencies called Proximity [4] and confirmed that existing automated techniques for the identification of coordination needs, like Proximity, find far too many dependencies. We then examined other task properties that could be used to supplement measures like Proximity to identify coordination needs. Finally, we used machine learning with both Proximity and the identified task properties to evaluate whether such an approach is successful in identifying only the critical coordination needs. The tasks involved in the reduced set of coordination requirements presented very different performance profiles from the rest of the tasks when examining task durations. These differences may be attributed to the criticality of the coordination needs among those tasks especially since many seem to be unrecognized and unmanaged. Our results imply that not all coordination requirements are created equal and current techniques for detecting coordination requirements could be supplemented by additional task properties beyond just work dependencies to better predict only the critical coordination requirements.

In the remainder of the paper, we first discuss related work (Section 2) and review the Proximity algorithm (Section 3). We then introduce our research questions and describe the setting of

ESEC/FSE'13, August 18–26, 2013, Saint Petersburg, Russia.

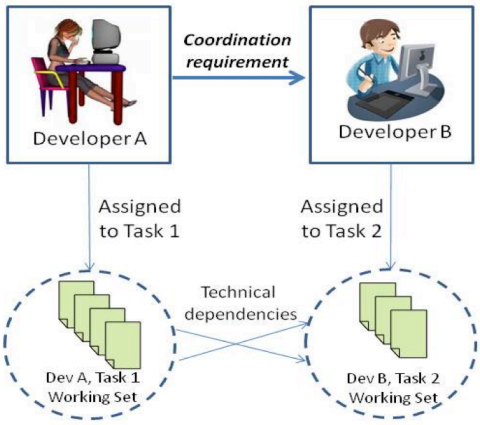

Figure 1. Conceptualization of coordination requirements in Cataldo et al. approach [7,9].

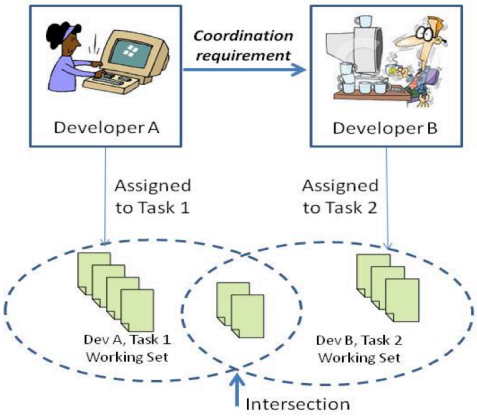

Figure 2. Conceptualization of Coordination Requirements through Proximity [4].

our empirical study (Section 4). Next, we describe the method and results of our research (Section 5). Finally, we discuss the significance of our contributions (Section 6) and offer some concluding remarks (Section 7).

2. BACKGROUND AND RELATED WORK

Several techniques have been proposed to detect the need to coordinate between developers in large software development projects [4,7,9]. Cataldo et al. [7,9] were the first to introduce a framework for establishing coordination requirements between developers (depicted in Figure 1). They found that when coordination requirements are fulfilled, for example, by acts of communication, productivity is likely to increase [7,8,9]. Their framework establishes coordination requirements between developers who are working on dependent tasks. To ascertain work dependencies they look at the artifacts committed during each task and the dependencies between those artifacts. Logical coupling [15] is used to determine technical dependencies between artifacts if they have been checked in together in the past. Cataldo et al. [8] found that logical couplings are more likely than syntactic couplings to provide a reliable representation of technical dependencies for their coordination requirement conceptualization. A limitation of this conceptualization is that it requires mining the source control repository of the project for the commit history of software artifacts. This type of data is typically available only towards the end of the development work for a task, and the coordination awareness garnered from this approach may not be actionable by the developers at the time coordination is needed to reap those performance benefits.

In Blincoe et al. [4], we proposed an alternative conceptualization of work dependencies for detecting coordination needs. That conceptualization uses the *Proximity* metric, which is described in detail in Section 3. Proximity (as shown in Figure 2) evaluates the intersection of the working sets of a pair of developers and the actions developers take on the artifacts in those working sets to determine and weigh coordination needs. We found that by looking at both artifact consultation and editing actions, it is possible to accurately discover coordination requirements without the need to model and consider technical dependencies between artifacts. Since artifact consultation and edit actions can be captured in real-time through existing IDE monitoring facilities [11,17,18,25] Proximity can provide timely detection of coordination needs. This timely detection of coordination needs provides awareness [14] to developers while their work is still underway. Developers can then act upon and resolve their coordination needs as they surface.

Current coordination requirement detection methods, including Proximity, abstract coordination requirements by detailing only pairs of developers who may need to coordinate. Developers may work on multiple tasks at the same time, so coordination requirements at the developer level may encompass the work dependencies of many tasks (see Figure 3). The many existing awareness tools [3,12,13,16,21,23,24] that exist to support developer awareness of coordination needs do not indicate which tasks are involved in coordination requirements. This puts the burden on the developers to identify what to coordinate about. If awareness tools were able to provide finer-grained coordination needs at the task level, that burden would be removed.

In this work, we extend the Proximity technique to identify coordination requirements between pairs of tasks rather than pairs of developers. However, without the abstraction that occurs when rolling coordination requirements up to the developer level, the work dependencies may signal a plethora of coordination requirements including those that are trivial or insignificant. For this reason, we augment the Proximity method by also including other task properties to detect only the most critical coordination requirements.

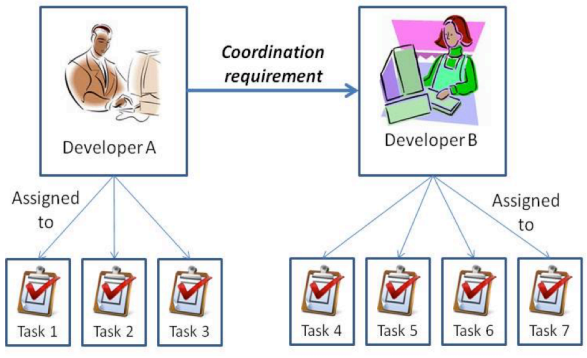

Figure 3. Coordination requirements between developers are typically a result of their work on more than one development task.

3. PROXIMITY

Proximity is a metric for measuring coordination needs in software development teams. Unlike more traditional coordination requirement detection techniques, it does not obtain information from the source control repository system nor rely on technical dependencies between artifacts. These differences make Proximity timely and turn coordination requirements into an actionable concept for managing coordination in software projects.

To determine coordination requirements, the Proximity algorithm examines the similarity of artifact working sets as they are constructed during developers' tasks. To do this, it obtains developer actions such as artifact consultation or edits as they occur. It uses the Mylyn framework [17, 18] to obtain this information. Mylyn is a tool that transforms a developer's Individual Developer Environment (IDE) to a task-centric view to make context switching between tasks easier. To fulfill its own purposes, Mylyn records all developer IDE interactions as they occur. These events are stored as context data for the task in focus. For convenience, ProxiScientia [5], the tool which implements the Proximity measure, is built on top of Mylyn so it can easily obtain these developer actions.

The Proximity measure looks at artifact consultation and modification activities captured by Mylyn and weighs the overlap that exists between the working sets associated to pairs of developers. It considers all actions recorded for each artifact in each working set in order to apply a numeric weight to that artifact's Proximity contribution. Weights are applied based on the type of overlap where the most weight is given when an artifact is edited in both working sets (weight = 1) and the least amount of weight is given when an artifact is simply consulted in both working sets (weight = 0.59). When an artifact is edited in one working set and consulted in the other working set, we consider this a mixed overlap (weight = 0.79). The weights are directly based on the weights Mylyn itself uses for its degree-of-interest model [17, 18]. Figure 4 illustrates an example of the Proximity computation process [4]. The algorithm computes the ratio of actual to potential overlap. Actual overlap considers the intersection of the two working sets while potential overlap considers the union of the two working sets. Potential overlap represents the maximum possible Proximity score had there been perfect overlap between the two sets of actions. Proximity scores can then be scaled based on the number of overlapping events to place greater weight on complex tasks that are likely to require coordination. Proximity scores range from zero to infinity. Through empirical analysis, we found that higher Proximity scores are indicative of a stronger need to coordinate [4].

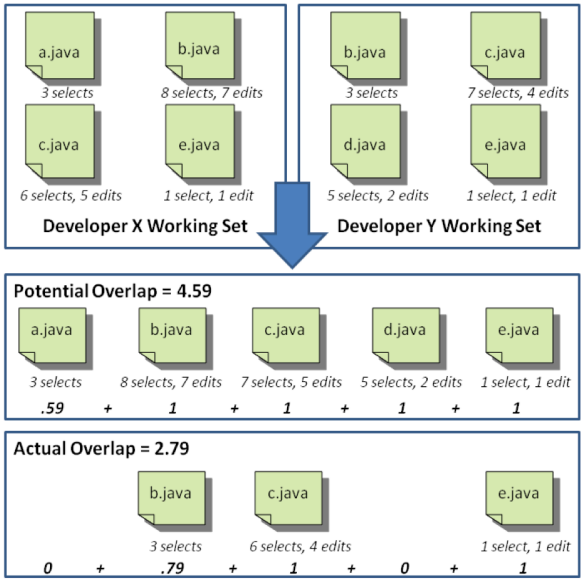

Figure 4. Proximity Algorithm Example [4].

4. RESEARCH APPROACH

We build on the Proximity method since it is the only existing real-time coordination requirement detection method. Conceptually, Proximity can be easily applied to pairs of tasks simply by aggregating the captured developer actions at the individual task level rather than at the developer level. However, even in moderately sized projects, a large number of potential coordination needs could be created when calculating Proximity between tasks. It is currently not possible to know whether all work dependencies that are detected between task pairs require actual coordination. Our working hypothesis is that current coordination requirement detection algorithms cast too wide a net in considering all work dependencies as candidates for coordination. We explore this hypothesis by comparing the tasks pairs with Proximity to the dependencies identified by the team:

RQ1: Is there a correspondence between tasks with identified dependencies and development tasks with Proximity?

To answer this research question, we look for evidence of task dependencies that have been identified by the project team and recorded within a change request. Change request repositories, like Bugzilla[1], are commonly used to define, assign and manage project tasks. In the remainder, we refer to the task dependencies established by the project team and recorded in the change request repository as the "identified dependencies". We then compare those identified dependencies to the Proximity scores computed between tasks. We expect to find high levels of recall paired with low levels of precision indicating that while Proximity is able to successfully detect many identified dependencies, the identified dependencies represent only a small subset of the task pairs with Proximity. If this hypothesis is proven correct, research questions RQ2 and RQ3 will begin to explore solutions for this problem.

RQ2: What properties of task pairs, other than work dependencies ranked by Proximity, are also indicative of actual coordination needs?

We examine various properties of task pairs to look for differences between the identified dependencies and all other task pairs. We inspect the statistical difference in proportion and distribution of these properties using Chi-squared and Mann-Whitney tests.

RQ3: Can we supplement Proximity with additional task pair properties to identify the most critical coordination needs?

This research question builds on the findings of RQ2. If there are properties beyond work dependencies ranked by Proximity that greatly differ between tasks with identified dependencies and all other tasks, we can use those properties with machine learning to better infer coordination requirements and to supplement current methods like Proximity. We conceptualize critical coordination requirements as those coordination needs that have suffered the most in terms of performance (task duration) and use this criterion in analyzing the output of such a machine learning technique on

[1] http://www.bugzilla.org

our data set. Therefore, to answer RQ3, we evaluate the results of the machine learning approach comparing the task performance of the tasks involved in machine learning predicted coordination requirements and all other tasks.

To answer our three research questions, we carried out an empirical study on the Mylyn open source project itself. The Mylyn project represents an ideal case study because its developers make routine use of the Mylyn plugin in their IDE allowing us to collect the context data needed for the Proximity calculation. We mined the project repositories and collected all Bugzilla change requests and developer activities (Mylyn context data) from two releases of the Mylyn project, releases 3.1 and 3.2. On the Mylyn project, developers are assigned change requests as their unit of work and encouraged to deliver their work as code patches that correspond to (and resolve) a single change request. The bug tracking database is, therefore, the way the Mylyn team defines and assigns developer tasks, and we refer to Bugzilla change requests as tasks.

Mylyn release 3.1 spanned from June 2008 to March 2009. At the time we mined the repository, in July 2012, we obtained 512 Bugzilla tasks with context data for which development work occurred during the development of release 3.1. This yielded 130,816 task pairs for that release. Similarly, Mylyn release 3.2 spanned from March 2009 to June 2009 and contained 251 tasks (31,375 task pairs). Our analysis focused on release 3.2. Release 3.1 was used as a training data set for the machine learning technique that we apply to address RQ3.

5. RESEARCH METHOD AND RESULTS

5.1 Applying Proximity to Tasks

RQ1: Is there a correspondence between tasks with identified dependencies and development tasks with Proximity?

To answer RQ1, we mined the project's change request database, Bugzilla, to obtain evidence of work dependencies between submitted change requests and the tasks associated with those change requests that have been identified by the project team. A recent study by Aranda and Venolia [1] found that repositories like Bugzilla often provide incomplete information because of omission, oversight, or simply because of project conventions. For this reason, we sought dependencies beyond those explicitly marked by the project team. By inspecting the change request reports, we found three main types of dependency identification evidence: explicitly marked dependencies, duplicates, and discussion cross-references.

Explicitly marked: These dependencies appear in the 'Depends On' and 'Blocks' fields in the Bugzilla database. The tasks listed in the 'Depends On' field of a change request report must be resolved before the task associated with that change request can be resolved. Conversely, the task associated with the current change request must be completed before the tasks listed in the 'Blocks' field can be resolved. In this dataset, this is always a reciprocal relationship. If task one 'Depends On' task two, task two will also 'Block' task one. These types of dependencies are marked between 33 task pairs in release 3.2.

Duplicates: These dependencies exist when one task is marked as a duplicate of another task. The project team realizes the tasks are performing overlapping work and close one task, so the remaining work can be completed jointly in the remaining task. There are only two duplicate pairs in our dataset in release 3.2.

Discussion cross-references: Change requests are cross-referenced in the discussion of another change request. For example, a part of the implementation for change request #274790 is reverted when it is decided that a better solution would be possible after the completion of change request #278708. This is an example of a work dependency that is discussed in the Bugzilla comments. A similar discussion occurs on change request #235439 where it is mentioned that part of the implementation must be completed after change request #211096. These task pairs may also be marked explicitly as dependencies, but that is not always the case. In fact, neither of the above examples is explicitly marked as a dependency within their Bugzilla records. There are 21 pairs with cross-references in release 3.2.

These three types of dependencies established by the project team are the identified dependencies. We compared the identified dependencies with measures of Proximity for those same pairs of development tasks. In release 3.2, we have 1,468 task pairs with a Proximity score >0 (4.7% of 31,375 pairs). The average Proximity score for all task pairs with Proximity >0 is 1.2 with a maximum score of 79.43. The average for the 39 task pairs with Proximity and an identified dependency is higher with an average of 3.3. A Mann-Whitney test shows that these task pairs with identified dependencies have significantly higher Proximity scores than other task pairs with Proximity ($W = 40705.5$, p-value = 4.165e-07). This indicates that these dependencies are also ranked well by the Proximity metric. In addition, a larger number of tasks with identified dependencies also have Proximity scores >0 (as shown in Table 1). However, while recall is high, the number of task pairs with Proximity is much greater than the number of identified dependencies resulting in low precision.

As we expected, not all task pairs with high Proximity were marked as dependencies by the development team. One reason for this could be that the Mylyn team found other dependencies during development but they were not all reported within Bugzilla. To shed light on this, we contacted a lead Mylyn developer to attempt to gain insight on the type of dependencies that are captured within their Bugzilla repository. He told us that the Mylyn team uses *"Bugzilla's 'Depends on' field to track subtasks"* of some coarser-grained umbrella task. The same developer also mentioned that there is often a dependency between subtasks of the same umbrella task stating that *"often subtasks need to be completed in a certain order"*, but those dependencies are usually not marked explicitly. The same developer also mentioned that task pairs that are marked with the same tag in Bugzilla *"often share subtasks or are directly linked as subtasks"*.

The insight obtained from the lead developer provides two additional dependency identification types that we had not included in our analysis. In release 3.2, we found 183 pairs of subtasks of the same umbrella task and 352 pairs that share the same tag. Over 40% of these two types of dependency relationships have Proximity > 0, but very few are marked as being dependent in the Bugzilla change reports. However, since

Table 1. Precision/Recall: Identified Dependencies vs Proximity

Dependency	Precision	Recall
Explicitly Marked	26 of 1468 (1.8%)	26 of 33 (78.8%)
Duplicates	2 of 1468 (0.14%)	2 of 2 (100%)
Discussion cross-referenced	18 of 1468 (1.2%)	18 of 21 (85.7%)
Total	39 of 1468 (2.7%)	39 of 49 (79.6%)

we cannot assume with confidence that sharing the same umbrella task or the same tag is a purposeful indication of a dependency relationship by the Mylyn team, we did not include these types of evidence as part of the identified dependencies set for our analysis. We note them here simply to illustrate the incompleteness of the Bugzilla database in terms of dependencies marked between tasks.

While the incompleteness of the dependency information in the Bugzilla database certainly contributes to the low precision scores, it is also likely that Proximity, when applied at the task level as opposed to the developer level, signals coordination needs between too many task pairs. When considering only Proximity scores for detection of coordination requirements, a large majority of the tasks in Release 3.2 (234 out of 251) would be involved in at least one coordination requirement with other tasks, which adds to the suspicion that current methods may cast too wide a net.

In answering RQ1: The low precision indicates that Proximity casts too wide a net. Therefore, we looked for additional task properties that also indicate coordination requirements (RQ2) and could be used to refine the recommendations of the Proximity algorithm to detect only the critical coordination needs (RQ3).

5.2 Analysis of Task Pair Properties

RQ2: What properties of task pairs, other than work dependencies ranked by Proximity, are also indicative of actual coordination needs?

To answer this research question, we examined task pair properties to look for differences between identified dependencies and all other task pairs. The task properties we examined include (1) architecture-related properties directly available from the project's change request database such as: the affected product, component, platform and operating system of the task and (2) modularity characteristics of the software artifacts involved in each task.

We examined the *architecture-related properties* by checking, for each task pair, if the tasks involved in that pair shared any of those properties (i.e. if they affect the same product, component, platform, or operating system). A Chi-squared test of difference in proportion for each of these properties shows that there is a significant difference between the identified dependencies and all other task pairs for all but one of the tested properties: there is not a statistically significant difference for the number of task pairs marked for the same component (results shown in Table 2).

To *characterize the software artifacts* involved in each task, we derived a Design Rule Hierarchy (DRH) [27] of the Mylyn code base for the two releases of interest. DRHs are computed from Design Structure Matrices (DSMs) [2]. A DRH assigns software artifacts to modules based on technical dependencies within the code. Consistent with Parnas' definition of modularization [22], these independent modules can be worked on in parallel without incurring coordination overhead. Considering the number of overlapping modules for each task pair allows us to quantify the amount of dependencies between the two tasks.

A DRH also clusters modules into "layers" where each layer depends only on the layers above. These layers can be used to differentiate artifacts that represent influential design decisions from low-level artifacts that depend on (changes to) those decisions. Wong et al. [27] observed that developers working on tasks that involve software modules in different layers of a DRH tend to communicate (a dominant form of coordination in

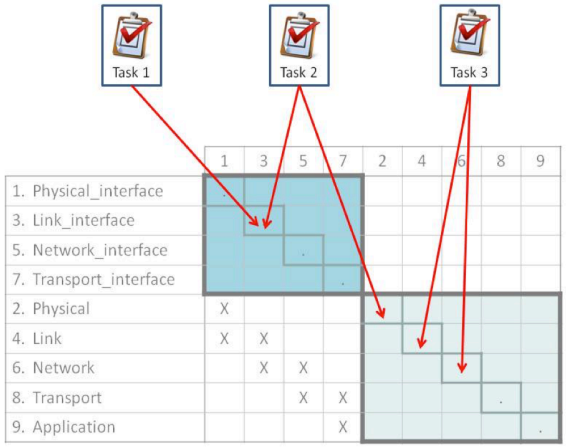

Figure 5. Design Rule Hierarchy Example [27].

software development [19]) significantly more than developers working only on modules in the same layer. The number of overlapping layers allows us to identify when task pairs are operating on similar areas of the code hierarchy.

For illustration purposes, Figure 5 shows an example of a hypothetical two-layer DRH. The large thick-bordered boxes represent the two different layers while the boxes within the layers represent modules. The figure shows three different tasks operating on the artifacts. Tasks 1 and 2 have one overlapping layer and one overlapping module. Tasks 2 and 3 have one overlapping layer and no overlapping modules. Tasks 1 and 3 have no overlapping layers or modules.

The Mylyn Project DRH of release 3.1 consists of 11 layers and 671 modules. The release 3.2 DRH consists of 11 layers and 786 modules. We identified the associated DRH layer and module for each artifact consultation and edit action associated with java artifacts for each task. Using this information, we obtained the number of overlapping layers and modules for each task pair.

Table 2. Task Property Comparison

Property	Identified Dependencies	Other Task Pairs	Chi-squared Test
Task Pair Count	49	31,326	–
# with Proximity	39	1,428	$\chi^2 = 617.96$ $p < 2.2e\text{-}16$
# in the same product	40	21,104	$\chi^2 = 4.5$ $p = 0.03$
# in the same component	37	20,636	$\chi^2 = 2.0$ $p = 0.15$
# in the same platform	38	14,960	$\chi2 = 17.4$ $p = 3.001e\text{-}05$
# for the same OS	37	11,939	$\chi2 = 29.0$ $p = 7.214e\text{-}08$
			Mann-Whitney Test
Mean Overlapping Layers	1.29	0.86	$W = 971716$ $p = 0.0002$
Mean Overlapping Modules	1.57	0.33	$W = 1243704$ $p < 2.2e\text{-}16$

We then analyzed each of these properties to identify any that appear significantly different between the identified dependencies and all other task pairs. A Mann-Whitney test of difference in distribution shows that the difference is statistically significant for both of these properties (results shown in Table 2).

In answering RQ2: We determined the following set of task pair properties that differentiate task pairs with identified dependencies from all other task pairs:

- Within same product
- Within same platform
- Within same operating system
- Number of overlapping DRH layers
- Number of overlapping DRH modules

5.3 Applying Machine Learning to Proximity and Identified Task Properties

RQ3: Can we supplement Proximity with additional task pair properties to identify the most critical coordination needs?

RQ2 provided confirmation that there are properties that differ with statistical significance in task pairs with identified dependencies and, therefore, are indicative of coordination requirements. To answer RQ3, we explored supplementing the Proximity algorithm with these properties to infer the most critical coordination needs by applying the k-nearest neighbor machine learning algorithm [10].

To analyze the results of the machine learning approach, we computed precision and recall against the identified dependencies as they are the best available approximation of ground truth (section 5.3.2). We scrutinized the cases of false positives (section 5.3.3) and false negatives (section 5.3.4) of the machine learning algorithm to determine if they are truly falsities of the machine learning algorithm or rather a result of an incomplete picture of ground truth for coordination requirements. An in-depth analysis of the false positives and false negatives revealed that the machine learning outcomes are a more reliable indication of coordination requirements than the identified dependencies. Next, to determine if the machine learning algorithm identified the critical coordination requirements, we examined the task performance between the tasks with coordination requirements and tasks without coordination requirements as predicted by the machine learning algorithm (section 5.3.5). We found strong statistical results that the machine learning algorithm is able to find the critical coordination requirements when compared to the coordination requirements found using only Proximity.

In the remaining analysis, we use the following terms to describe the different sets of task pairs:

- Identified Dependencies: task pairs that have been either marked as dependent or duplicate or cross-referenced in the discussion within their Bugzilla change request reports.
- Coordination requirements: task pairs that have been detected by the machine learning algorithm as needing coordination.
- Recognized coordination requirements: coordination requirements that are also identified dependencies.
- Unrecognized coordination requirements: coordination requirements that are not identified dependencies.

5.3.1 Machine Learning Approach

The k-nearest neighbor algorithm considers the distance from an unknown pair to each of the pairs in the training set. It then considers a majority vote from the k-nearest neighbors in the training set to decide if the unknown pair is a coordination requirement or not. For this study, we used nine as the k-value. Euclidean distance was used to determine the distance between the unknown pairs and the training set instances. We used the properties determined to have statistical significance in RQ2 to calculate the distance between instances.

As a training set, we used a subset of task pairs from release 3.1. The task pairs with identified dependencies were the positive examples of coordination requirements. On the other hand, we selected a subset of 175 task pairs that were not identified dependencies as the negative examples in the training set.

5.3.2 Evaluating Precision and Recall

After training the machine learning algorithm with the data from release 3.1, we applied it to release 3.2, and we were able to significantly reduce the number of predicted coordination requirements. Proximity identified 1,468 coordination requirements, whereas machine learning predicted only 244 coordination requirements, a reduction of about 83%. This caused precision to increase almost four-fold. Conversely, recall decreased due to missing an additional 17 identified dependencies. The differences in precision and recall of the two approaches are shown in Table 3. The identified dependencies, however, do not represent a complete picture of ground truth regarding coordination requirements since we have found them to be incomplete and heavily weighted towards one particular type of dependency (task decomposition).

5.3.3 Evaluating False Positives

Precision of 9% is still quite low. However, we hypothesized that the low precision is a result of a lack of complete data in the identified dependencies, and it is not indicative of the promise of our approach. To verify this, we performed an in-depth examination of the content of change request reports for some of the 222 unrecognized coordination requirements. We looked for evidence of other dependency relationships where awareness would have helped the productivity of the team. One example is the task pair of #233158 and #278494. Task #278494 is opened two days after the patch for #233158 is committed, and it notes an issue that originated from the committed code for #233158. A team member who does not appear to be aware of task #233158 creates the change request and submits a patch to fix the issue. Shortly after the patch is submitted, the assignee of the first task becomes aware of the issue and, having more expertise on the original task, suggests a different implementation.

The above is a clear example where early coordination could have prevented additional work. It highlights how there are certainly other types of dependencies that exist between tasks that are

Table 3. Precision/Recall: Identified Dependencies vs Coordination Requirements

Method	Precision	Recall
Proximity Only	39 of 1,468 (2.7%)	39 of 49 (79.6%)
Machine Learning	22 of 244 (9.0%)	22 of 49 (44.9%)

Table 4. Properties for Recognized Coordination Requirements, Unrecognized Coordination Requirements, and No Coordination Requirements

	Recognized Coordination Requirements	Unrecognized Coordination Requirements	No Coordination Requirements
Number of Task Pairs	22	222	31,131
Number with Proximity	22 (100%)	212 (95.5%)	1234 (4%)
Number within same product	22 (100%)	209 (94.1%)	20,913 (67.2%)
Number within same platform	18 (81.8%)	140 (63.1%)	16,539 (53.1%)
Number for same OS	17 (77.3%)	132 (59.5%)	15,154 (48.7%)
Average Proximity	5.33	5.23	0.02
Mean # of Overlapping DRH Layers	1.73	1.99	0.85
Mean # of Overlapping DRH Modules	2.45	2.48	0.31

missing from the Bugzilla records, but which our machine learning technique successfully identified. This also suggests that the precision/recall measures are not an accurate evaluation of the ability of our method to recognize coordination requirements.

For this reason, we conducted a more in-depth and relevant validation of the machine learning outcomes. First, we examined the differences between the unrecognized coordination requirements and the recognized coordination requirements. Table 4 shows the properties of interest for the unrecognized coordination requirements and, for comparison, both the recognized coordination requirements and the task pairs without coordination requirements. We also performed Chi-squared and Mann-Whitney tests comparing the properties of the unrecognized coordination requirements to the two other groups. The tests found that there are no statistical differences between any of the properties when comparing the unrecognized and recognized coordination requirements. In contrast, there is a large and significant difference between the unrecognized coordination requirements to the task pairs without coordination requirements (see Table 5). While this is not surprising given that the goal of the k-nearest neighbor algorithm is to pick out the best possible matches on these properties, this shows that while we only have a small set of identified dependencies, we have additional task pairs which look remarkably similar to those identified dependencies. This suggests that the machine learning approach successfully identifies coordination requirements that were missed by the team or, perhaps, were tracked in another way besides the use of the dependency, duplicate and cross-reference relationships.

5.3.4 Evaluating False Negatives

In order to identify possible reasons for the exclusion of the identified dependencies that were not established as coordination requirements by the machine learning algorithm, we analyzed the differences between the identified dependencies that were

Table 5. Chi-squared and Mann-Whitney Tests Comparing Unrecognized Coordination Requirements to other groups

Unrecognized Coordination Requirements Compared to	Recognized Coordination Requirements	No Coordination Requirements
Number with Proximity	$\chi2 = 0.52$ $p = 0.47$	$\chi2 = 107.23$ $p < 2.2e\text{-}16$
Number within same product	$\chi2 = 0.26$ $p = 0.61$	$\chi2 = 1349.47$ $p < 2.2e\text{-}16$
Number within same platform	$\chi2 = 1.04$ $p = 0.31$	$\chi2 = 4134.67$ $p < 2.2e\text{-}16$
Number for same OS	$\chi2 = 0.63$ $p = 0.43$	$\chi2 = 2051.28$ $p < 2.2e\text{-}16$
Proximity score	$W = 1471$ $p = 0.57$	$W = 318375$ $p < 2.2e\text{-}16$
Number of Overlapping DRH Layers	$W = 1486.5$ $p = 0.56$	$W = 1945548$ $p < 2.2e\text{-}16$
Number of Overlapping DRH Modules	$W = 1775$ $p = 0.09$	$W = 672795$ $p < 2.2e\text{-}16$

Table 6. Properties for Identified Dependencies

	Identified Dependencies with Coordination Requirements	Identified Dependencies without Coordination Requirements	Other No Coordination Requirements
Number of Task Pairs	22	27	31,104
Number with Proximity	22 (100%)	17 (63%)	1217 (3.9%)
Number within same product	22 (100%)	18 (66.7%)	20,895 (67.2%)
Number within same platform	18 (81.8%)	17 (63%)	16,522 (53.1%)
Number for same OS	17 (77.3%)	16 (59.3%)	15,138 (48.7%)
Average Proximity	5.33	0.42	0.02
Average Number of Overlapping DRH Layers	1.73	0.93	0.85
Average Number of Overlapping DRH Modules	2.45	0.85	0.31

detected as coordination requirements and the remaining identified dependencies not detected as coordination requirements.

As a baseline, we also compared the identified dependencies not selected as coordination requirements to all other task pairs that are not coordination requirements. Table 6 shows the properties of these groups. Tables 7 and 8 report the Chi-squared and Mann-Whitney tests for these groups showing that for many of the properties the identified dependencies without coordination

Table 7. Chi-squared tests comparing Identified Dependencies

Identified Dependencies without Coordination Requirements Compared to:	Identified Dependencies with Coordination Requirements	Other No Coordination Requirements
Number with Proximity	$\chi^2 = 10.24$ p = 0.001	$\chi^2 = 454.39$ p < 2.2e-16
Number within same product	$\chi^2 = 8.98$ p = 0.003	$\chi^2 = 6967.997$ p < 2.2e-16
Number within same platform	$\chi^2 = 2.11$ p = 0.15	$\chi^2 = 4369.94$ p < 2.2e-16
Number for same OS	$\chi^2 = 1.79$ p = 0.18	$\chi^2 = 4239.04$ p < 2.2e-16

requirements are significantly different from the other identified dependencies. The task pairs in this group are less likely to have Proximity, more likely to have lower Proximity scores, less likely to be in the same product, and less likely to have overlapping DRH modules and DRH layers. In addition, for the number of overlapping layers property there is not a statistically significant difference between these identified dependencies without coordination requirements and all other task pairs without coordination requirements. This could indicate that the identified dependencies established by the team are not always true work dependencies for this group of task pairs. Since many of the identified dependencies capture a task/subtask relationship in our dataset, we believe that some of the subtasks are not as closely related to the parent task, and actual work dependencies do not exist for some of the task/subtask relationships.

We, therefore, examined some of these 27 task pairs to determine the nature of the dependencies that exist between them. One example, change request #271019 is used simply to track the preparation of a maintenance release. Dependencies are created for each of the tasks that are to be included in the release. This is purely a management task to ensure the release does not occur until these necessary changes are completed. This accounts for five of the 27 task pairs. Analysis of the remaining task pairs showed that actual work dependencies exist between just two of the 27 task pairs, while the remaining 25 were similar management type relationships. This confirms that, although recall decreases when compared to the identified dependencies, our machine learning algorithm is successfully identifying the task pairs that truly require coordination.

5.3.5 Evaluating Task Performance

Since we define critical coordination requirements as those that suffer in terms of performance when unmanaged, we examined the task performance of the coordination requirements. We focus on task duration as our measure of task performance. We hypothesize that because many of the coordination requirements are unrecognized and therefore unmanaged, the task pairs with coordination requirements tend to have longer task durations [6,7,8,9,22]. We compare the outcome after machine learning to the outcome of the Proximity algorithm alone to determine if supplementing Proximity was able to identify the critical coordination needs.

We compared the task durations for tasks with coordination requirements to tasks without coordination requirements. Table 9 shows this comparison before the machine learning algorithm is applied when all task pairs with Proximity >0 are considered coordination requirements. We, again, observe that Proximity casts too wide a net. When computing coordination requirements in this way, there are very few tasks (7%) with no coordination needs. So while the 17 tasks without any coordination needs have shorter duration on average, no significance is found. The Chi-squared test results show there is no statistically significant difference between task durations. This shows that Proximity alone is not enough to identify the critical coordination needs.

Table 8. Mann-Whitney tests comparing Identified Dependencies

Identified Dependencies without Coordination Requirements Compared to:	Identified Dependencies with Coordination Requirements	Other No Coordination Requirements
Average Proximity	W = 528 p = 1.632e-06	W = 168784 p < 2.2e-16
Average Number of Overlapping DRH Layers	W = 439 p = 0.0008	W = 389642 p = 0.24
Average Number of Overlapping DRH Modules	W = 486.5 p = 1.763e-05	W = 239041.5 p = 3.096e-07

When we apply the machine learning algorithm the number of predicted coordination requirements is greatly reduced. In addition, the number of tasks that are involved in at least one coordination requirement is significantly lowered compared to the outcome of the Proximity algorithm. With this approach, 45% of tasks require no coordination. Table 10 shows performance measures after machine learning is applied. We now see a strong, significant difference in task duration. In addition, a Mann-Whitney test shows the task durations of the tasks involved in the coordination requirements detected when using our machine learning approach (Table 10) are significantly different than the task durations when considering Proximity alone (Table 9) where

Table 9. Performance: Proximity Only

	Coordination Requirements	No Coordination Requirements	Mann-Whitney Test
Number of Tasks	234	17	—
Average Task Duration	42.4 days	5.2 days	W = 2327 p = 0.12

Table 10. Performance: After Machine Learning

	Coordination Requirements	No Coordination Requirements	Mann-Whitney Test
Number of Tasks	139	112	—
Average Task Duration	52.38 days	24.4 days	W = 9742 p = 0.0003

W = 8238.5 and the p-value = 0.0006. The average task duration is now 52.38 days, almost 10 days longer than the average task duration when considering only Proximity. This shows that supplementing the Proximity algorithm using the machine learning techniques described in this paper allows for the detection of critical coordination needs when criticality is conceptualized as most likely to increase task duration.

In answering RQ3: We conclude that we can use machine learning to supplement Proximity with additional properties of task pairs to identify the most critical coordination needs. Our machine learning techniques reduced the set of potential coordination requirements identified by Proximity by 83%. The remaining 17% were found to be most critical when coordination criticality was conceptualized as most likely to cause disruption to task duration.

6. DISCUSSION

We described our exploratory investigation of techniques to identify critical coordination needs in a software project. Since developers often work on multiple tasks simultaneously, computing coordination needs between tasks provides developers better (more accurately scoped) awareness of where coordination is needed. Our research approach, therefore, computed coordination requirements between tasks rather than between developers, but we found that existing techniques for detecting coordination needs cast too wide a net when applied to pairs of tasks. We identified other task properties that can be used to supplement Proximity – our conceptualization of work dependencies – and used machine learning on those properties to identify the critical coordination needs. Our machine learning technique greatly reduced the number of detected coordination requirements and indicated the most critical when considering task duration. These results have several implications to both research and practice.

This is the first attempt to explore the possibility that differences exist within the universe of potential coordination requirements. We identified a set of properties of tasks that, when coupled with Proximity, can find the critical coordination needs. The properties we considered (or similar properties) are commonly available in most change request databases, which are frequently used tools for software projects. Therefore, these properties can be applied to the analysis of coordination needs in a wide variety of projects.

We have shown how code modularization properties that can be derived from the system DRH are also useful indicators of coordination needs. These findings build upon and reinforce previous empirical results that found that DRHs are adept at highlighting the intertwined relationships between issues of coordination and issues of modularity [27].

6.1 Implications for Tools

Existing awareness tools that detect coordination needs identify only the involved developers and do not provide scoped awareness by detailing the task dependencies involved in those coordination needs. This puts the burden on the developers to identify what to coordinate about. We envision a tool that could remove that burden by performing (semi-)automated recommendation and management of task dependencies. The tool could be based on a coordination requirement conceptualization like Proximity, which enables timely recognition of coordination needs as they form, and could be supplemented with machine learning techniques as described in this paper.

In addition, the tool could have a continuous learning component that would suggest candidate coordination requirements to project personnel. Project experts could confirm those candidates as either true positives or true negatives. This would allow the component to incrementally learn how to tease out the salient characteristics that indicate work dependencies for which coordination awareness is critical. Such a learning component would also be able to pick up and adapt to some of those characteristics that may be project-dependent in nature. The envisioned tool could also be adjusted to learn unobtrusively from coordination actions taken by the team within the tool itself (discussions, cross-referencing of task pairs, etc.). That is likely to improve the machine learning accuracy.

Such a tool could be used to automate task dependency management, provide coordination awareness both within and across teams, and support coordination among developers. To achieve this, the tool could provide a list of the most critical task dependencies tailored for each developer along with in-tool coordination mechanisms. The tool could allow developers to select a dependency to view details on the dependent task or highlight areas of code where overlap exists.

A management view of the tool could provide a view of the most critical coordination needs across the team. It could also suggest task assignments based on the premise of trying to minimize coordination needs and the related overhead. In addition, an architect view of the tool could highlight the areas of the code where the most overlap occurs indicating a possible need for refactoring.

After development of such a tool, user studies could be performed to validate our findings. We envision a study that would compare the task performance of a team prior to/after the introduction of the tool to examine whether the awareness provided by the tool improves task performance. In addition, the in-tool coordination mechanisms could be monitored to identify how often the developers follow-up on the coordination recommendations made by the tool.

6.2 Threats to Validity

The most significant threat to this study was the partial information in the Bugzilla repository available to represent ground truth about task dependencies. That issue impacts the training set that was used for machine learning and possibly its efficacy. Even more importantly, it complicates the evaluation of results since standard metrics such as precision and recall are not very meaningful in such an uncertain context. To overcome this limitation, we supplemented those metrics by conducting a qualitative analysis of many of the task pairs.

Another threat to validity is that our findings derive from a single case study with a relatively moderate number of developers and number of tasks. Our results could be affected by specificities of the case. For that reason, our findings should be corroborated by different case studies to ensure that our approach works across a spectrum of software projects of diverse scales.

Another issue is that we were limited in the number of task and code properties that we could investigate. There may be additional, or even better, properties that could be used to differentiate the overall set of potential coordination requirements and highlight the most important ones. In addition, all of the selected properties may not be portable across different bug tracking systems. This study should be repeated using projects hosted on different bug tracking systems to evaluate other available properties that characterize the architecture of a task like Trac's[2] component, Redmine's[3] category, and Jira's[4] components

[2] http://trac.edgewall.org

and labels. For the artifacts involved in the tasks, we focused on DRH properties. Among the slew of possible metrics and properties describing a project code base, we choose the DRH since it was conceived directly to analyze and segment a code base in modules that can be independently assigned to developers for parallel work.

Finally, we cannot exclude that our results could be caused by some other factors that underlie the properties we selected which we did not take into account. This threat is mitigated by the relatively large size and diversity of our data set.

6.3 Future Work

To continue this exploration, we plan to examine the potential of additional task properties beyond those identified in this paper, work to develop a better source of ground truth to use for evaluation purposes, and analyze the feasibility of the proposed approach to detect the critical coordination needs in real time.

We will examine additional task properties to search for further properties that may also be useful in supplementing Proximity to compute the critical coordination needs. We also plan to compare the identified properties to one another to understand which have the most predictive power.

To develop a better sense of the ground truth of coordination requirements in the project, we plan to perform content analysis of a random sampling of task pairs, and manually code their likeliness of having or not having coordination requirements.

To evaluate the feasibility of using the proposed methods on development tasks incrementally while development is under way, we also plan to develop a prototype of a tool that incorporates the algorithms described in this paper. We will run that prototype on pre-collected data from the Mylyn 3.2 release. The prototype will take all Mylyn context events and all Bugzilla data as input. The pre-collected data will be time-ordered and played back to mimic the real-time progression of development work and the live collection of the corresponding data. The prototype will receive each context event and Bugzilla update in a time-ordered series as they occurred. This will allow us to determine exactly when coordination needs can be established using our prototype and how actionable our method is.

7. CONCLUSION

The investigation of what work dependencies result in critical coordination requirements is a new line of research. We took a first step in distinguishing between work dependencies when detecting coordination needs to promote awareness. In this paper, we contribute and discuss a list of properties that help to characterize task pairs that require coordination, and we demonstrate how enhancing the Proximity algorithm with machine learning techniques helps in the selection of the most critical coordination needs. This initial exploration shows the promise in this line of research for detecting the most critical coordination needs. Such a method has implications for both change request management and triage support tools. Tools that incorporate and implement the techniques we described can increase coordination awareness among development teams and support more focused coordination efforts. More work is needed in this area to better understand the detailed nature of coordination requirements and the characteristics of task pairs that indicate when coordination is necessary and critical.

[3] http://www.redmine.org

[4] http://www.atlassian.com/software/jira

8. ACKNOWLEDGMENTS

This work was partially supported by the NSF through grants no. CCF-0916891 and VOSS OCI-1221254.

9. REFERENCES

[1] Aranda, J. and Venolia, G. 2009. The secret life of bugs: Going past the errors and omissions in software repositories. In Proc. ICSE 2009.

[2] Baldwin, C. Y. and Clark, K. B. 2000. Design Rules, Vol. 1: The Power of Modularity. MIT Press.

[3] Begel, A., Phang, K.Y., and Zimmermann, T. 2010. Codebook: discovering and exploiting relationships in software repositories. In Proc. ICSE 2010.

[4] Blincoe, K., Valetto, G. and Goggins, S. 2012. Proximity: a measure to quantify the need for developers' coordination. In *Proc.* CSCW 2012.

[5] Borici, A., Blincoe, K., Schröter, A., Valetto, G., and Damian, D. 2012. ProxiScientia: Toward Real-Time Visualization of Task and Developer Dependencies in Collaborating Software Development Teams. In Proc. CHASE 2012.

[6] Brooks, F.P. 1995. The Mythical Man-Month: Essays on Software Engineering. Addison Wesley. Reading, MA.

[7] 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. In *Proc.* ESEM 2008.

[8] 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*, vol.35, no.6, pp.864-878, Nov.-Dec. 2009.

[9] 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. In Proc. CSCW 2006

[10] Cover, T. and Hart, P. 1967. Nearest neighbor pattern classification. IEEE Transactions of Information Theory, January 1967.

[11] Cubeon, http://code.google.com/p/cubeon/

[12] de Souza, C.R., Quirk, S., Trainer, E., and Redmiles, D.F. 2007. Supporting collaborative software development through the visualization of socio-technical dependencies. In Proc. of the 2007 international ACM conference on Supporting group work..

[13] Dewan, P. and Hegde, R. 2007. Semi-Synchronous Conflict Detection and Resolution in Asynchronous Software Development. In Proc. E-CSCW 2007.

[14] Dourish, P., and Bellotti, V. 1992. Awareness and Coordination in Shared Workspaces. In Proc. CSCW 1992

[15] Gall, H., Hajek, K. and Jazayeri, M. 1998. Detection of Logical Coupling Based on Product Release History. In *Proc* ICSM 1998.

[16] Guzzi, A. and Begel, A. 2012. Facilitating communication between engineers with CARES. In Proc. ICSE 2012.

[17] Kersten, M. and Murphy, G.C. 2005. Mylar: a degree-of-interest model for IDEs. In Proc. AOSD 2005.

[18] Kersten, M. and Murphy, G.C. 2006. Using task context to improve programmer productivity. In Proc. SIGSOFT '06/FSE-14.

[19] Kraut, R. and Streeter, L. 1995. Coordination in software development. Communications of the ACM. 38, 3, 69-81.

[20] Kwan, I.; Schroter, A.; Damian, D. 2011. Does Socio-Technical Congruence Have an Effect on Software Build Success? A Study of Coordination in a Software Project. *IEEE Transactions on Software Engineering,* , vol.37, no.3, pp.307-324, May-June 2011.

[21] Minto, S. and Murphy, G.C. 2007. Recommending emergent teams. In Proc. MSR 2007.

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

[23] Sarma, A., Noroozi, Z., and van der Hoek, A. 2003. Palantír: raising awareness among configuration management workspaces. In Proc. ICSE 2003.

[24] Sarma, A., Maccherone,L., Wagstrom, P., and Herbsleb, J. 2009. Tesseract: Interactive visual exploration of socio-technical relationships in software development. In Proc ICSE 2009.

[25] Tasktop Dev, http://tasktop.com/products/visual-studio.php

[26] Valetto, G., Chulani, S., and Williams, C. 2008. Balancing the value and risk of socio-technical congruence. In Proc. STC 2008.

[27] Wong, S., Cai, Y., Valetto, G., Simeonov, G., and Sethi, K. 2009. Design rule hierarchies and parallelism in software development tasks. In Proc. ASE 2009.