Facilitating Coordination between Software Developers: A Study and Techniques for Timely and Efficient Recommendations

Kelly Blincoe, *Member, IEEE*, Giuseppe Valetto, *Member, IEEE*, and Daniela Damian, *Member, IEEE*

Abstract—When software developers fail to coordinate, build failures, duplication of work, schedule slips and software defects can result. However, developers are often unaware of when they need to coordinate, and existing methods and tools that help make developers aware of their coordination needs do not provide timely or efficient recommendations. We describe our techniques to identify timely and efficient coordination recommendations, which we developed and evaluated in a study of coordination needs in the Mylyn software project. We describe how data obtained from tools that capture developer actions within their Integrated Development Environment (IDE) as they occur can be used to timely identify coordination needs; we also describe how properties of tasks coupled with machine learning can focus coordination recommendations to those that are more critical to the developers to reduce information overload and provide more efficient recommendations. We motivate our techniques through developer interviews and report on our quantitative analysis of coordination needs in the Mylyn project. Our results suggest that by leveraging IDE logging facilities, properties of tasks and machine learning techniques awareness tools could make developers aware of critical coordination needs in a timely way. We conclude by discussing implications for software engineering research and tool design.

Index Terms—Computer-supported cooperative work, human factors in software design, management, metrics/measurement, productivity, programming teams

1 Introduction

Software developers often work on tasks in parallel when working on large projects. Technical dependencies between tasks can result in coordination needs between the developers. When those developers are unaware or do not obtain timely awareness [27] of the coordination that is required to manage their work dependencies (their coordination needs), there is potential for problems with respect to software productivity or quality [10], [60]. Studies have found that unfulfilled coordination needs can result in an increase in task resolution time, an increase in software faults, build failures, redundant work, and schedule slips [12], [19], [21], [24]. Therefore, awareness of coordination needs can be critical.

To provide the most benefits, awareness must be both timely and efficient. Awareness is timely if coordination needs are made known as they emerge, while development is still underway. Without timely awareness, developers may continue to work in isolation, without recognizing and acting on their coordination needs. Awareness is efficient if the recommendations on coordination needs (1) minimize information overload and (2) are easily understood by the developers. Without efficient recommendations, developers may incur additional and unnecessary coordination overhead. There are no existing methods or tools that provide both timely and efficient awareness of coordination needs to developers.

In this paper, we describe techniques to provide timely and efficient coordination recommendations. Our techniques leverage logging facilities that capture developers' actions in their Integrated Development Environments (IDE) as they occur to automatically detect developer coordination needs. We developed our techniques in an extensive study of coordination needs of one software project, the Mylyn development project. The Mylyn project was a useful case to study because Mylyn is the most well known and widely used software development support tool that, to fulfill its own purposes, records all developer IDE interactions as they occur. The Mylyn development team makes routine use of the Mylyn tool and, thus, its repository provides a large dataset of logged developer actions for analysis. Other examples of tools that capture developer actions as they occur within an IDE are Tasktop Dev and Cubeon.

We used the data available from IDE logging for *timely* coordination recommendations in our Proximity method [8]. Statistical analysis proved that Proximity's recommendations were timely, but its recommendations are not efficient. Proximity identifies coordination needs between pairs of developers which can result in information overload on large teams [11], [20]. Further, its recommendations can be unclear for developers since developers often work on multiple tasks in parallel and the recommendations do not identify the tasks that need coordinaton.

To understand how to produce *efficient* coordination recommendations, we interviewed developers. Following their

- *K. Blincoe and D. Damian are with the Software Engineering Global Interaction Lab, University of Victoria, Victoria, BC, Canada. E-mail: kblincoe@acm.org, danielad@uvic.ca.*
- *G. Valetto is with Fondazione Bruno Kessler, Trento, Italy. E-mail: valetto@fbk.eu.*

Manuscript received 17 Oct. 2014; revised 25 Mar. 2015; accepted 25 Apr. 2015. Date of publication 10 May 2015; date of current version 16 Oct. 2015. Recommended for acceptance by H. Sharp.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2015.2431680

advice, we (1) modified the original Proximity concept to identify coordination needs between pairs of tasks instead of developers, and (2) sought to minimize coordination recommendations to those that are more critical to developers. To identify ways to focus on the more critical coordination needs, we used content analysis and manual coding techniques in an in-depth repository analysis of task reports and developers' associated discussions.

This analysis led us to leverage properties of tasks and the software artifacts edited for the tasks. We developed ProximityML [6], which uses machine learning techniques on Proximity scores and these task properties to identify the more critical coordination needs between task pairs. We found ProximityML to be timely and efficient.

Overall, the contributions of this work include:

1. We report on a study of coordination and coordination needs in which we used developer interviews, repository data and statistical analysis to conceptualize timely and efficient coordination needs.
2. We show that logging of developer actions within their IDE can support timely, automatic and non-intrusive coordination recommendations.
3. We provide quantitative evidence that properties of tasks and their associated artifacts can be used to identify the more critical coordination needs.
4. We describe machine learning techniques to identify timely and efficient coordination recommendations.

Our previous conference publications reported on some elements of this work [6], [8]; however, this paper introduces numerous extensions to our work and provides a full overview of our techniques and their development through the study of coordination needs of the Mylyn team. Specifically this paper extends our previous publications by describing: (1) the motivation for our techniques through developer interviews, (2) a method for evaluating and understanding developers' critical coordination needs through content analysis and manual coding, (3) improved machine learning techniques and consideration of different task properties in ProximityML that improved the precision of our results from 0.09 to 0.77, and (4) further evaluation of our results by considering an additional research question focusing on the reliability and timeliness of ProximityML.

2 Background

2.1 Importance of Coordination in Software Engineering

Large software projects have many work dependencies [65]. Dependencies between tasks can lead to coordination needs between the task assignees [12], [24], [40], and these work dependencies must be managed [13], [40], [67]. Initially, research focused on ways to streamline the technical dependencies between modules as a way to maximize task parallelism [2], [60], [70]. However, it is not possible to eliminate all dependencies. Therefore, research began to focus on ways to satisfy, as opposed to reduce, work dependencies through coordination [40].

It has been found that a decrease in communication, the main form of coordination in software teams [47], can cause team members to be unaware of work dependencies resulting in coordination problems [23], [33], [39]. Missed coordination can result in build failures, duplication of work, schedule slips and software defects [12], [19], [21], [24]. On the other hand, it has been shown that aligning software teams based on the tasks they must complete can bring about productivity benefits [15], [49] and communication can reduce integration problems [38]. Coordination, therefore, has important effects on software quality and productivity.

Even if developers are willing to coordinate, they may often be unaware of their coordination needs. Maintaining awareness of coordination needs can be especially difficult when teams are geographically distributed [37]. Even in colocated teams, keeping aware of coordination needs is difficult since developers' coordination needs are often fluid and change throughout development [22].

2.2 Providing Awareness of Coordination Needs

There are two existing methods of providing awareness of coordination needs: configuration management conflict detection and Coordination Requirements detection. We first define and review the concepts of timeliness and efficiency from existing literature before we discuss the timeliness and efficiency of these methods.

Timeliness. Timeliness is a key requirement for presenting awareness information [30], [35], [42], [55], [63], [68]. Timeliness is important because awareness is only valuable if it is obtained when the information is useful. For example, becoming aware of a coordination need is only beneficial if the coordination need still exists. In the words of a senior developer that we interviewed, "*If you find out next week that you should have talked to this guy last week, that's not helpful.*" We define timely as occurring while development is underway. With timely awareness of coordination needs, developers can act upon and resolve their coordination needs as they surface to reap the proven productivity and quality benefits.

Efficiency. Gutwin and Greenburg [34] note that awareness can make coordination more efficient. Of course, awareness must be accurate. Futher, to create efficiency, awareness must provide "the appropriate amount of information, relevant to the user's sphere of activity [4]." We, therefore, consider three characteristics of the coordination needs in defining efficiency: the accuracy of the recommendations, the amount of information and the relevance to the user's current activity. Methods that notify developers of a large number of coordination needs risk information overload [41], [69]. Too many recommendations may force developers to take time away from development to sift through large amounts of data, which is inefficient. Likewise, methods that do not provide details such as which tasks or files another developer is working on may not provide enough details to enable developers to relate the coordination need to their current task and efficiently act on their coordination needs. Developers are left to understand the source of the coordination need on their own. Therefore, efficient awareness must be accurate and reduce the number of recommendations while providing enough details on each coordination need for it to be easily understood.

2.2.1 Configuration Management Conflict Detection

Configuration management conflict detection tools, like Palantir [64] and CollabVS [25], were the first to attempt to

provide awareness of coordination needs. They alert developers of possible conflicts by providing alerts when other developers are making changes to the files they are currently modifying in their local workspace.

While these methods are timely, their notifications are incomplete. They monitor the changes developers are making in their local workspaces and provide notifications while development is underway. However, they do not identify all indirect conflicts that can occur when changes in one file affect another file. Therefore, they do not provide a comprehensive view of coordination needs.

These methods are also inefficient. They risk information overload since they provide notification of every potential conflict at the source code level, and any concurrent modification will generate a notification regardless of complexity. This amount of information can be distracting for developers and is likely to contain a high number of trivial conflicts and false positives. This is inefficient since developers must sift through a large number of notifications to determine which conflicts really matter.

2.2.2 Coordination Requirement Detection

Cataldo and Herbsleb [12] were the first to introduce a framework for establishing a comprehensive view of coordination needs between developers. Their method performs matrix multiplication on task assignments and task dependencies to quantify the need to coordinate between developers working on dependent tasks in what they called *Coordination Requirements*. Coordination Requirements are a computable approximation of the coordination needs that occur in software development projects. Although many awareness tools [3], [24], [56], [62] have been created based on this method, its computations rely on commit data. This commit data is typically available only towards the end of the development work for a task, so the awareness this approach provides is not timely.

This method is also inefficient since it establishes Coordination Requirements between pairs of developers. Recent studies [11], [20] found that tools that recommend which developers should coordinate risk information overload on large teams. In addition, providing information only on which developers should coordinate may not enable developers to efficiently act on their coordination needs. Since developers are often working on multiple tasks in parallel, coordination needs at the developer level may encompass the work dependencies of many tasks. This puts the burden on the developers to identify what to coordinate about and introduces inefficiency.

There are, therefore, no existing methods that compute coordination recommendations (as approximations of coordination needs) that are both timely and efficient. In addition, the method that is timely does not produce a comprehensive view of coordination needs. Our research intends to fill this gap and is guided by the following research questions:

RQ1: Can a comprehensive set of coordination recommendations be identified in a timely way?
RQ2: Can a smaller set of more efficient coordination recommendations be identified?
RQ3: Can this smaller set of more efficient coordination recommendations be identified in a timely way?

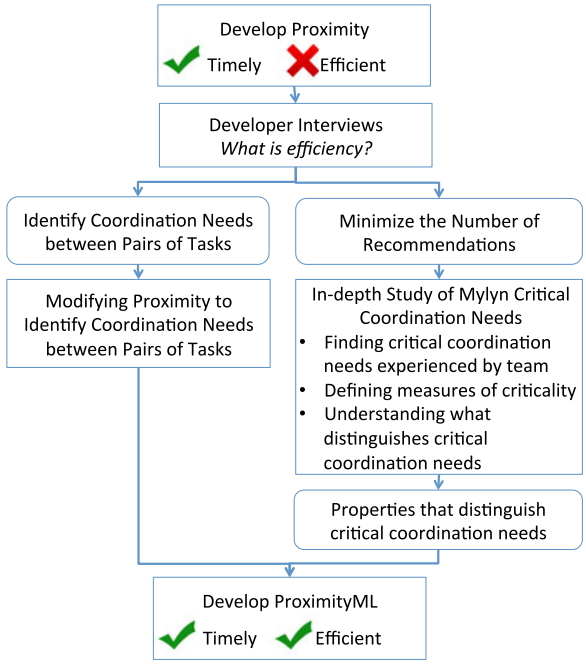

Fig. 1. Developing our techniques through a study of coordination needs of Mylyn developers.

3 Developing Techniques for Timely and Efficient Coordination Need Detection

3.1 Methodology

To answer our research questions, we performed an extensive study of the coordination data and developers' coordination needs in the Mylyn software project. We followed an iterative process of technique design, development and evaluation. Our research methods included interviews with Mylyn developers, in-depth analysis of coordination needs from the project repository, and statistical validation of our techniques. An overview of how our techniques were developed is depicted in Fig. 1.

Mylyn is a tool that transforms a developer's IDE to a task-centric view to make context switching between tasks easier. For this, Mylyn records all developer IDE interactions as they occur. These events are stored as context data for the task in focus. The techniques we developed use data on developer actions, which we obtained from the Mylyn plugin [44]. The Mylyn developers make routine use of the Mylyn plugin and use the bug-tracking database, Bugzilla, to define and assign developer tasks. We refer to Bugzilla change requests as tasks. The team documents all task details on the Bugzilla task report and uses these reports as a form of coordination.

We analyzed eight releases of the Mylyn project, releases 2.0 to 3.3, which spanned nearly three years of development. Each release lasted three to nine months. There were 51 contributors over all eight releases, with up to 32 contributors active during a release. Over the eight releases, there were 1,127 change requests that had both Mylyn context data and commit data. We determined the time of development for a change request by the artifact selection and edit activity obtained through the Mylyn context data attached to the Bugzilla record.

RQ1: Can a comprehensive set of coordination recommendations be identified in a timely way? To address our first

research question, we *developed Proximity* [8], a technique that identifies coordination needs between pairs of developers by leveraging logging facilities that monitor and record fine-grained actions developers take within their IDEs. We evaluated the accuracy and timeliness of Proximity scores relative to the Coordination Requirements of Cataldo and Herbsleb method [12]. This method was used for comparison since it is the most well known method for detecting coordination needs, and many awareness tools are based on this method. Proximity considers developer actions as they occur by utilizing IDE logging facilities, while the Cataldo et al. approach considers developer modification activity after development is mostly complete by examining commit records. We refer to the Cataldo et al.'s computed Coordination Requirements as *CCRs* throughout the remainder for brevity.

We found Proximity's recommendations of coordination needs between pairs of developers to be accurate and timely. However, we also observed that the recommendataions were not efficient since considering coordination needs between pairs of developers can result in information overload [11], [20] and recommendations which only note which developers must coordinate can be vague when developers are working on multiple tasks in parallel. Proximity is described in Section 3.2.

RQ2: Can a smaller set of more efficient coordination recommendations be identified? We conducted semi-structured *interviews* with six developers of the Mylyn project to gain a deep understanding on coordination needs (Section 3.3.1). Our interviewees recommended identifying coordination needs between pairs of tasks (instead of developers) since tasks are their unit of work. Further, they said the number of recommendations must be minimized to those coordination needs that are more critical to the developers. We explored techniques to address each recommendation: to identify coordination needs between pairs of tasks, we *modified Proximity* (Section 3.3.2); to minimize the number of coordination recommendations, we performed a *thorough analysis of coordination needs* resulting in a deep understanding of how to focus on more critical coordination needs (Section 3.3.3). Our analysis involved the following:

- Identify a set of the more critical coordination needs experienced by the team through content analysis and manual coding [48] of the task records; these coordination needs served as our ground truth for evaluation of our resulting technique, ProximityML. We developed a coding scheme, which details scoring criteria, for this analysis. The coding scheme was verified through interviews with Mylyn developers.
- Conceptualize measures of criticality of coordination needs using task duration and complexity.
- Analyze properties of the critical coordination needs experienced by the team to find ways to automatically identify only the more critical coordination needs.

This analysis resulted in a set of properties of the development tasks and the software artifacts manipulated during the completion of those tasks that can be used to distinguish between critical and trivial coordination needs. To answer RQ2, we *developed ProximityML*, a technique that uses machine learning on Proximity scores in combination with these properties to efficiently identify coordination needs (Section 3.3.4). We evaluated the accuracy (Section 3.3.4.) and criticality (Section 3.3.4) of the resulting coordination recommendations using the ground truth established through content analysis and manual coding.

RQ3: Can this smaller set of more efficient coordination recomendations be identified in a timely way? Finally, we evaluated ProximityML's reliability (Section 3.4.1) and timeliness (Section 3.4.2) to ensure that the addition of machine learning and additional properties did not affect the timeliness provided by the Proximity algorithm. For this analysis, we streamed pre-collected data, one event at a time, to replicate the actual progression of development work and the live collection of the data. We identified when ProximityML detected each coordination recommendation. We compared ProximityML's detection of the coordination need with the time the need was recognized by the team and the start of overlapping work.

Table 1 summarizes the validation steps for each of the three research questions.

3.2 Timely Detection of Coordination Needs: Proximity

RQ1: Can a comprehensive set of coordination recommendations be identified in a timely way?

We developed Proximity [8], a method that computes timely coordination needs between software developers using IDE logging data. Proximity is timelier than the existing Coordination Requirement detection methods because, instead of obtaining data from commits, it monitors the actions developers take in their IDE as they occur. When a developer looks at or edits an artifact in their IDE, the Mylyn framework captures that action. The actions are collected non-intrusively while the developers work and are stored as context data for the task in focus.

Proximity looks at artifact consultation and modification activities logged by Mylyn and weighs the overlap that exists between pairs of developers. It considers all actions recorded for each artifact in each developer's 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 is given when an artifact is simply consulted in both working sets (weight = 0.59). When an artifact is edited in one and consulted in the other working set, this is a mixed overlap (weight = 0.79). The weights are based on the weights Mylyn uses for its degree-of-interest model [44].

Fig. 2 illustrates an example of a Proximity computation. The algorithm computes the ratio of actual to potential overlap. Actual overlap is calculated as the intersection of the two working sets. In Fig. 2, b.java has an actual overlap score of 0.79 because it was edited in developer X's working set and consulted in developer Y's working set. Potential overlap represents the maximum possible Proximity score had there been perfect overlap between the two sets of actions and is calculated as the union of the two working sets. In Fig. 2, b.java has a potential overlap score of 1 since it was edited in one of the working sets.

TABLE 1
Summary of Research Methods and Validation Steps

Research Question	*Step*	*Validation*
RQ1	Develop Proximity	Statistical analysis. (1) Accuracy: Compare Proximity to CCRs through correlations and precision/recall. (2) Timeliness: Compare time of detection of Proximity and CCRs with Probability Density functions.
RQ2	What is Efficency?	Semi-structured interviews with six developers of the Mylyn project.
	In-depth Study of Mylyn Critical Coordination Needs	Content analysis and manual coding of 350 task pairs to identify set of critical coordination needs experienced by the team.
		Statistical analysis of task properties comparing properties of critical coordination needs to task pairs without critical coordination needs through Chi-Square and Mann-Whitney Tests.
	Develop ProximityML	Statistical analysis. (1) Accuracy: Compare ProximityML to ground truth established through content analysis and manual coding through precision/recall. Cross-validation of classifier. (2) Criticality: Analysis of change size and task duration of resulting recommendations.
RQ3		Statistical analysis. (1) Reliability: Analysis of the fluctuations of ProximityML recommendations over time. (2) Timeliness: Compare ProximityML's time of detection of a coordination need with the time the need was recognized by the team and the start of overlapping work.

Proximity outputs a score for each pair of developers indicating the strength of their coordination need. A score > 0 indicates a coordination need, and higher scores denote stronger coordination needs. Scores are scaled by the number of overlapping events to place greater weight on complex tasks that likely require coordination.

3.2.1 Evaluation of Proximity

Efficiency: Accuracy. To evaluate the accuracy of our Proximity scores, we calculated Proximity scores and CCRs for each pair of developers in each release. We performed a point-biserial correlation with Proximity scores and a binary vector denoting the presence of a CCR. Higher values of proximity correlate with the likelihood of a CCR (rho $= 0.55$, $p < 0.001$). We performed a Spearman correlation between the count of CCRs for each developer pair and the proximity scores (rho $= 0.69$, $p < 0.001$). We used a Spearman correlation because the data was not normally distributed. Both tests were statistically significant and showed strong positive correlations (results in Table 2).

We observed high levels of precision and recall when comparing against the CCRs (Table 3). Moreover, a thorough examination of the supposed false positives and false negatives revealed that Proximity can be even more accurate than CCRs. For example, we saw many cases where CCRs were not detected simply because work by one or both of the developers was never committed to the code base. However, Mylyn context events prove that those developers were, for some time, engaged in development on the very same artifacts—the epitome of a coordination need. In open source projects, where commit access is limited to only a select few developers, Proximity produces coordination recommendations between the actual code contributors rather than incorrectly detecting coordination needs simply because a user committed someone else's code contribution.

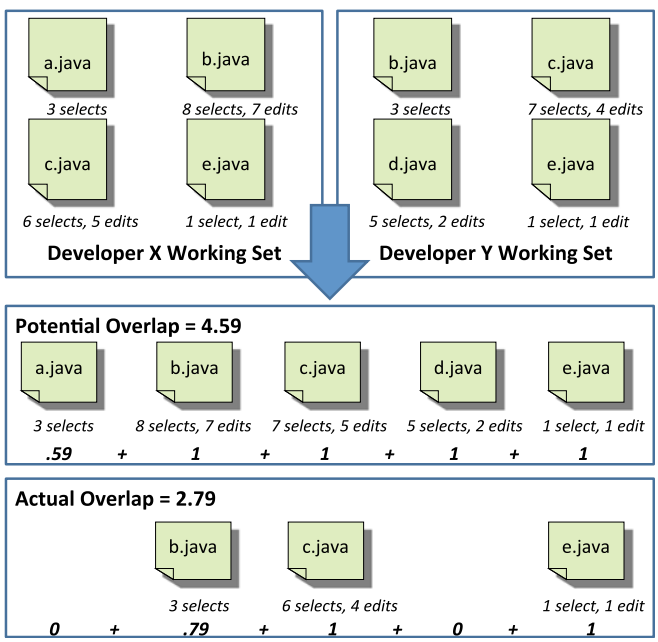

Fig. 2. Proximity algorithm example.

TABLE 2
Proximity versus CCRs Correlations

Test	p-value	Rho
Spearman	2.4e-11	0.69
Point-biserial	4.9e-07	0.55

TABLE 3
Proximity versus CCRs Precision/Recall

Precision	*Recall*
$42/58 = 0.72$	$42/46 = 0.91$

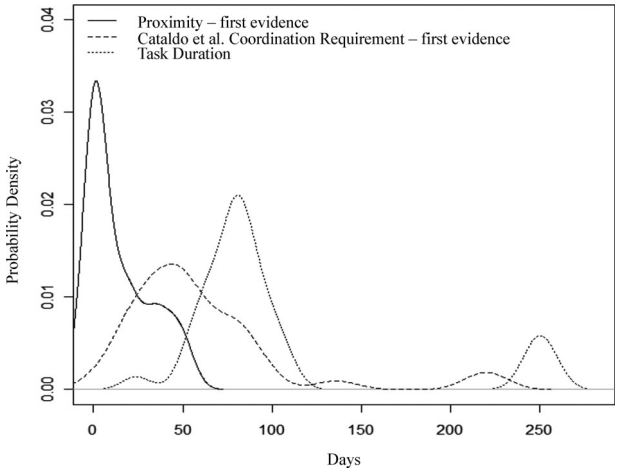

Fig. 3. Proximity algorithm timeliness relative to that of CCRs.

We also observed cases where CCRs were identified due to a technical dependency between two semantically unrelated tasks because they involved files that had been historically changed together by other developers often enough to cause a logical dependency to be established.

Additionally, the context events used in Proximity provide more granular information than is available from commit data. The Mylyn context data identifies the file name, class name and even the name of the class element (method or attribute) being consulted or edited. This allows Proximity to determine coordination needs more granularly, for example, to see whether two developers were working on the same area of code within a large file.

Timeliness. To evaluate Proximity's timeliness, we compared the time Proximity scores appear with the time CCRs are established. Proximity scores are calculated using events garnered instantaneously; while, CCRs are established after changes are committed. We obtained the date when the first contribution to the Proximity score occurred by considering the timestamp for the first overlapping event for a developer pair recorded in the Mylyn context data. Similarly, we considered the time the CCRs are first identified by considering the timestamp when the first technical dependency appears in the commits for a developer pair. Parallel work intervals last 102 days on average. The first evidence of Proximity is detected on average 14.2 days after parallel work begins. The first CCR detection happens 60.7 days on average after the beginning of concurrent work (a delay of 46.5 days). Fig. 3 shows the probability density functions illustrating the distribution of days before Proximity is detected, days before CCRs are detected, and, for reference, task duration in days over the entire dataset. It illustrates that a coordination need is likely to be detected via Proximity much earlier than via CCRs.

In answering RQ1. Timely coordination recommendations are possible with Proximity, which obtains developer actions as they occur through existing IDE monitoring facilities and analyzes the overlap of those actions to detect a comprehensive set of coordination needs.

3.2.2 Limitations of Proximity

While Proximity was able to detect coordination needs between pairs of developers in a timely way, it was not efficient. Developers often work on many tasks in parallel, so being aware of only which other developers they need to coordinate with does not provide enough context to allow for focused and efficient coordination. Developers are left to decide which of their tasks or code changes require coordination. The techniques we developed in answering RQ2 and RQ3 intend to address this limitation.

3.3 Efficient Detection of Coordination Needs

RQ2: Can a smaller set of more efficient coordination recommendations be identified?

We interviewed developers and performed an in-depth analysis of coordination needs on the Mylyn project to understand how we could make our recommendations more efficient. After this deep exploration, we developed a technique, ProximityML, which automatically detects coordination needs efficiently.

3.3.1 Developer Interviews

We conducted semi-structured interviews with six developers of the Mylyn project (one junior and five senior developers). The goal was to understand their perspectives on coordination recommendations. The interviews lasted 45 minutes on average. We describe the two main recommendations that emerged from these interviews.

Identify coordination needs for pairs of tasks. We asked the interviewees: *"Would a tool that recommended who to coordinate with be useful?"* Many of the senior Mylyn developers stated that coordination recommendations would be most useful at the task level as tasks are their logical unit of work. One developer stated *"if there was a lot more, than just talk to Joe, if it said like a new defect was filed or look at this related bug and Joe is the assignee, then I would consider it."* Receiving recommendations at the task level would allow for efficient coordination since there would be appropriate context around the recommendation. The developers described many issues relating to a lack of awareness of how their development tasks affect other tasks, how other tasks affect their own tasks, and who is responsible for tasks. They stated that this lack of awareness often resulted in duplication of work or unmanaged tasks. Coordination recommendations at the task level could help to mitigate these issues. Ko et al. [45] also found that developers are more interested in awareness about what information was relevant to their tasks.

Minimize the number of recommendations. To better understand the types of recommendations that would be most useful to developers we asked: How would you decide if you would act on coordination needs? What type of time window for recommending coordination needs do you think would be useful and why? The interviewees stated that the number of recommendations must be small. Too many recommendations would overwhelm the developer. This could affect the efficiency of the developers and cause them to ignore all recommendations. This is in line with previous research on the risk of information overload in awareness tools [41], [69].

The developers noted that not all dependencies between tasks require coordination. For example, when looking at a potential coordination need, one developer stated, "[the two tasks] are both working on the same area of code, but I don't see a direct need for coordination." Another developer focused on the simplicity of some tasks regardless of their technical dependencies saying on simple tasks, "I wouldn't consider coordinating anything with anyone, I would just go in fix it, close the bug and be done with it." The coordination recommendations should be limited to those that matter to the developers.

We used these two recommendations – identify coordination needs between pairs of tasks and minimize the number of recommendations – to explore ways of providing more efficient coordination recommendations. We adjusted Proximity to identify coordination needs between pairs of tasks instead of developers to provide better-scoped awareness (Section 3.3.2). We analyzed coordination needs to identify ways to minimize the number of recommendations (Section 3.3.3). Section 3.4 decribes the resulting technique, ProximityML, and its evaluation.

3.3.2 Modifying Proximity to Identify Coordination Needs between Pairs of Tasks

To detect coordination needs between pairs of tasks, we applied Proximity at the individual task level rather than at the developer level. This was done by aggregating the captured developer actions at the individual task level. Since the events were aggregated at the task level, a Proximity scores indicate the existence of and strength of a coordination need between pairs of tasks.

We calculated Proximity scores between tasks in the Mylyn release 3.2 which had 245 tasks (29,890 task pairs). We found 2,209 task pairs with Proximity scores>0, and 226 of the 245 tasks (>92 percent) were found to require coordination with at least one other task. This led us to believe that Proximity, when computed between pairs of tasks as opposed to pairs of developers, signaled too many coordination reccomendations. We, therefore, considered ways to minimize the number of recommendations.

3.3.3 In-Depth Study of Mylyn Critical Coordination Needs to Minimize the Number of Recommendations

We examined ways to focus our recommendations on those that the team would act on – the more critical coordination needs. When considering any technical dependency between a pair of tasks as a coordination need, we risk identifying even very trivial dependencies that do not need coordination. We, therefore, performed a thorough analysis of task records to identify when more critical coordination needs occur using content analysis and manual coding. We define our conceptualization of a critical coordination need and evaluate the criticality of the coordination needs experienced by the team. Finally, we performed a thorough analysis of these more critical coordination needs to identify additional properties that characterize those coordination needs. We used these properties in the development of ProximityML, which automatically identifies these more critical coordination needs.

Finding critical coordination needs experienced by team. A reliable way of capturing coordination needs between tasks is not recorded in existing software repositories. Bugzilla, for example, allows developers to indicate dependencies between tasks, but this may not capture all coordination needs or may capture trivial dependencies that do not require coordination. In addition, a recent study by Aranda and Venolia [1] found repositories like Bugzilla often provide incomplete information because of omission, oversight, or project conventions.

To better understand the more critical coordination needs experienced by the team, we turned to content analysis and manual coding techniques [7] that are well established in other research fields [48] and have also recently been used in Software Engineering [54]. To perform the manual coding, we developed a coding scheme with detailed task pair scoring criteria. We used a data driven method and reviewed several task pairs in which the need for coordination is explicitly discussed within the task reports. By analyzing these task pairs, we established four characteristics that appeared within the task reports indicating a coordination need. These were: (1) task summary similarity, (2) task discussion similarity, (3) evidence of task conflict, and (4) artifact overlap.

We obtained practical validation of these four characteristics through interviews with the Mylyn developers. Without indicating our identified characteristics, we asked three senior developers what they would look for within task reports that would indicate a need for coordination that they would act on. All three developers stated they would review the discussion threads on the task reports looking for references to similar features or problems, similar areas of the code, or conflicts occurring between the tasks. Two of the developers did not think the task summary would provide enough information since the summary is often incomplete or inaccurate. None of the developers suggested looking at overlapping artifacts between the two tasks. Artifact overlap suffers from the same problem that we are trying to solve, that is, it considers coordination needs between too many task pairs.

We, therefore, removed two task characteristics and established the two characteristics – task discussion similarity and evidence of task conflict – that allow for the identification of the more critical coordination needs between tasks. We put together a coding scheme that provided guidance on how each task pair should be rated for the two characteristics. The guidelines, which rate each characteristic on a three-point scale, are shown in Table 4.

To perform the content analysis, we used the relevant task information collected from the Bugzilla change requests for releases 3.1 and 3.2, the two releases in our dataset with the largest number of tasks. Each task was summarized in an easily digestible format, which allowed for side-by-side comparison.

To prepare the set of task pairs, we identified each task pair as either a potential critical coordination need or not. We considered a pair of tasks as a potential critical coordination need if the pair met one or more of the following criteria: the tasks had a high Proximity score where high is

TABLE 4
Manual Coding Guidelines

	No Coordination Need		*Critical Coordination Need*
Characteristic	*No*	*Somewhat*	*Very*
Task Discussion Similarity: Task discussions often include details of the task and any problems that have been encountered. We asked the coders to rate the similarity of the discussions occurring on each task.	The discussions of the two tasks do not share any of the same concepts.	The two task discussions refer to common aspects of the system from the perspective of EITHER the user (system features) or the system architecture (specific reference to code, modules, etc.) OR The two task discussions indicate that the problems may be occurring in the same area of the code.	The two task discussions refer to common aspects of the system from the perspective of BOTH the user (system features) and the system architecture (specific reference to code, modules, etc.) OR The two task discussions refer to the same or similar problems.
Evidence of Task Conflict: Task conflict is the epitome of a coordination need and often indications of conflicts exist in the task discussions (explicitly or implicitly). We asked the coders to look for such evidence.	The discussion in the two tasks does not seem to indicate that the two tasks were conflicting in any way.	The discussion in one of the tasks does not explicitly mention a conflict between the two tasks. However, based on reviewing the timing of the tasks and their discussions, it seems there may have been a conflict between the two tasks that the team may not have been not aware of at the time.	It is apparent based on the timing of the tasks and the discussion thread that there was a conflict between the pair of tasks. The conflict is clearly discussed and may or may not explicitly link the two tasks by ID.

greater than mean + (2 × stddev) of Proximity scores over all pairs; the tasks were marked as dependent or duplicate within their Bugzilla records; the tasks were cross-referenced in their discussions; the tasks were dependent on the same task (the team often uses this relationship to track subtasks of a large task); or the tasks were marked with the same tag. Once each task pair was designated as either a potential coordination need or not, we used a random number generator to select pairs from each set. We selected 155 potential critical coordination needs and 195 that were likely not coordination needs for a total set of 350 pairs. The number of pairs was based on the time availability of the coders.

We used two external people familiar with software development practices to perform the manual coding. To ensure higher confidence, the two coders performed the content analysis and coding independently. After each of the coders completed 12 task pairs, the two coders compared their findings and discussed differences as a way to calibrate amongst each other. Another comparison and calibration round was carried out after 100 task pairs. We checked intercoder reliability with Krippendorff's alpha measure [48]. We obtained a Krippendorff score of .91 for task discussion similarity and .87 for evidence of task conflict, which are indicative of high intercoder reliability.

We considered any task pair that was rated positively (*somewhat* or *very* on our scale in Table 4) for either characteristic as a more critical coordination need experienced by the team. We removed the task pairs for which the coders had a conflicting outcome leaving us with 313 task pairs. These task pairs serve as our ground truth, which we use for evaluation and analysis purposes in the rest of the paper. In this ground truth, 32 task pairs were identified as more critical coordination needs by the coders.

Defining measures of criticality. With a set of coordination needs that matter to the developers – the more critical coordination needs – we explored a way to measure criticality to use for the evaluation of our techniques. While previous research has proposed ways to rank the most important coordination needs at the developer level by considering the number of task dependencies involved in those coordination needs [28], [51], no prior research has examined the criticality of coordination needs at the task level. We considered two measures to evaluate the criticality of coordination needs at the task level: task duration and change size.

First, fulfilling coordination needs has been shown to reduce task resolution time [12], therefore we examined the durations of the tasks involved in the coordination needs. We computed task duration using the Mylyn context events. Since these events detail exactly when developers begin and complete their consultation and modification of artifacts for each task, we can compute the actual time developers spent working on a task. Long-duration tasks with coordination needs are likely the ones that can benefit the most from the productivity benefits provided by increased awareness and focused coordination.

Second, since the Mylyn team noted that they do not coordinate on simple or trivial tasks, we examined the complexity of the tasks involved in coordination needs. Cataldo and Herbsleb found that change size, measured as the number of code files modified for a task, is an accurate measure of task complexity [12]. We, therefore, adopted change size as our metric of task complexity.

Change size and task duration are moderately correlated in our data set (Spearman rho = 0.58, $p < 2.2e\text{-}16$). Task complexity is one of many factors that may influence a task's duration. Considering complexity with task duration helps us to avoid a bias towards tasks whose long duration may be due to some other factors that would not benefit as much from awareness and coordination, like low priority or inexperienced developers.

To examine whether the 32 coordination needs identified by the coders are indeed critical using our measures, we

TABLE 5
Criticality: Manual Coding Results

	Coordination Needs	*Other Task Pairs*	*Mann-Whitney Test*
Num. Tasks	152	93	–
Change Size	8.2 files	5.3 files	$U = 9398$; $Z = 4.5$; $p < 0.001$; $r = 0.25$
Task Duration	26.8 days	19.9 days	$U = 8603$; $Z = 3.1$; $p = 0.002$; $r = 0.18$

TABLE 6
Task Property Comparison

Property	*Coordination Needs*	*Other Task Pairs*	*Chi-Squared Test*
Task Pair Count	32	281	–
# with Proximity > 0	29	99	$x^2 = 34.2$; $p < 0.001$; $\Phi = 0.34$
# with same Product	26	228	$x^2 = 0$; $p = 1$; $\Phi = 0.001$
# with same Component	23	65	$x^2 = 29.5$; $p < 0.001$; $\Phi = 0.32$
# with same Platform	27	146	$x^2 = 10.9$; $p < 0.001$; $\Phi = 0.20$
# with Same OS	21	117	$x^2 = 5.8$; $p = 0.02$; $\Phi = 0.15$
			Mann-Whitney Test
Mean SLSM	5.7	0.80	$U = 7100.5$; $Z = 6.9$; $p < 0.001$; $r = 0.39$
Mean SLDM	6.94	3.22	$U = 1756.5$; $Z = 0.54$; $p = 0.29$; $r = 0.03$
Mean AL	7.3	0.93	$U = 7247.5$; $Z = 6.5$; $p < 0.001$; $r = 0.37$

analyzed their task duration and change size. The tasks have significant differences in both measures (results in Table 5). We therefore consider task duration and change size as our measures of criticality henceforth.

Understanding what distinguishes critical coordination needs. To identify techniques that could minimize recommendations and focus on the more critical coordination needs, we thoroughly examined the critical coordination needs experienced by the team identified through manual coding. We examined task pair properties of these critical coordination needs and compared them to the other manually coded task pairs to identify properties that can distinguish the more critical coordination needs. The task properties we examined include (1) architecture-related properties available from the project's change request database such as: the affected product, component, platform and operating system (OS) of the task and (2) modularity characteristics of the software artifacts involved in each task.

We examined the architecture-related properties by checking if the tasks involved in each task pair shared the same product, component, platform, or OS. We found that the more critical coordination needs are more likely to share the same component, platform and OS when compared to all other task pairs (results in Table 6).

To consider the modularity characteristics of the software artifacts involved in each task, we derived a design rule hierarchy (DRH) [70] of the Mylyn code base for the two releases of interest. A DRH assigns software artifacts to modules based on technical dependencies within the code. It clusters modules into "layers" where each layer depends only on the layers above. The DRH modules and layers allow us to identify potential coordination needs by considering three categories of work:

1. *Same layer same module (SLSM) pairs*: Two tasks include edits to artifacts that have a dependency and are in the same module. These represent potential coordination needs.
2. *Across layer (AL) pairs*: Two tasks include edits to artifacts that have a dependency and are in different modules and different layers. These represent potential coordination needs.
3. *Same layer different module (SLDM) pairs*: Two tasks include edits to artifacts that are in different modules of the same layer. By definition, there are no dependencies between these artifacts, so these are not coordination needs.

For illustration purposes, Fig. 4 shows a hypothetical DRH. The large thick-bordered boxes represent the layers while the boxes within the layers represent modules. The X's show the dependencies between the modules. Tasks 1 and 2 are an SLSM pair since they are operating on the same module. Tasks 2 and 3 are a SLDM pair since they are operating on the same layer but on different modules. Tasks 1 and 3 are an AL pair since they are operating on modules in different layers with a dependency.

The Mylyn DRH consists of 11 layers and 671 modules in release 3.1 and 11 layers and 786 modules in release 3.2.

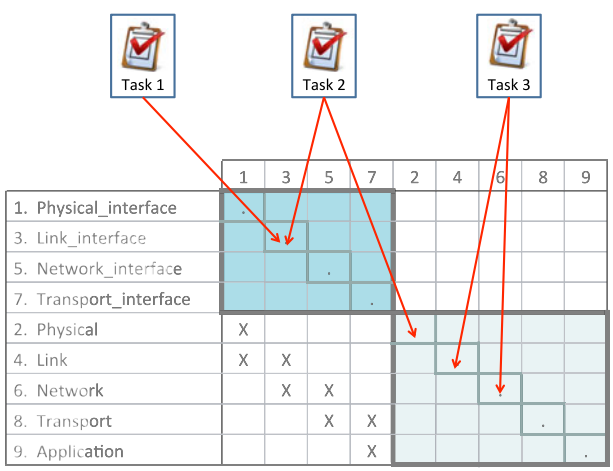

Fig. 4. Design rule hierarchy example [70].

We identified the associated DRH layer and module for each java artifact edited for each task and calculated the number of SLSMs, SLDMs and ALs for each task pair.

A Mann-Whitney test of difference in distribution shows that there is a statistically significant higher number of SLSMs and ALs in the critical coordination needs, but there is not a significant difference for the number of SLDMs (results shown in Table 6). This is consistent with the results by Wong et al. [70] that found developers engaged in SLSM and AL pairs coordinate significantly more than those engaged in SLDM pairs.

In addition to Proximity scores, we determined the following set of task pair properties that differentiate critical coordination needs from all other task pairs:

- Within same component
- Within same platform
- Within same operating system
- Number of SLSMs
- Number of ALs

3.3.4 Automatically Detecting Critical Coordination Needs: ProximityML

We developed ProximityML, which considers Proximity scores and these properties to automatically detect critical coordination needs. It identifies coordination needs between tasks and uses support vector machine (SVM) classification techniques [16] to minimize the recommendations to the more critical coordination needs. The name, ProximityML, means Proximity + machine learning.

An SVM is a supervised machine learning classification algorithm. Given a training set, it produces a model that can be used to predict the classification of unknown instances given a set of known parameters of those unknown instances [16]. We used Proximity scores and the other properties that were found to distinguish critical coordination needs in Section 3.3.3 as the known parameters for each task pair. SVM was selected because of its accuracy in general and its tolerance to noise and irrelevant, redundant and interdependent attributes [46].

In our previous publication, we used the k-nearest neighbor algorithm [17] due to the simplicity of implementing the algorithm and the exploratory nature of that study. We also previously examined the DRH properties differently, considering the number of overlapping layers and modules between task pairs. Here we report from an improved analysis that considers the number of SLSMs and ALs as a much better predictor of coordination needs since these types of overlaps are directly related to dependencies in the code base. The results we achieved with SVM far surpass those achieved using the k-nearest neighbor algorithm in our previous publication where we achieved high recall but a precision score of 0.09 [6].

We used LIBSVM [14] as our implementation of the SVM algorithm. LIBSVM is a java software package that provides support vector classification. It performs data scaling, parameter selection and model creation automatically. It ensures the data scaling is consistent across all data sets based on the range of each parameter in each set. For example, if a parameter in the training set had a range of [–10, +10] and the same parameter had a range of [–9, +12] in the test set, that parameter would be scaled to a range of [–1, +1] in the training set and to a range of [–0., +1.2] in the test set. To perform parameter selection, the LIBSVM library uses the radial basis function (RBF) kernel. It estimates the accuracy of each combination of parameters through cross validation (CV). The parameter combination with the highest CV score is selected.

TABLE 7
Accuracy: Ground Truth Critical Coordination Needs versus Coordination Recommendations

	Precision	*Recall*	*F1-score*
Proximity	0.33	1	0.5
ProximityML	0.77	0.71	0.74

Evaluation of ProximityML. We examined the accuracy, criticality, reliability and timeliness of the ProximityML recommendations. We used the dataset that had been manually coded through content analysis as our ground truth (in Section 3.3.3) to train and evaluate the machine learning algorithm. This set includes 313 total task pairs with 32 coded as critical coordination needs. The task pairs from release 3.1 (200 task pairs with 18 critical coordination needs) were used as a training set, while the task pairs from release 3.2 (113 task pairs with 14 critical coordination needs) were used as the evaluation set. Each parameter in our training set was linearly scaled to the range [–1, +1]. The parameters in the unknown and evaluation sets were scaled accordingly based on their range compared to the training set.

Efficiency: Accuracy. ProximityML significantly reduced the number of coordination recommendations compared to Proximity alone. Proximity produced 2,209 coordination recommendations, whereas ProximityML only 394, a reduction of 82 percent.

We compared the Proximity and ProximityML coordination recommendations with the ground truth critical coordination needs established through content analysis and manual coding. The differences in precision, recall, and f1-score of Proximity and ProximityML are shown in Table 7 for the 113 task pairs in our evaluation set. ProximityML had both high precision (low false positives) and recall (low false negatives) resulting in high overall accuracy, as shown by the f1-score. While a small number of coordination needs may be missed when employing ProximityML, it does not risk introducing a large number of false positives. On the other hand, Proximity has no false negatives, but a high number of false positives. Overall, ProximityML is much more accurate than Proximity.

A receiver operating characteristic (ROC) curve plots the true positive rate against the false positive rate for a binary classifier. The ROC curve shown in Fig. 5 illustrates the good performance of our classifier with the Area Under the Curve (AUC) equal to 0.8544. To prevent over fitting, we performed a grid search on C and γ using 10-fold cross-validation (CV) and obtained an average cross-validation rate of 92.0 with the best $C = 2^{11}$ and $\gamma = 2^{-7}$. Across the 10 folds, we saw little variance in our results with the lowest accuracy at 85 percent and the highest accuracy at 100 percent (standard deviation was 4.6). This high CV rate across each fold indicates we have a stable model that is able to

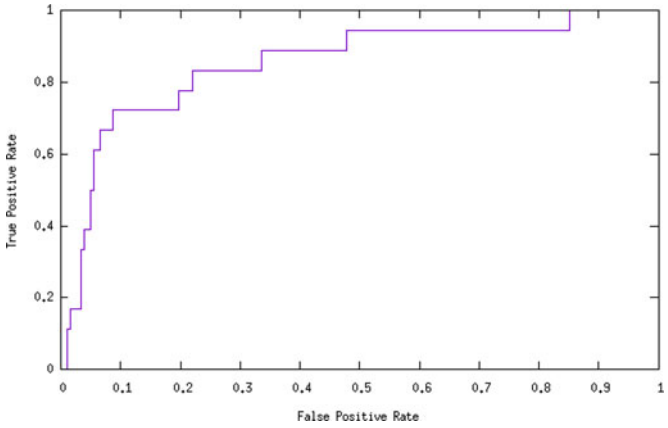

Fig. 5. ROC curve shows ProximityML is able to accurately identify more critical coordination needs.

accurately predict different samples; thus, we have avoided overfitting our model.

Efficiency: Criticality. We examined the ProximityML coordination recommendations using our two measures of criticality described in Section 3.3.3: change size and task duration. We see a strong, significant difference in both change size and task duration between the ProximityML coordination recommendations and all other tasks pairs (Table 8).

In addition, Mann-Whitney tests show both the change size and task durations of the tasks involved in the coordination recommendations are significantly different when comparing the ProximityML and Proximity methods (Table 9). The ProximityML coordination recommendation tasks' durations are significantly longer and change size is significantly bigger. This suggests that the properties used to enhance the Proximity metric and our machine learning techniques are identifying the more critical coordination needs.

In answering RQ2. We conclude that, by using additional task properties, ProximityML, was able to identify efficient coordination recommendations by identifying coordination needs between pairs of tasks and narrowing the set of identified coordination needs. Of course, we can not conclude that we have identified all of or the most critical coordination needs. However, we have shown that ProximityML can identify a subset of the more critical coordination needs with a low number of false positives and false negatives.

3.4 Timely and Efficient Detection of Coordination Needs

RQ3: Can this smaller set of more efficient coordination recommendations be identified in a timely way?

To answer this research question, we examined the (1) reliability of ProximityML recommendations over time and (2) timeliness of the recommendations to ensure that the addition of machine learning and additional properties did not affect the timeliness provided by Proximity.

We ran our machine learning techniques on our time-ordered data, which included each Mylyn context event and Bugzilla update event (task creation and task modifications). We streamed the data, one event at a time, to replicate the actual progression of development work and live collection of the data. We ran the machine learning algorithms to calculate coordination recommendations after every event. We performed this exercise on Mylyn release 3.2 data. The machine learner was pre-trained with the training set. Since data was streamed one event at a time, the machine learner initially had no knowledge of any data beyond the training set. This allowed us to evaluate the start-up behavior of ProximityML.

3.4.1 Reliability

To evaluate its reliability, we recomputed the ProximityML coordination recommendations after each event. Reliability is important because a tool that continuously changes its recommendations would not be trusted. Murphy and Murphy-Hill [58] found that users' trust immediately drops when a tool produces an irrelevant recommendation. A recommendation must only be made once it is firmly established as a critical coordination need. We would not expect one developer action to drastically alter the predicted coordination recommendations.

After each event, we examined the identified ProximityML coordination needs. At each point in time, we looked to see how many of the coordination recommendations were not included in the final set of ProximityML recommendations (those detected after all events have been streamed). We consider any coordination recommendation that does not appear in the final set of ProximityML recommendations a false positive for the purposes of this exercise. Fig. 6 shows the number of coordination recommendations as well as false positives that have been identified after each

TABLE 8
Criticality: ProximityML Coordination Recommendations

	Coordination Recommendations	*Other Task Pairs*	*Mann-Whitney Test*
Tasks (count)	152	93	–
Change Size	5.6 files	4.0 files	$U = 22709; Z = 4.8; p < 0.001; r = 0.31$
Task Duration	12.16 days	2.3 days	$U = 9666; Z = 7.7; p < 0.001; r = 0.49$

TABLE 9
Coordination Recommendations Criticality

	Proximity	*ProximityML*	*Mann-Whitney Test*
Change Size	4.3 files	5.6 files	$U = 8976; Z = 6.9; p < 0.001; r = 0.56$
Task Duration	9.1 days	12.16 days	$U = 7829; Z = 4.4; p < 0.001; r = 0.35$

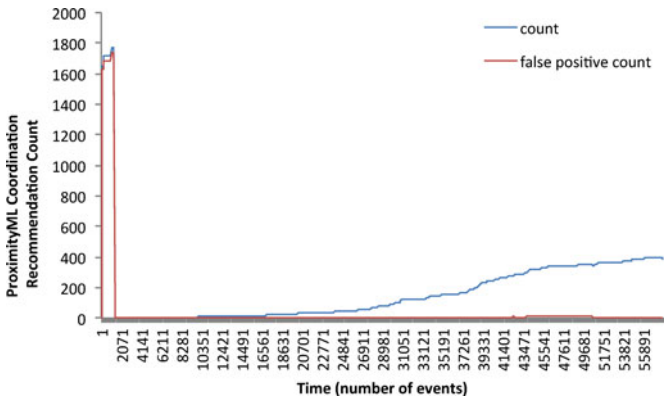

Fig. 6. Evolution of ProximityML coordination recommendations over time.

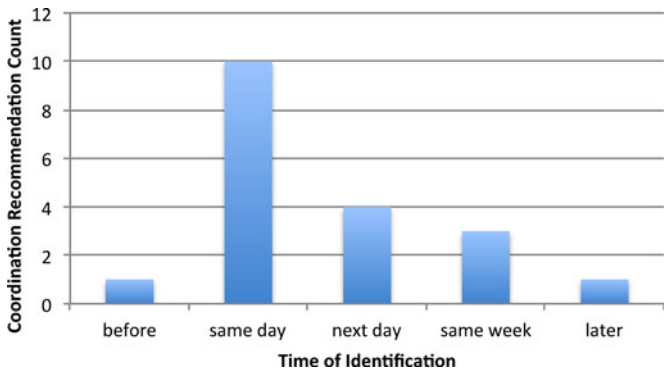

Fig. 7. Coordination recommendation timeliness for set of 19 ProximityML coordination recommendations that are recognized by the team in Bugzilla.

new event is introduced over the duration of the entire dataset. We observe that there is a small period of unreliability during the early stage of the data streaming where our technique produces many recommendations due to the limited amount of data available. After a brief initialization period, the results are reliable with a minimal number of false positives.

3.4.2 Timeliness

To examine the timeliness of the ProximityML coordination recommendations, we identified the timestamp when ProximityML first identified each of the 394 ProximityML coordination recommendations. Ideally, we could compare when ProximityML identified each of these coordination recommendations with when the team first identified the corresponding coordination need. However, we only have evidence of when 19 of these coordination needs were recognized by the team through the dependency data available in the Bugzilla reports. We identified dependencies within Bugzilla reports in three ways: (1) the explicitly marked "depends on/blocks" relationship, (2) the "duplicate" relationship between tasks, and (3) the task cross-referencing relationship. We examined these 19 coordination needs that were both identified by ProximityML and in their Bugzilla reports.

Sixteen of these 19 recognized coordination needs were known dependencies at the time of the second task creation. These tasks represent either task/subtask relationships or offshoot tasks where some new task is created based on something that was discovered during the development of the first task. In these cases, we cannot expect ProximityML to perform better than the development team. Still, in all but one case, ProximityML automatically identifies these recognized coordination needs promptly after the creation of the second task: as shown in Fig. 7, most are identified on the same day or the day after the second task is created. The team did not identify the remaining three recognized coordination needs until sometime later during the development of the second task. ProximityML identifies two of these recognized coordination needs on the same day as the team and one more than a month before the team.

While this represents only a small set of recognized coordination needs, it shows the promise of ProximityML to automatically provide timely awareness to the development team. Since it provides both accurate detection and early recognition, ProximityML delivers recommendations that are actionable. This is especially important when those coordination needs are not immediately evident to the team members. Since we have no direct way to compare the time of detection for the remaining 375 unrecognized coordination needs, we instead analyzed the timeliness of the detection of CRs relative to the start of overlapping work. The start of overlapping work was calculated by considering the timestamp of overlapping Mylyn context events for each coordination need. ProximityML coordination recommendations are identified on average 3.6 days after the start of overlapping work with the median detection on the same day as the start of overlapping work. This provides actionable recommendations considering the average development duration for tasks in this data set is nearly 25 days. Fig. 8 illustrates the timeliness with probability density functions showing that ProximityML typically produces coordination recommendations when overlapping work starts or shortly after. In some cases, ProximityML even produced coordination recommendations just before the start of overlapping work due to the inclusion of the other task properties triggering the recommendation. We believe that this early detection makes the ProximityML coordination recommendations actionable.

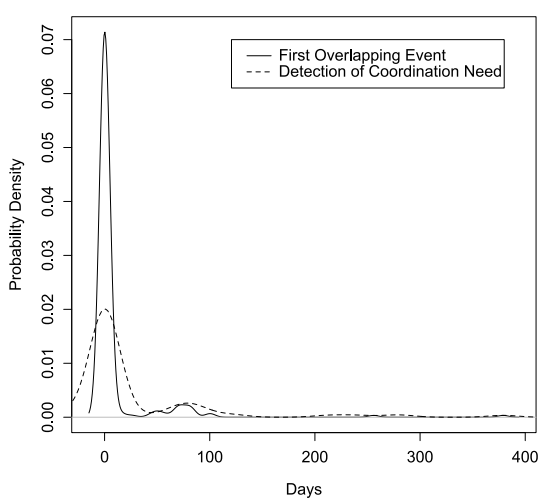

Fig. 8. ProximityML timeliness probability density.

In answering RQ3. ProximityML produces a smaller set of coordination recommendations as the coordination needs emerge making ProximityML timely and efficient.

4 Discussion

In this paper, we described our exploration to provide timely and efficient coordination recommendations in software teams. Our work started from the premise that IDE logging facilities provide a rich set of data that can support automatic detection of coordination needs. Tools, such as Mylyn, Tasktop Dev and Cubeon, log the actions developers take as they interact in their IDE. Since IDE monitoring captures developer actions as they occur it can be used for timely identification of coordination needs. We studied the development of the Mylyn project itself, since their developers consistently utilize the Mylyn plugin and, thus, the dataset of captured developer actions is large. We used this data to develop and validate our techniques. Our first technique, Proximity, considered the overlap between the artifacts consulted and modified by developers to identify timely coordination needs. Proximity provided timely and accurate recognition of coordination needs between pairs of developers. To provide more efficient recommendations, we also explored ways to detect coordination needs between pairs of tasks, a level of analysis more granular than the developer level. To avoid information overload and a high number of false positives, we leveraged a set of task properties that distinguished the more critical coordination needs. We used machine learning on Proximity scores and those task properties to filter recommendations to the more critical coordination needs, providing a smaller number of critical coordination recommendations. The final outcome of this exploration is our technique, ProximityML, which uses task properties for timely and efficient coordination needs recommendation.

4.1 Using ProximityML in Other Projects

ProximityML can be used in other projects, provided they utilize IDE logging facilities, by considering each of its components: (1) a ground truth of critical coordination needs, (2) task properties that distinguish critical coordination needs, and (3) an SVM machine learning algorithm.

Develop ground truth. First, a set of task pairs with known critical coordination needs and a set of task pairs that do not require coordination must be identified. These can be established using the manual coding guidelines we developed in Section 3.3.3 or through consultation with the development team. The ground truth will be used to analyze task properties and train the machine learner.

Identify relevant task properties. A list of properties that, in addition to Proximity scores, can distinguish critical coordination needs must be identified for the project. While it is likely that the properties described in our analysis of the Mylyn project data will also apply to other projects, they may not be universally applicable due to specific project practices or conventions. A list of project-specific task properties can be identified by comparing the ground truth critical coordination needs with task pairs that do not require coordination as described in Section 3.3.3 for the Mylyn project. A statistical evaluation can support this discovery processs, and help identify properties that differ significantly for the known critical coordination needs.

SVM machine learning algorithm. With the ground truth and task properties, the technique described in Section 3.3.4 can be applied. The input to the SVM machine learner is the training set (ground truth) where each instance of the training set is classified (critical coordination need or not) and described by the selected properties. After training the machine learner, unknown task pairs can be classified by providing the values of the selected properties. When task pairs that are classified as critical coordination needs, coordination recommendations can be made to the developers assigned to those tasks.

4.2 Implications for Research

Our study has several important implications for software engineering research.

Implication #1: *IDE logging data holds significant promise for software engineering researchers.* We found that the data that can be obtained by monitoring the actions developers take within their IDE can be useful for coordination awareness since the data is so timely and provides rich information about the context of a developer's activities. While there are some other examples of software engineering studies that have leveraged this data [32], [36], [43], [59], IDE logging facilities have so far received limited attention in research. However, the data they provide has significant potential, and researchers can continue to study ways to exploit this valuable source of data.

Implication #2: *Awareness of tasks leads to forms of implicit coordination.* An important finding emerging from our interviews was how the developers described they would attend to coordination needs. Communication has been found to be the main form of coordination in software teams [47]. However, all of the senior developers indicated that, upon receiving a recommendation of a coordination need between tasks, they would simply review the related task to obtain details of how that task impacts their own work as a first step rather than communicating with the assignee of the other task. This review would result in awareness of the related task. They would avoid interrupting the developer assigned to the other task, even if it meant delaying their own task. One developer stated "*just by looking at the bug report too, you can rule out your potential need to go interrupt that person or figure out, alright I'll just hold off my development until they are done or whatever the case may be, instead of actually going and interrupting that person. So you can glean a lot of information from that report just by being aware of the similar reports you should be looking at*." Reviewing the appropriate related tasks could be seen as a form of stigmergic coordination [9], [29] on the Mylyn project since the team encourages documentation of all task details within the task report. We discuss how a tool based on our technique can support implicit and stigmergic coordination in Section 4.3.

Implication #3: *Effects of implicit coordination in software engineering need further study.* Many existing empirical studies on coordination examine explicit (and easily traceable) means of communication such as email, chat or meetings. We believe it is equally important to take into account other means of coordination. For example, studies that use measures for socio-technical congruence

(STC) [12], [52] could be improved by also considering metrics for awareness about tasks as sufficient coordination to fulfill a coordination need. Future studies should examine this possibility by considering either tasks that developers are watching or have subscribed to or tasks that have been reviewed by developers, which can be obtained through IDE monitoring facilities when developers view the task report within the IDE. Further information on developer awareness of tasks or of other developers can be garnered from "social" features that have recently been introduced in software repositories and development communities like GitHub [18].

4.3 Implications for Tool Design

Our work shows the potential of a support tool for developers that automatically recognizes coordination needs between pairs of tasks as they emerge by leveraging IDE logging. Such a tool could be used to automate task dependency management, provide awareness both within and across teams, and support coordination among developers. The envisioned tool could incrementally and unobtrusively learn from coordination actions taken by the team (discussions, cross-referencing of task pairs, etc.) to continuously improve its accuracy. It should have a large pool of potential parameters and perform parameter selection based on incremental learning to ensure that the parameters are best suited for the development processes and practices of the team. Our work indicates some main design guidelines for such a tool.

Guideline #1: *The tool must be unobtrusive.* The developers we interviewed suggested displaying coordination recommendations either within the task reports themselves, which developers often consult throughout development, or within an IDE plug-in. The recommendations should include links to the other task reports and any other relevant task information to allow the developer to easily gather information about the task on their own, without interrupting the developer assigned to the other task. It could also include an easy way to display the areas of code that are overlapping or conflicting. There should be in-tool coordination mechanisms including email, Skype, Yammer or other communication software used by the project. However, developers should have a way to flag themselves as busy to avoid interruption when necessary. A tool might also consider the priority of a task when making recommendations or when displaying the available options on how to fulfill the coordination need. Perhaps low priority tasks would only suggest implicit types of coordination.

Guideline #2: *The tool must balance the relevance and timeliness of the coordination need to provide the most valuable recommendations.* A tool would likely introduce a form of decay for the coordination recommendations as the involved tasks aged. It would also need to identify a time window of interest for tasks to incur a coordination need (e.g. only currently overlapping tasks or tasks that were worked no more than two weeks apart). This time window should be a tunable attribute since different developers may have different preferences. From our interviews, we learned that understanding very relevant tasks that were completed much earlier in the project's timeframe could still be useful in some cases. For example, when the new task is attempting to tackle the same issue as a previous unsuccessful task. One developer said "*you may be doing something that someone tried five years ago, and they have information about why it failed.*" This illustrates another way developers may use such a tool, i.e. for gaining awareness of tasks rather than for explicit coordination. Relevant coordination needs may be displayed regardless of the completion status of the other task.

Guideline #3: *The tool should consider the experience level of the developer when making recommendations.* The developers we interviewed believed more experienced developers would benefit the most from awareness of coordination needs since they have the knowledge to understand the related tasks. While previous research [66] found that developers consider the expertise of others before initiating coordination, our findings suggest that the expertise of the developers themselves may impact what coordination they deem necessary. More junior developers may want a smaller set of only extremely relevant recommendations. Tools, therefore, may need to consider not only the properties of tasks, but also the task assignee.

Guideline #4: *The tool should support implicit coordination [9], [29].* Our interviewees would prefer to gather task information themselves rather than interrupting a task assignee. While the Mylyn team strives to record all information related to each task within the corresponding Bugzilla report, tools for augmented support could be devised. There has been some research in this area; for example, Rastkar and Murphy [61] summarize email threads related to a specific bug report. However, there are many other forums (IRC, Skype chats, etc.) and information sources (design documentation, requirement specifications, etc.) that may hold information relevant to a task. The awareness tool described above could be improved by providing a summary of the tasks involved in a coordination need so the developer can quickly browse the information. A tool could summarize the task report [53] and all task information from these various sources and prioritize and highlight the most relevant information. The developer could view additional details of tasks that require further investigation. Such a tool would enable more efficient awareness of other tasks.

4.4 Threats to Validity

One threat to validity is that our findings derive from a study of a single project with a moderate number of developers and tasks. Our results could be affected by specificities of the project. To mitigate this risk, we performed a detailed analysis of Proximity using eight versions of the Mylyn project. ProximityML was evaluated using a release with a large number (29,890) of task pairs. We interviewed developers to understand the team's coordination practices and problems. Our detailed analysis of this project allowed us to better understand critical coordination needs and how to identify them.

Our interviewees were all Mylyn contributors and were self-selected. While we reached saturation in our results, they may not generalize to other projects. However, many interviewees discussed their experiences on other software projects in addition to the Mylyn development project.

The manual coding and content analysis also introduce a possible risk of unreliable results since the coding is subjective in nature. However, we had two independent coders and achieved high intercoder reliability indicating that this risk has been mitigated.

Another issue is that we were limited in the number of task 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 needs and highlight the most important ones. The properties that are relevant in this study may not be as relevant in others. In addition, all properties may not be portable across different bug tracking systems.

Our measure of task duration, which we used to evaluate the criticality of coordination needs between tasks, could be affected by other factors, such as the priority of the tasks, workload of the team, physical location of the developers, and experience level of the developers. However, Cataldo and Herbsleb [12] found that while these factors impact development time, the impact of unmanaged coordination needs is also significant. In our Mylyn study, this risk is further mitigated by the characteristics of the Mylyn project itself and the general nature of open source projects. The Mylyn team is comprised of well-established, experienced developers. Open source projects are accustomed to working in distributed environments [57], [66], and developer overload is not a large concern, since contributors choose which tasks to work on [57].

5 Related Work

Several lines of research are dedicated to reducing conflicts and promoting coordination between developers including schedule optimization techniques [26], [43], configuration management conflict detection techniques [5], [25], [64], and Coordination Requirement detection methods [12], [28], [50], [51].

Schedule optimization is one way to reduce conflicts between tasks. di Penta et al. found that optimization of project scheduling can reduce coordination overhead through evaluation of their search-based optimization techniques [26]. A more recent tool, Cassandra [43], identifies potential conflicts between tasks based on the files in their workspaces and suggests optimal scheduling to avoid those conflicts. While these schedule optimization techniques can certainly reduce coordination needs of a development team, they will not be able to fully eliminate the need for coordination. This is particularly true when the schedule is tight and large amounts of work need to be done in parallel despite the conflicts that may arise.

Another approach is to detect conflicts early to allow for coordination. Tools, like Palantir [64], were built on top of configuration management systems. Palantir helps alert developers of possible conflicts by letting them know which other developers are making changes to the files they are currently modifying. It also considers one type of indirect conflicts by considering changes made to the signature of a method that affect other artifacts that call that method. However, Palantir provides only a list of notifications regarding each potential conflict and does not provide a cumulative view of coordination needs. It also risks information overload since it provides notifications for every potential conflict at the source code level. FastDash [5] and CollabVS [25] are other examples of conflict detection tools that suffer from the same limitations as Palantir. Compared to ProximityML, these approaches are not efficient due to their risk of information overload and their lack of complete view of coordination needs.

Cataldo and Herbsleb [12] were the first to introduce a framework for establishing a cumulative view of coordination needs between developers. Their method establishes Coordination Requirements between developers who are working on dependent tasks. Task dependencies are approximated by logical dependencies [31] between artifacts involved in those tasks. Data about logical dependencies is obtained by mining the source control repository of the project for commits. Although some ways to rank Coordination Requirements by importance [28], [51] have been presented, current methods do not differentiate between less or more intense Coordination Requirements or distinguish between different kinds of coordination needs. There are several awareness tools [3], [24], [56], [62] that detect Coordination Requirements between pairs of developers. Compared to ProximityML, the existing Coordination Requirement detection methods and tools are not timely due to their reliance on commit data. They are also less efficient, since they provide recommendations only between pairs of developers and leave the developers to determine what to coordinate about.

6 Conclusion

Our techniques leverage IDE logging facilities that capture developer actions as they occur allowing for timely detection of coordination needs. They also leverage a set of task properties to focus our recommendations on the more critical coordination needs. The techniques described in this paper can be used to create a support tool for developers that automatically and non-intrusively recognizes coordination needs between pairs of tasks as they emerge and focuses on the more critical recommendations to minimize information overload. Such a tool could be used to automate task dependency management, provide awareness both within and across teams, and support coordination among developers. Avenues for future work include developing such a support tool and continuing our investigation to identify additional properties that characterize critical coordination needs.

Acknowledgments

The authors thank Sean Goggins and Nora McDonald for their manual coding efforts and the developers who dedicated time for interviews. This work was supported in part by the US National Science Foundation (NSF) grant VOSS OCI-1221254 and NECSIS Grant.

References

[1] J. Aranda and G. Venolia, "The secret life of bugs: Going past the errors and omissions in software repositories," in *Proc. 31st Int. Conf. Softw. Eng.*, 2009, pp. 298–308.

[2] C. Y. Baldwin and K. B. Clark, *Design Rules: The Power of Modularity*, vol. 1. Cambridge, MA, USA: MIT Press, 2000.

[3] A. Begel, Y. P. Khoo, and T. Zimmermann, "Codebook: Discovering and exploiting relationships in software repositories," in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng.*, 2010, pp. 125–134.

[4] V. Bharadwaj, "Supporting awareness in heterogeneous collaboration environments," Ph.D. dissertation, West Virginia Univ., Morgantown, WV, USA, 2005.

[5] J. T. Biehl, M. Czerwinski, G. Smith, and G. G. Robertson, "FASTDash: A visual dashboard for fostering awareness in software teams," in *Proc. Int. Conf. Human Factors Comput. Syst.*, 2007, pp. 1313–1322.

[6] K. Blincoe, G. Valetto, and D. Damian, "Do all task dependencies require coordination? The role of task properties in identifying critical coordination needs in software projects," in *Proc. 9th Joint Meeting Found. Softw. Eng.*, 2013, pp. 213–223.

[7] K. Blincoe, G. Valetto, and D. Damian, "Uncovering critical coordination requirements through content analysis," in *Proc. Int. Workshop Social Softw. Eng.*, 2013, pp. 1–4.

[8] K. Blincoe, G. Valetto, and S. Goggins, "Proximity: A measure to quantify the need for developers' coordination," in *Proc. Conf. Comput. Supported Cooperative Work*, 2012, pp. 1351–1360.

[9] F. Bolici, J. Howison, and K. Crowston, "Coordination without discussion? Socio-technical congruence and stigmergy in free and open source software projects," in *Proc. 2nd Int. Workshop Socio-Tech. Congruence*, 2009.

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

[11] M. Cataldo and K. Ehrlich, "The impact of communication structure on new product development outcomes," in *Proc. Int. Conf. Human Factors Comput. Syst.*, 2012, pp. 3081–3090.

[12] M. Cataldo and J. D. Herbsleb, "Coordination breakdowns and their impact on development productivity and software failures," *IEEE Trans. Softw. Eng.*, vol. 39, no. 3, pp. 343–360, Mar. 2013.

[13] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb, "Software dependencies, work dependencies, and their impact on failures," *IEEE Trans. Softw. Eng.*, vol. 35, no. 6, pp. 864–878, Nov./Dec. 2009.

[14] C. C. Chang and C. J. Lin, "LIBSVM: A library for support vector machines," *ACM Trans. Intell. Syst. Technol.*, vol. 2, no. 3, p. 27, 2001.

[15] M. E. Conway, "How do committees invent?" *Datamation*, vol. 14, no. 4, pp. 28–31, 1968.

[16] C. Cortes and V. Vapnik, "Support vector machine," *Mach. Learn.*, vol. 20, no. 3, pp. 273–297, 1995.

[17] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Trans. Inf. Theory*, vol. IT-13, no. 1, pp.21–27, Jan. 1967.

[18] L. Dabbish, C. Stuart, J. Tsay and J. Herbsleb, "Social coding in GitHub: Transparency and collaboration in an open software repository," in *Proc. Conf. Comput. Supported Cooperative Work*, 2012, pp. 1277–1286.

[19] D. Damian, L. Izquierdo, J. Singer, and I. Kwan, "Awareness in the wild: Why communication breakdowns occur," in *Proc. 2nd Int. Conf. Global Softw. Eng.*, 2007, pp. 81–90.

[20] C. R. de Souza, J. M. Costa and M. Cataldo, "Analyzing the scalability of coordination requirements of a distributed software project," *J. Brazilian Comput. Soc.*, vol. 18, no. 3, 2012, pp. 201–211.

[21] C. R. de Souza and D. F. Redmiles, "An empirical study of software developers' management of dependencies and changes," *Proc. 30th Int. Conf. Softw. Eng.*, 2008, pp. 241–250.

[22] C. R. de Souza and D. F. Redmiles, "The awareness network, to whom should I display my actions? and, whose actions should i monitor?" *IEEE Trans. Softw. Eng.*, vol. 37, no. 3, pp. 325–340, May/Jun. 2011.

[23] C. R. de Souza, D. Redmiles, L. T. Cheng, D. Millen, and J. Patterson, "How a good software practice thwarts collaboration: the multiple roles of APIs in software development," *ACM SIGSOFT Softw. Eng. Notes*, vol. 29, no. 6, pp. 221–230, Oct. 2004.

[24] C. R. de Souza, S. Quirk, E. Trainer, and D. F. Redmiles, "Supporting collaborative software development through the visualization of socio-technical dependencies," in *Proc. Int. Conf. Supporting Group Work*, 2007, pp. 147–156.

[25] P. Dewan and R. Hegde, "Semi-synchronous conflict detection and resolution in asynchronous software development," in *Proc. 10th Eur. Conf. Comput. Supported Cooperative Work*, 2007, pp. 159–178.

[26] M. Di Penta, M. Harman, G. Antoniol, and F. Qureshi, "The effect of communication overhead on software maintenance project staffing: A search-based approach," in *Proc. Int. Conf. Softw. Maintainence*, 2007, pp. 315–324.

[27] P. Dourish and V. Bellotti, "Awareness and coordination in shared workspaces," in *Proc. Conf. Comp.-Supported Cooperative Work*, 1992, pp. 107–114.

[28] K. Ehrlich, M. Helander, G. Valetto, S. Davies, and C. Williams, "An analysis of congruence gaps and their effect on distributed software development," in *Proc. 1st Int. Workshop Socio-Tech. Congruence*, 2008.

[29] M. A. Elliott, "Stigmergic collaboration: The evolution of group work," *M/C J.*, vol. 9, no. 2, May 2006.

[30] G. Fitzpatrick, P. Marshall, and A. Phillips, "CVS integration with notification and chat: Lightweight software team collaboration," in *Proc. Conf. Comput. Supported Cooperative Work*, 2006, pp. 49–58.

[31] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Proc. Int. Conf. Softw. Maintainence*, 1998, pp. 190–198.

[32] S. P. Goggins, G. Valetto, C. Mascaro, and K. Blincoe, "Creating a model of the dynamics of socio-technical groups," *User Model. User-Adapted Interaction*, vol. 23, no. 4, pp. 345–379, 2013.

[33] R. E. Grinter, J. D. Herbsleb and D. E. Perry, "The geography of coordination: Dealing with distance in R&D work," in *Proc. Int. Conf. Supporting Group Work*, 1999, pp. 306–315.

[34] C. Gutwin and S. Greenberg, "A descriptive framework of workspace awareness for real-time groupware," *Comput. Supported Cooperative Work*, vol. 11, no. 3–4, pp. 411–446, 2002.

[35] C. Gutwin and S. Greenberg, "Workspace awareness for groupware," in *Proc. Conf. Companion Human Factors Comput. Syst.*, 1996, pp. 208–209.

[36] A. Guzzi, M. Pinzger, and A. van Deursen, "Combining microblogging and IDE interactions to support developers in their quests," in *Proc. Int. Conf. Softw. Maintenance*, 2010, pp. 1–5.

[37] J. D. Herbsleb, "Global software engineering: The future of socio-technical coordination," in *Proc. Future Softw. Eng.*, 2007, pp. 188–198.

[38] J. D. Herbsleb and R. E. Grinter, "Architectures, coordination, and distance: Conway's law and beyond," *IEEE Softw.*, vol. 16, no. 5, pp. 63–70, Sept./Oct. 1999.

[39] J. D. Herbsleb and A. Mockus, "An empirical study of speed and communication in globally distributed software development," *IEEE Trans. Softw. Eng.*, vol. 29, no. 6, pp. 481–494, Jun. 2003.

[40] J. D. Herbsleb, A. Mockus, and J. A. Roberts, "Collaboration in software engineering projects: A theory of coordination," in *Proc. 27th Int. Conf. Inf. Syst.*, 2006, p. 38.

[41] S. R. Hiltz and M. Turoff, "Structuring computer-mediated communication systems to avoid information overload," *Commun. ACM*, vol. 28, no. 7, pp. 680–689, 1985.

[42] B. Jiang, B. Jiajun, and C. Chen, "Providing awareness of cooperative efficiency in collaborative graphics design systems through reaction mining," *J. Comput.*, vol. 3, no. 10, pp. 101–108, 2008.

[43] B. K. Kasi and A. Sarma, "Cassandra: Proactive conflict minimization through optimized task scheduling," in *Proc. 35th Int. Conf. Softw. Eng.*, 2013, pp. 732–741.

[44] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *Proc. 14th Int. Symp. Found. Softw. Eng.*, 2006, pp. 1–11.

[45] A. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development teams," in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 344–353.

[46] S. B. Kotsiantis, "Supervised machine learning: A review of classification techniques," *Informatica*, vol. 31, pp. 249–268, 2007.

[47] R. Kraut and L. Streeter, "Coordination in software development," *Commun. ACM*, vol. 38, no. 3, pp. 69–81, 1995.

[48] K. Krippendorff, *Content Analysis: An Introduction to Its Methodology*. Newbury Park, CA, USA: SAGE, 2012.

[49] I. Kwan, M. Cataldo, and D. Damian, "Conway's law revisited: The evidence for a task-based perspective," *IEEE Softw.*, vol. 29, no. 1, pp. 90–93, Jan./Feb. 2012.

[50] I. Kwan and D. Damian, "Extending socio-technical congruence with awareness relationships," in *Proc. Int. Workshop Social Softw. Eng.*, 2011, pp. 2–11.

[51] I. Kwan, A. Schröter, and D. Damian, "A weighted congruence measure," in *Proc. Workshop Socio-Tech. Congruence*, 2009, pp. 1–10.

[52] I. Kwan, A. Schröter, and D. Damian, "Does socio-technical congruence have an effect on software build success? A study of coordination in a software project," *IEEE Trans. Softw. Eng.*, vol. 37, no. 3, pp. 307–324, May/Jun. 2011.

[53] R. Lotufo, Z. Malik, and K. Czarnecki, "Modelling the 'hurried' bug report reading process to summarize bug reports," in *Proc. 28th Int. Conf. Softw. Maintenance*, 2012, pp. 430–439.
[54] W. Maalej and M. Robillard, "Patterns of knowledge in API reference documentation," *IEEE Trans. Softw. Eng.*, vol. 39, no. 9, pp. 1264–1282, Sep. 2013.
[55] P. Markopoulos, B. de Ruyter, and W. E. Mackay, "Awareness systems: Known results, theory, concepts and future challenges," in *Proc. CHI Extended Abstracts Human Factors Comput. Syst.*, 2005, pp. 2128–2129.
[56] S. Minto and G. C. Murphy, "Recommending emergent teams," in *Proc. 4th Int. Workshop Mining Softw. Repositories*, 2007, p. 5.
[57] A. Mockus, R. T. Fielding, and J. Herbsleb, "A case study of open source software development: the Apache server," in *Proc. Int. Conf. Softw. Eng.*, 2000, pp. 263–272.
[58] G. C. Murphy and E. Murphy-Hill, "What is trust in a recommender for software development?" in *Proc. 2nd Int. Workshop Recommendation Syst. Softw. Eng.*, 2010, pp. 57–58.
[59] I. Omoronyia, J. Ferguson, M. Roper, and M. Wood, "Using developer activity data to enhance awareness during collaborative software development," *Comput. Supported Cooperative Work* 18, no. 5–6, 2009, pp. 509–558.
[60] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972.
[61] S. Rastkar and G. Murphy, "Summarizing software artifacts: a case study of bug reports," in *Proc. 32nd Int. Conf. Softw. Eng.*, 2010, pp. 505–514.
[62] A. Sarma, L. Maccherone, P. Wagstrom, and J. Herbsleb, "Tesseract: Interactive visual exploration of socio- technical relationships in software development," in *Proc. 31st Int. Conf. Softw. Eng.*, 2009, pp. 23–33.
[63] A. Sarma, D. Redmiles, and A. van der Hoek, "Empirical evidence of the benefits of workspace awareness in software configuration management," in *Proc. 16th Int. Symp. Found. Softw. Eng.*, 2008, pp. 113–123.
[64] A. Sarma, D. F. Redmiles, and A. van der Hoek, "Palantir: Early detection of development conflicts arising from parallel code changes," *IEEE Trans. Softw. Eng.*, vol. 38, no. 4, pp. 889–908, Jul./Aug. 2012.
[65] S. Sawyer, "Software development teams," *Commun. ACM*, vol. 47, no. 12, pp. 95–99, 2004.
[66] W. Scacchi, "Free/open source software development: Recent research results and methods," *Adv. Comput.*, vol. 69, pp. 243–295, 2007.
[67] M. E. Sosa, S. D. Eppinger, and C. M. Rowles, "The misalignment of product architecture and organizational structure in complex product development," *Manag. Sci.*, vol. 50, no. 12, pp. 1674–1689, 2004.
[68] M. A. Storey, D. Davor Čubranić, and D. M. German, "On the use of visualization to support awareness of human activities in software development: A survey and a framework," in *Proc. Symp. Softw. Vis.*, 2005, pp. 193–202.
[69] P. Sullivan, "Information overload: Keeping current without being overwhelmed," *Sci. Technol. Libraries*, vol. 25, no. 1–2, pp. 109–125, 2004.
[70] S. Wong, Y. Cai, G. Valetto, G. Simeonov, and K. Sethi, "Design rule hierarchies and parallelism in software development tasks," in *Proc. Int. Conf. Automated Softw. Eng.*, 2009, pp. 197–208.

Kelly Blincoe received the BE degree in computer engineering from Villanova University in 2004, the MS degree in information science from Pennsylvania State University in 2008, and the MS and PhD degrees in computer science from Drexel University in 2011 and 2014, respectively. She is currently a postdoctoral fellow at the University of Victoria. She was previously with Lockheed Martin as a proposal manager and software engineer. Her research interests lie in collaborative software engineering and computer-supported cooperative work. She is a member of the IEEE.

Giuseppe Valetto received the Laurea degree in electronic engineering from Politecnico di Torino, Italy in 1992, and the MS and PhD degrees in computer science from Columbia University, US, in 1994 and 2004, respectively. He has mostly worked on collaborative software engineering and distributed and self-adaptive systems. He has held positions at Xerox Research, CEFRIEL, Telecom Italia Lab, IBM T.J. Watson Research Center, and Drexel University. He is currently a researcher in the Distributed Adaptive Systems (DAS) unit at Fondazione Bruno Kessler, Italy. He is a member of the IEEE.

Daniela Damian received the BSc degree in computer science from the Babes Bolyai, Romania, in 1995, and the MSc and PhD degrees from the University of Calgary, Canada, in 1997 and 2001, respectively. She is currently a professor in the Department of Computer Science, University of Victoria, where she leads the Software Engineering Global interAction Laboratory (SEGAL, thesegalgroup.org). Her research interests include software engineering, requirements engineering, computer-supported cooperative work, and empirical software engineering. She has served on the program committee boards of several software engineering conferences and was the program co-chair for the First International Conference on Global Software Engineering (ICGSE06). She is currently serving on the editorial boards of the Journals *Transactions on Software Engineering, Requirements Engineering*, *Empirical Software Engineering*, and *Software and Systems*. She is a member of the IEEE.