Timely and Efficient Detection of Coordination Needs to Support Collaboration among Software Developers

Kelly Blincoe
Computer Science Department
Drexel University
kelly.blincoe@drexel.edu

ABSTRACT

Developers in large software projects often need to coordinate when they are working on interdependent tasks. However, those developers are not always aware of those coordination needs. When that coordination does not occur, productivity can be affected. There are existing methods that detect coordination needs between developers, but they suffer from two main limitations: they are not timely, and they may overwhelm the developers with many recommendations. I have devised a method that solves these limitations and detects only the critical coordination needs in a timely way.

1. PROBLEM / MOTIVATION

In large software projects, developers work on tasks in parallel, or on tasks that are interdependent. This often results in work dependencies and, consequently, coordination needs. When developers remain unaware or do not obtain timely awareness of the coordination that is required to manage work dependencies, there is potential for software productivity or quality problems [6][23].

The design literature, beginning with Parnas' recognition of the workflow implications of modularization [23], has largely focused on ways to streamline the technical dependencies between modules as a way to maximize task parallelism [27][28]. However, it is impossible to eliminate all technical dependencies between modules. Therefore, some coordination between development tasks that operate on those modules must occur. In large, distributed development projects, it is particularly difficult for developers to stay aware of their coordination needs [7]. Developers must be made aware of the dependencies that exist and the coordination that is required of them.

There has been some research on methods [8][10][21][29] and tools [2][12][13][22][24][25][26] to detect coordination requirements and increase awareness of coordination needs. However, the state of the art in this area is limited in two principal ways:

1. Detection of coordination requirements is not timely. Recognition of coordination needs occurs when much of the affected development work has already taken place.
2. Detection methods tend to introduce too much noise and risk to overwhelm the developers since they enumerate all potential coordination requirements between pairs of developers. A large number of these needs may be already known, may have a low impact on efficiency of collaboration, or may be immaterial for the developers' work.

My work begins to solve these two limitations. I describe a technique to identify coordination requirements in real-time and methods to differentiate between trivial and critical coordination needs. I define a critical coordination requirement as one that can cause the most disruption and inefficiency to the development process if not properly and timely managed. Therefore, my work contributes to increasing the productivity of software development teams. The methods I have conceived can be generalized to enhance coordination awareness and increase efficiency to other industries that complete a majority of their work electronically.

2. BACKGROUND / RELATED WORK

A recent work shows that, for a software engineer, the most important form of awareness is locating and keeping up to date with other developers whose work is relevant to her own [2]. One of the first approaches to providing this type of awareness was introduced with configuration management conflict detection systems like Palantír [24][26] and CollabVS [13]. These tools are built on top of configuration management tools and changed the flow of information from "pull" to "push" by providing notifications to keep a developer abreast with what happens in her colleagues' workspaces. They issue instantaneous warnings to developers as an individual instance of conflict emerges, but they do not offer a model for quantifying the strength of coordination needs. So while they are timely, CollabVS and Palantír do not draw a complete view of coordination needs across the whole team and simply provide a stream of notifications to developers.

Cataldo et al. [8][10] were the first to introduce a framework for establishing coordination requirements between developers (depicted in Figure 1). They found that when coordination requirements are fulfilled, for example, by acts of communication, productivity is likely to increase [8][9][10]. Their framework establishes coordination requirements between developers who are working on dependent tasks. To ascertain work dependencies they look at the artifacts committed during each task and the dependencies between those artifacts. Logical coupling [16] is used to determine technical dependencies between artifacts if they have been checked in together in the past. Cataldo et al. [9] found that logical couplings are more likely than syntactic couplings to

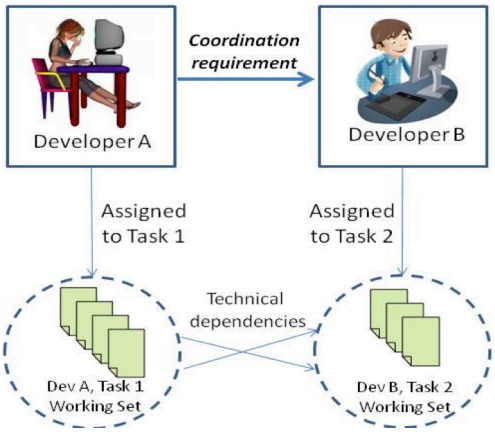

Figure 1. Conceptualization of coordination requirements in Cataldo et al. approach [8][10].

provide a reliable representation of technical dependencies for their coordination requirement conceptualization. A limitation of this conceptualization is that it requires mining the source control repository of the project for the commit history of software artifacts. This type of data is typically available only towards the end of the development work for a task, and the coordination awareness garnered from this approach may not be actionable by the developers at the time coordination is needed to reap those performance benefits. Several other techniques have been proposed to detect the need to coordinate between developers in large software development projects [21][29], but they suffer from the same limitation.

Examples of systems that try to achieve awareness by employing abstractions similar to CRs include Ariadne [12], EEL [22], Tesseract [25], and Codebook [2]. Those systems offer various mechanisms including visualization of socio-technical networks (Ariadne and EEL), dashboards (Tesseract), and query and search facilities (Codebook) to try to identify and show important work relationships within a team. All of those systems depend on establishing technical dependencies among artifacts using commit data in the source code repository. They then use these dependencies to compute relationships between developers. Therefore, these tools are unable to provide timely notifications that can raise the developers' awareness.

Another drawback of current coordination requirement detection methods (and the associated tools) is that they do not indicate which tasks are involved in coordination needs. They detail only the developers who may need to coordinate. Developers may work on multiple tasks at the same time, so coordination requirements at the developer level may encompass the work dependencies of many tasks. This puts the burden on the developers to identify what to coordinate about. If awareness tools were able to provide finer-grained coordination needs at the task level, that burden would be removed.

To begin to address this limitation, Kwan et al. [21] proposed an enhanced weighted communication model which counts the number of communications that occur and the content of those communications to better understand which technical dependencies have been fulfilled. However, it is difficult and error-prone to identify which specific task dependencies are being discussed in coordination traces. In addition, Kwan's model does not propose ways to distinguish between trivial and critical coordination needs.

In the remainder, I present my solution to these problems. In Section 3, I propose methods to identify coordination needs in real-time. In section 4, I introduce techniques to begin to differentiate between trivial and critical coordination needs. Finally, in Section 5, I discuss the significance of my results and how my methods contribute to the field of computer science.

3. TIMELY DETECTION OF COORDINATION REQUIREMENTS

3.1 Approach and Uniqueness

With existing methods, detection of coordination requirements is not timely. To address this limitation, we created an alternative conceptualization of coordination needs in software development teams (shown in Figure 2) using our *proximity* metric. Unlike the more traditional coordination requirement detection techniques, it does not obtain information from the source control repository system nor rely on technical dependencies between artifacts. These differences make proximity timely and turn coordination requirements into an actionable concept for managing coordination in software projects.

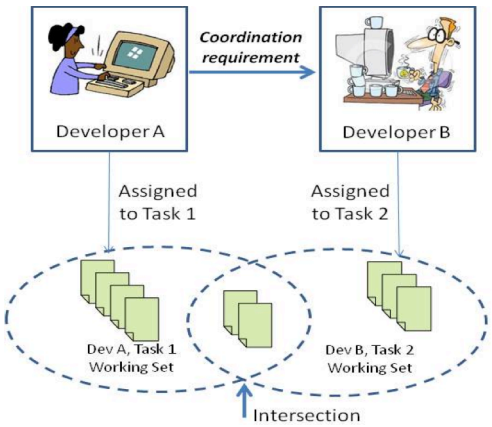

Figure 3. Conceptualization of Coordination Requirements through Proximity [3].

To determine coordination requirements, the proximity algorithm examines the similarity of artifact working sets as they are constructed during developers' tasks. To do this, it obtains developer actions such as artifact consultation or edits as they occur. It uses the Mylyn framework [18][19] to obtain this information. Mylyn is a tool that transforms a developer's Individual Developer Environment (IDE) to a task-centric view to make context switching between tasks easier. To fulfill its own purposes, Mylyn records all developer IDE interactions as they occur. These events are stored as context data for the task in focus. For convenience, ProxiScientia [5], the tool which implements the Proximity measure, is built on top of Mylyn so it can easily obtain these developer actions.

The proximity measure looks at artifact consultation and modification activities captured by Mylyn and weighs the overlap that exists between working sets associated to pairs of developers. It considers all actions recorded for each artifact in each working set in order to apply a numeric weight to that artifact's proximity

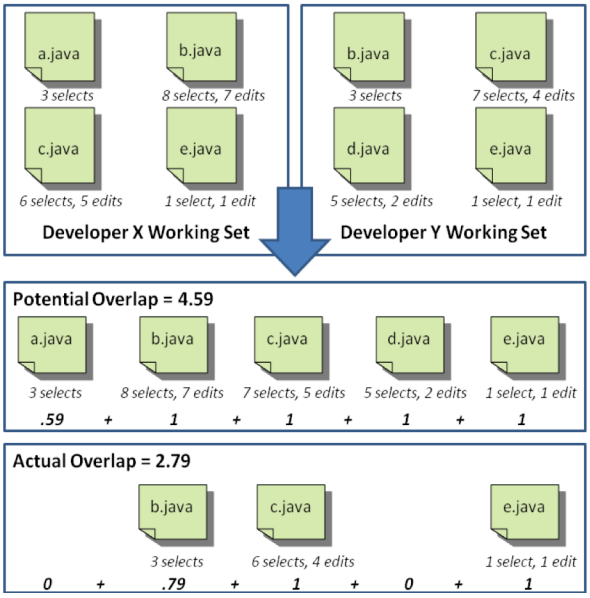

Figure 3. Proximity Algorithm Example [3].

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 and the least amount of weight is given when an artifact is simply consulted in both working sets. The weights are directly based on the weights Mylyn itself uses for its degree-of-interest model [18][19]. The weights are shown in Figure 3 with an illustration of an example of the proximity computation process [3]. The algorithm then computes the ratio of actual to potential overlap. Actual overlap considers the intersection of the two working sets while potential overlap considers the union of the two working sets. Potential overlap represents the maximum possible proximity score had there been perfect overlap between the two sets of actions.

3.2 Results

To evaluate proximity, I performed an empirical study [3] that compared proximity to the CRs detected by the Cataldo et al. [10] detection algorithm. The Cataldo et al. approach was selected for comparison since it is the most well-known method for detecting coordination requirements, and many of the awareness tools created to detect coordination requirements are based on this method. Our study examined the open source project that developed the Mylyn framework itself. I found that higher values of proximity correlate with the likelihood of a CR. Table 1 shows these correlations for two different units of work. The Mylyn context event identifies the file name, class name and even the name of the class element (method or attribute) when available. So, we can choose to consider artifacts at different granularity levels allowing us to determine if two developers were working on the same area of code within a large file.

Table 1. Proximity vs CRs Coorelations

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

I found that proximity has high levels of precision and recall when matched to the coordination requirements computed with the Cataldo et al. approach which I used as the best available approximation of ground truth (see Table 2). I examined the cases when proximity scores did not align to those CRs, and all cases examined turned out to be false positives or negatives of the traditional CR detection method. More importantly, several of those cases highlight drawbacks of that methods' reliance on post-mortem information and dependency conceptualizations.

Table 2. Proximity vs CRs Precision/Recall

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

For example, a coordination requirement is established between developers 6 and 7 in release 3.2 using the Cataldo et al. method. Developer 6 committed BugzillaClient.java, while developer 7 committed BugzillaTaskEditorPage.java. The changes by developer 6 involve a character encoding method that is private to the BugzillaClient class. Developer 7 added a new section to the Mylyn task editor. Although we could ascertain those changes were semantically unrelated, the two involved files had been historically changed together by other developers often enough to cause a logical dependency to be established by the Cataldo et al. detection algorithm. That coordination requirement is therefore a false positive of the traditional method that our proximity algorithm correctly eliminates.

In another case, involving developers 3 and 7 during release 3.3, proximity contributions came exclusively by selection and mixed overlaps. The pair had seven mixed overlaps and six selection overlaps. Meaning that developers 3 and 7 viewed 13 of the same artifacts, of which seven were edited at some point by either developer 3 or developer 7, but no single artifact was edited by both developer 3 and developer 7. Since there were no overlapping commits, the Cataldo et al. method does not allow for a coordination requirement to be detected. However, since we have the advantage of knowing not only what files are edited but also what files are consulted by a developer in the process of completing a task, our algorithm picks up what is likely to be an actual work dependency. Developer 3 and developer 7 repeatedly examined the same area of the software code base and consulted each other's code during their work for release 3.3.

An even more interesting case is provided by an additional 15 pairs of developers who have proximity but do not have a coordination requirement when using the Cataldo et al. approach. In each of these cases, the developers edit from 1 to 67 of the same artifacts (17 on average). Coordination requirements could not be established in any of these cases because at least one of the developers did not commit her changes. However, task contexts prove that those developer pairs were at one time engaged in development on the very same artifacts - the epitome of a Coordination Requirement. A conclusion of this study is, therefore, that looking at both artifact consultation and editing actions as they occur, it is possible to accurately discover coordination requirements without the need to look at technical dependencies between artifacts.

I also evaluated the timeliness of the proximity approach. To do that I obtained the time when the first contribution to the proximity score would have occurred by examining the timestamps for the overlapping events recorded in the working sets of developer pairs. I then compared the first proximity event with both the first day of concurrent work by that pair and the day in which the first CR is identified for the same pairs using the Cataldo et al. method. I found that proximity significantly

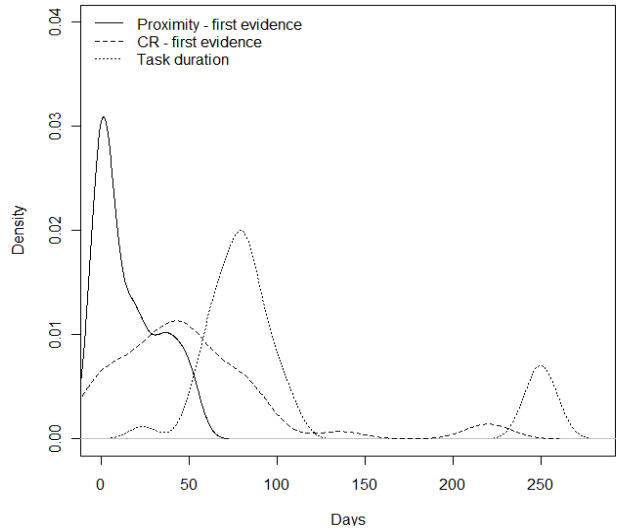

Figure 5. Timeliness Probability Density: Proximity and CRs.

improves the timeliness of CR detection (see Figure 4). For example, in one data set, concurrent work intervals last 31.4 days on average. Proximity is detected on average 6.2 days after parallel work begins while the CR detection method would not detect the CR until 25.2 days on average after the start of concurrent work [3]. This timely detection of coordination needs may provide developers awareness of their coordination needs while their work is still underway. Developers can then act upon and resolve their coordination needs as they surface.

3.2.1 Applying Proximity to Groups

Proximity computes coordination needs between pairs of developers. Coordination, however, can also be a group activity. As large software projects progress, sub-groups often begin to spontaneously form within the organization. These *emergent groups* are informal but can be critical to the performance and success of the project. We leveraged our work on proximity to detect emergent groups, by looking at the intersections of multiple working sets [17].

We did this by computing the artifact working set intersections between development tasks. We constructed a weighted bi-partite network in which nodes represent, on one hand, developers involved in those tasks and, on the other hand, those intersections. The edges are weighted according to the number of artifacts the developer consulted or edited for each of her tasks in the intersection. We then used the edges in the bi-partite network with weights above the median to construct bi-cliques [4]. These bi-cliques capture the groups of developers who tend to consult and manipulate the same artifact sets and, therefore, are most likely to need to coordinate their work with one another. Based on the bi-cliques, we computed a structural correlation matrix between developers. This matrix is a developer-by-developer network in which the weights of the edges represent how similar two developers are in terms of the bi-cliques they are part of. In these networks, groups of nodes (e.g. cliques) linked by highly weighted edges reveal emergent groups.

To evaluate this approach, we constructed these networks for eight releases of Mylyn development. We validated the groups established in these networks with qualitative information on the history of the project and its open-source community, which was collected from project repositories on the web. In addition, we used the communication traces of the team to construct alternate "talk" social networks for comparison to the "work" networks constructed with our approach. These alternate communication-based networks offer a confirmatory view of the Mylyn team obtained through our analysis.

Given the incremental and near-real time nature of the work traces we leverage with our method, this technique can be used to recognize informal teams as they emerge, which leads to actionable management support and better team awareness.

4. IDENTIFYING CRITICAL COORDINATION REQUIREMENTS

4.1 Approach and Uniqueness

Existing techniques to support coordination awareness assume all dependencies may require coordination and enumerate the universe of those potential coordination needs in a project. This can lead to an overwhelming number of recommendations and alert developers of even trivial coordination needs. This is especially problematic in large projects or when coordination requirement detection occurs at fine granularity, for example, at the level of individual tasks. However, detecting coordination needs at a more granular level would provide more information to software developers and allow for more focused coordination.

To address these concerns, we computed the Proximity metric between pairs of tasks. We compared the resulting coordination needs with dependencies that were identified by the Mylyn development team within the Bugzilla records either as explicitly marked dependencies/duplicates or through discussions that link the two tasks. We saw that many of the identified dependencies were detected using the Proximity metric (Recall: 79.6%). However, the identified dependencies constituted only a very small portion of all task pairs with Proximity (Precision: 2.7%). We believe that a major contributor to the low precision is that existing automated techniques that detect coordination needs, like Proximity, find far too many dependencies.

We examined other task properties that could be used to supplement measures like Proximity to identify coordination needs. We looked for differences between identified dependencies and all other task pairs. The task properties we examine included (1) architecture-related properties such as the affected product, component, platform and operating system of the task and (2) characteristics of artifacts involved in each task.

We examined the *architecture-related properties* by checking, for each task pair, if the tasks involved in that pair shared any of those properties (i.e. if they affect the same product, component, platform, or operating system). To *characterize the artifacts* involved in each task, we derived a Design Structure Matrix (DSM) [1] of the Mylyn code base for the two releases of interest. A DSM is a square matrix that identifies technical dependencies between software modules. From that DSM, we computed a Design Rule Hierarchy (DRH) [30], which clusters modules into "layers". These layers can be used to differentiate artifacts that represent influential design decisions from low-level artifacts that depend on (changes to) those decisions. In a DRH, modules in one layer have technical dependencies only on modules in the layers above, whereas modules clustered in the same layer are mutually independent. Consistent with Parnas' definition of modularization [23], these independent modules can be worked on in parallel without incurring coordination overhead. Wong et al. [30] observed that developers working on tasks that involve software modules in different layers of a DRH tend to communicate (a dominant form of coordination in software development [20]) significantly more than developers working only on modules in the same layer. Using properties of the DRH allows us to consider the modularization decisions made in the project code base. We identified the associated DRH layer and module for each of the artifact consultations and edit actions associated with java artifacts for each task. Using this information, we obtained the number of overlapping DRH layers and modules for each task pair.

Finally, we explored supplementing the Proximity algorithm with these properties to infer the most critical coordination needs by applying the k-nearest neighbor machine learning algorithm [11]. The k-nearest neighbor algorithm considers the distance from an unknown pair to each of the pairs in the training set. It then considers a majority vote from the k-nearest neighbors in the training set to decide if the unknown pair is a coordination requirement or not. For this study, we used nine as the k-value. Euclidean distance was used to determine the distance between the unknown pairs and the training set instances.

As a training set, we used a subset of task pairs from the previous release. The task pairs with identified dependencies were the positive examples of coordination requirements. On the other

hand, we selected a subset of 175 pairs that were not identified dependencies as the negatives examples in the training set.

4.2 Results

We analyzed each of the described properties to identify any that appear significantly different between the identified dependencies and all other task pairs for one release of the Mylyn development project. For the architecture-related properties, Chi-squared tests of difference in proportion show that there is a significant difference between the identified dependencies and all other task pairs for all but one of the tested properties: there is not a statistically significant difference for the number of task pairs marked for the same component. For the artifact characterization properties, Mann-Whitney tests of difference in distribution show that the difference is statistically significant for both properties (Table 3). Therefore, we confirmed that there are task and artifact properties that indicate a need to coordinate.

Table 3. Task Property Comparison

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

We then applied the k-nearest neighbor machine learning algorithm to the properties identified as statistically significant. We focused our evaluation on the criticality of the tasks involved in the identified coordination needs since it is difficult to compute precision and recall without a complete picture of ground truth of coordination needs in the project. We are not able to use the identified dependencies because we have found them to be incomplete and heavily weighted towards one particular type of dependency (task decomposition).

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

Our machine learning techniques identified a set of critical coordination requirements between tasks that was 16% of the large set of potential requirements identified by Proximity. The average task duration in this set is almost 10 days longer than the average task duration when considering only proximity. A Mann-Whitney test of the coordination requirements resulting from the different methods shows there is a statistically significant difference in task durations ($W = 8238.5$ and $p = 0.0006$).

5. CONTRIBUTIONS

In large and distributed software development projects, developers are often unaware of their coordination needs. Existing methods to detect coordination needs are not timely and cannot detect coordination needs until a large majority of the development work has already been completed because they rely on commit data obtained from source control repositories. My method provides a live view of coordination needs as they are established by considering software developers' actions on artifacts in real-time. This live view of coordination needs allows developers to act on those coordination needs as they surface. Further, this work contributes a set of task characteristics which are indicative of the need to coordinate and a method to use those properties for the selection of critical coordination needs. This allows developers to focus their attention on tasks where coordination is truly needed.

Using these methods, tools, such as a coordination recommendation engine, can be developed to make developers aware of their coordination needs in real-time while avoiding a large number of false positives. Such a tool will increase awareness of coordination needs in large software development projects. When coordination needs are fulfilled with actual acts of coordination, it has been found that productivity will increase and failures will decrease [8][9][10]. Therefore, this increased awareness can lead to increased productivity and efficiency in the software development process. As an alternative, project leads and managers could use such a tool to identify potential design changes or team structure changes [29] to lower the coordination overhead of the project. Early identification, provided by my method, allows for management action before much of the development work has been completed.

In addition, we have shown how proximity can be extended to identify emerging groups in software development teams. This allows developers and managers to obtain even greater awareness of how teams are coordinating. This insight can allow managers to better allocate tasks by, for example, allocating dependent tasks to a group of individuals who are working well together. It could also allow managers to recognize when an individual's coordination overhead becomes too great.

The lack of awareness of coordination needs is not unique to Software Engineering. The methods described in this paper can be generalized to help identify coordination needs in many other fields that perform their work electronically. Proximity can be computed by identifying the work artifacts that have been viewed or edited by individuals in the same way we looked at consultation and edit events of software artifacts. Properties specific to each industry could then be used to supplement those Proximity scores to identify the most critical coordination needs as was done with the Bugzilla task properties and the DRH characteristics of artifacts in this work. As industries continue to turn to electronic mediums, the need for Computer Science to provide ways to keep people aware of their coordination needs will continue to grow.

6. REFERENCES

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

[2] Begel, A., Phang, K.Y., and Zimmerman, T. 2010 Codebook: Discovering and Exploiting Relationships in Software Repositories. Proc. ICSE 2010.

[3] Blincoe, K., Valetto, G. and Goggins, S. 2012. Proximity: a Measure to Quantify the Need for Developers Coordination. Proc. CSCW 2012.

[4] Borgatti, S.P. and Everett, M.G. Network analysis of two mode data. Soc. Netw. 19(3), 243–269 (1997).

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

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

[7] Cataldo, M., Bass, M., Herbsleb, J., and Bass, L. 2007. On Coordination Mechanisms in Global Software Development. Proc. ICGSE 2007, 71-80.

[8] Cataldo, M, Herbsleb, J., Carley, K. 2008. Socio-Technical Congruence: A Framework for Assessing the Impact of Technical and Work Dependencies on Software Development Productivity. Proc. ESEM 2008.

[9] Cataldo, M., Mockus, A., Roberts, J.A. and Herbsleb J.D. 2009. Software Dependencies, Work Dependencies, and Their Impact on Failures. *IEEE Transactions on Software Engineering*, vol.35, no.6, pp.864-878, Nov.-Dec. 2009.

[10] Cataldo, M., Wagstrom, P.A., Herbsleb, J.D., and Carley, K.M. 2006. Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools. Proc. CSCW 2006.

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

[12] de Souza, C.R., Quirk, S., Trainer, E., and Redmiles, D.F. 2007. Supporting collaborative software development through the visualization of socio-technical dependencies. Proc. GROUP 2007. 147-156.

[13] Dewan, P. and R. Hegde. 2007. Semi-Synchronous Conflict Detection and Resolution in Asynchronous Software Development. Proc. E-CSCW 2007. p. 159-178.

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

[15] Ehrlich, K., Helander, M., Valetto, G., Davies, S., and Williams, C. 2008. An analysis of congruence gaps and their effect on distributed software development. Proc. STC 2008.

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

[17] Goggins, S., Valetto, P., Mascaro, C. and Blincoe, K. Creating A model of the Dynamics of Socio-Technical Groups using Electronic Trace Data. User Modeling and User-Adapted Interaction, 2012.

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

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

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

[21] Kwan, I., Schroeter, A. and Damian, D. A weighted congruence measure. Proc. STC 2009.

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

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

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

[25] Sarma, A., Maccherone, L., Wagstrom, P., and Herbsleb, J. 2009. Tesseract: Interactive visual exploration of sociotechnical relationships in software development. Proc. ICSE 2009.

[26] Sarma, A., Redmiles, D., van der Hoek, A. Palantir: Early Detection of Development Conflicts Arising from Parallel Code Changes. IEEE Transactions on Software Engineering, pp. 889-908, July-Aug., 2012

[27] Sosa, M.E., Eppinger, S.D., and Rowles, C.M. 2004. The misalignment of product architecture and organizational structure in complex product development. Management Science. 50, 12, 1674-1689.

[28] Sullivan, K.J., Griswold, W.G., Cai, Y., and Hallen, B. 2001. The structure and value of modularity in software design. Proc. FSE 2001, pp. 99-108.

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

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