

# Predicting Coordination Structure by Change Impact Analysis

Sunny Wong  
Yuanfang Cai  
Matthew Marron

Technical Report DU-CS-10-03  
Department of Computer Science  
Drexel University  
Philadelphia, PA 19104  
June 2010

# Predicting Coordination Structure by Change Impact Analysis

Sunny Wong, Yuanfang Cai, and Matthew Marron  
Department of Computer Science

Drexel University  
Philadelphia, PA 19460  
Email: {sunny, yfcai, mem98}@cs.drexel.edu

**Abstract**—Empirical studies reveal a strong correlation between the alignment of design structure and organizational structure, and both software quality and productivity. Estimating how people should communicate to accommodate modification requests have been recognized as a prominent problem in software maintenance. In this paper, we contribute a pluggable framework to anticipate coordination requirements based on change scope prediction, supporting interchangeable change scope prediction algorithms either solely based on architecture or co-change patterns. Using modifications requests from eleven releases of the open source Hadoop project, we evaluate our approach by comparing predicted coordination requirements with actual bug tracker communications. In our evaluation, we obtain an average precision of 68% and recall of 48% in accurately predicting coordination requirements, showing that this approach has the potential to assist managers in anticipating problematic coordination when performing software modifications.

## I. INTRODUCTION

Estimating how people should communicate to accommodate modification requests have been recognized as a prominent problem in software maintenance [1]–[5]. A developer needs to figure out who should be contacted to fix a bug; a program manager may need to know how developers should collaborate to accomplish the task and estimate the possibility of non-local communications in a globally-distributed project. Researchers have proposed various techniques to predict coordination requirements (e.g., [3]) or to suggest experts for bug fixing (e.g., [1], [6]). These approaches often rely upon the existence of well-established version history.

In this paper, we contribute a framework that predicts the coordination structure needed to accomplish a modification request (MR) by analyzing the change scope of the MR, and associating the impacted components with their owners. Different from existing work of coordination requirement (CR) prediction, our framework supports predictions from multiple types of dependency structures—derived from version history, source code, or even architecture models. As a result, our approach is applicable to software projects without established version history or newly refactored systems where previous structures became invalid. Different from existing expert recommendation work, we predict a coordination structure, that is, a set of people who should collaborate on a modification request so that a project manager can review the needs of expensive non-local coordination.

Given a *change source*, our approach predicts what other parts of the system may need to be changed, either based on the architectural structure, which can be derived from source code or design model, or based on logical dependencies [7], which are derived from how files changed together in the past. We also present an approach that automatically derives ownership relation from version history. The ownership relation between files and developers can also be provided by the project manager. Given the change scope and ownership relation, our framework predicts a group of developers that are likely to collaborate to accomplish the task.

We evaluate our change-scope-based coordination structure prediction approach by performing a retroactive study on the first eleven versions of the open source Hadoop<sup>1</sup> project. We predict the coordination structure of each modification request using three different change scope prediction techniques. To assess the accuracy of our approach, we calculate the *precision*, *recall*, and  $F_1$  of the predictions, by investigating how people actually communicated on the bug tracking system.

Given the existence of Hadoop version history, we first predict the coordination structure of each modification request based on the logical dependencies derived from co-change patterns, leveraging a logic-based change scope prediction technique [8], [9]. We also derive the ownership relation from the version history. To assess the feasibility and effectiveness of predicting coordination structure from architecture structures only, we perform an evaluation experiment that predicts change scope and hence coordination structure from an architectural model derived from source code, which we call the *DR hierarchy* [10].

We also conduct a third experiment to serve as the baseline of comparison. In this experiment, we investigate the accuracy of coordination structure prediction given a perfect scope prediction algorithm. We use the set of files that is actually modified to resolve the modification request, and call this technique the *oracle prediction* because it is the exact change scope. Our results show that both the logical and architecture prediction achieve similar accuracy as that of oracle prediction, showing the feasibility of predicting coordination structure

<sup>1</sup><http://hadoop.apache.org/common/>

from change impact analysis. In fact, the logical and architecture predictions achieve higher precision values than the oracle prediction, and their  $F_1$  scores are lower than that of the oracle prediction by only 0.1% and 3%.

The rest of the paper is structured as follows. Section II discusses related work. Section III describes our coordination requirement prediction framework and the change scope prediction approaches we use in our evaluation. Section IV details our evaluation and Section V discusses results and threats to validity. Section VI concludes.

## II. RELATED WORK

Our framework is related to the work of coordination requirement identification and expertise recommendation. We clarify our contribution by comparing and contrasting with some existing work.

The most closely related work to our approach is the Emergent Expertise Locator (EEL) tool of Minto and Murphy [2]. EEL leverages Cataldo’s work of coordination requirement [3] (CR) prediction that multiplies a file dependency matrix  $F_D$  with a task assignment matrix  $F_A$ . Each cell  $ij$  of  $F_D$  is populated with the number of times the files  $i$  and  $j$  are committed together in revision history; and each cell  $ij$  of  $F_A$  is populated with the number of times developer  $i$  modifies a file  $j$ . The resulting *expertise matrix*  $C$  contains the *amount of expertise* that each developer  $i$  has to developer  $j$ .

Both Cataldo et al’s CR prediction and Minto and Murphy’s EEL tool exclusively reply upon logical dependencies, that is, how often files change together in the past. Our previous work [11] showed that predicting change scope with logical dependencies may be inaccurate if extensive revision history is unavailable. In contrast, our framework, supported by the pluggable architecture of our tool, *Clio*, allows the swapping of underlying dependency structure and change prediction algorithms [12]. When well-established version history is not available or the system is newly refactored, the user can use architecture dependency structure, which can be derived from either source code or other design models, as the basis of change scope prediction.

By multiplying the  $F_D$  and  $F_A$  matrices in EEL, the logical dependencies between files may inaccurately associate people with files. For example, if files  $A$  and  $B$  are often committed together and Alice frequently works on  $A$  but not  $B$ , the logical dependency between  $A$  and  $B$  may incorrectly identify Alice as an expert on file  $B$ . In our approach, we only associates a person with a file if they directly modified it. Even though  $A$  and  $B$  are often committed together, if the commits made by Alice do not include file  $B$  then our approach would not associate her with file  $B$ . Instead of predicting coordination needs by multiplying matrices as in their work, our prediction is based on change impact analysis, associating coordination structure with predicted change scope.

Minto and Murphy evaluate their approach using modification requests from several open source projects by comparing the recommended experts with developers who actually communicated via the bug tracking system. We follow a similar

approach in our evaluation, but derive file ownership in a different way. Their work derive file ownership by investigating how often a developer commits to the file. As an architecture evolves, simply using the number of times a file is modified by a developer may be insufficient. For example, if Alice modified file  $A$  100 times in the past but has not worked on it in a long time, she may no longer be the person to coordinate with when modifying file  $A$ . To address this issue, Minto and Murphy restricted the period of revision history they analyzed to twelve months.

In our evaluation, we consider both the number of times a component is modified by a developer, and the proportional of total modifications to the file made by the developer (confidence). In other words, if since the time Alice stopped working on file  $A$ , 200 additional modifications have been by other people, the *confidence* we have that Alice is the person to coordinate with is only (at most)  $1/3$ . In many case of close-source project, supported by such tools as Rational Team Concert<sup>2</sup> (RTC), the ownership relation is known. Our pluggable framework allows the program manager of such projects to fill in the ownership relation directly.

Different from Cataldo et al. [3]’s work of predicting coordination structure from the perspective of the overall project, our approach attempts to predict CRs for completing individual tasks. Another example of CR prediction is the study of Morelli et al. [13] on an electronics manufacturing company. Their approach used interviews with project team members to identify types of tasks that would require coordination. In contrast, our approach does not require interviewing team members.

Expertise recommendation systems (e.g., [1], [2], [6], [14], [15]) help identify developers who are *experts* on certain components of the software in order to help triagers decide whom to assign modification requests, help developers find experts with whom to communicate for assistance, etc. While the purpose of our approach is to allow managers to predict CRs based on anticipated communication to experts, our framework can be used by as an expertise recommendation system and expertise recommendation approaches can also be used to predict CRs.

Existing expertise recommendation approaches are often based on heuristics on revision history, social network analysis, or machine learning. Our approach is different in that we recommend coordination structure based on change scope prediction, from either logical or architecture dependency structure. Approaches based on social network analysis (e.g., [16]) often create large networks that need to be pruned to extract the desired expertise information. In contrast, our approach requires the identification of a change source (the files a developer would likely start investigation or modify) for a modification request and automatically reports the potential CRs. Machine learning approaches (e.g., [5]) often require an extensive historical data in order to train the learning algorithm for accurate prediction. In contrast, our approach supports

<sup>2</sup><http://jazz.net/projects/rational-team-concert/>

prediction from architecture structure that can be derived from source code. Our evaluation shows that using data from a single minor release of Hadoop (two months of development time) and an architectural dependency-based change scope prediction algorithm, we can predict CRs with a precision of 82% and recall of 47%.

Bowman and Holt [17] introduced the concept of an *ownership architecture* that associates developers to files, that could be used to locate experts. However, the primary focus of their work was to aid program comprehension and they did not evaluate its effectiveness in identifying experts. They reverse engineered ownership architectures from artifacts such as copyright notices in files and revision history logs, similarly to our approach. However, while they associated a developer to a file after a single modification, in our evaluation, we use the heuristics of support and confidence to determine ownership because developers with more experience working with a file may have more expertise.

### III. APPROACH

In this section, we describe a motivating example and an overview of our coordination prediction approach. We also describe the change scope prediction algorithms we use in our evaluation.

#### A. Motivating Example

Eve is manager, triaging modification requests (MRs) for a software system. For each MR that needs to be assigned to developers for work on, in order to decide which team of developers to assign the task, Eve first determines the first set of components that the MR likely affects, which we call the *change source*. Before making the assignment, Eve feeds the change source into our coordination requirement prediction framework, *Clio*,<sup>3</sup> to anticipate the need for developers to coordinate or communicate. In one the modification task, a defect has been discovered in component *A*, which Eve considers assigning to Alice to fix. Our prediction framework determines that changing component *A* is likely to require changes to components *B*, *C*, and *D*, which Bob, Carol and Dave have frequently worked on in the past respectively. So if Alice is to change component *A*, she would likely communicate and coordinate with Bob, Carol and Dave to seek their expertise on the components' designs, coordinate concurrent changes, etc. As the project manager, Eve may realize that Alice and Bob speak different languages, or Alice and Dave are located in different countries. Based on this output from our framework, Eve can make further decisions about how to facilitate the coordination of this developer group, e.g. by allocating someone else to facilitate the communication between Alice and Bob, or assigning the task to someone else who would have less difficulty in communicating.

#### B. Framework Overview

Our coordination requirement prediction framework takes as input: (1) the dependency structure of the system, either the revision history from which logical dependency can be derived, or the source code or design model from which architecture dependency can be derived. (2) a modification request. We call the set of files that a developer would first start investigating or modifying to fulfill a modification request as the *change source*. A triager would typically identify the change source by examining the MR's description. Various techniques (e.g., [18], [19]) are available to help identify change sources for modification requests.

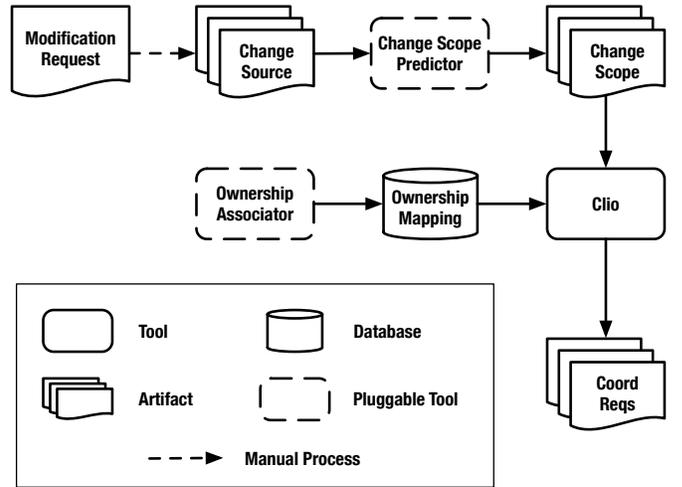


Fig. 1. Approach Overview

Figure 1 presents an overview of our coordination requirement prediction framework, which we implemented as a tool called *Clio*. The *ownership* plugin of *Clio* associates developers to files. A project manager can provide the mapping between developers and the components they are responsible for. If such information is not available, as it is the case in most open source projects, the user can use our approach, as described in the next subsection, to derive ownership relation from version history. Once we have developers associated to files, we use a change scope prediction algorithm [12] (we use the terms change scope prediction and change impact analysis interchangeable throughout this paper) to predict the set of files that is likely to change together to fulfill the modification request. Our pluggable framework allows the swapping of different change scope prediction algorithms (e.g., using design models [20], logical dependencies [9], or program slicing [21]). The *predict-owner* plugin of *Clio* reports the group of people who are likely to coordinate.

#### C. Ownership Analysis

Our *ownership* derivation algorithm builds on the idea of logical dependencies [7], using the heuristics of support and confidence. Consider a file *A* that has been modified  $n$  times (based on revision history), of which  $k$  modifications were

<sup>3</sup>Clio is the Greek muse of history

performed by Alice. Then we say that Alice owns file  $A$  with support  $k$  and confidence  $k/n$ . Given a set of files that we predict will need to change to resolve a modification request (obtained using a change scope prediction algorithm), we select the owners of those files, whose support and confidence are above a minimum support and confidence thresholds, and predict these owners to be the people that the MR developer will communicate with. Hence, if the change scope prediction algorithm predicts that file  $A$  will change to resolve the MR, and Alice owns file  $A$  with high support and confidence, then we predict that whoever works on the MR will have a coordination requirement with Alice.

#### D. Change Scope Prediction

This subsection describes the two change scope prediction algorithms used in our CR prediction framework. As described above, the change scope prediction is applied to the change source of a modification request. We first discuss the change scope prediction using logical dependencies. Then we describe our recently proposed change scope prediction algorithm that is based on the design rule theory [22].

##### Logical Dependency-based Change Scope Prediction

Following the previous research by Ying et al. [8] and Zimmermann et al. [9], we implement a change scope prediction algorithm based on change coupling (or logical dependencies) between files. Consistent with Zimmermann et al., the *frequency* of a set in a set of transactions  $T$  is  $frq(T, x) = |\{t \in T : x \subseteq t\}|$ . The *support* of a rule,  $x_1 \Rightarrow x_2$ , by a set of transactions  $T$  is  $sup(T, x_1 \Rightarrow x_2) = frq(T, x_1 \cup x_2)$ . The *confidence* of a rule is  $conf(T, x_1 \Rightarrow x_2) = \frac{frq(T, x_1 \cup x_2)}{frq(T, x_1)}$ . A file is predicted to be in the change scope if the corresponding co-change pattern's support and confidence are above a minimum support and confidence thresholds.

##### Design Rule-based Change Scope Prediction

We recently proposed an algorithm for predicting change scope for the purpose of detecting *design rule violations* [23]. In this paper, we describe a variation of that algorithm for our CR prediction approach. While the algorithm we used for design rule violation detection explicitly ignored certain parts of the design (the design rules [22]) when predicting change scope, the change scope algorithm we use in this paper does not discriminate against design rules in prediction. Our change scope prediction algorithm leverages Robillard's software dependency analysis algorithm [24] and our previous work on design rule hierarchies [10]. By analyzing the topology of a software dependency graph, Robillard's algorithm recommends relevant code from initial elements of interest (change source). We applied a slight variation of his algorithm to a directed-acyclic graph (called the DR hierarchy) that we previously presented [10]. A DR hierarchy can be automatically derived from a design model such as UML, source code, or compiled binaries.

This DR hierarchy graph provides a useful property that we leverage in our change scope prediction algorithm. Each vertex

of the DR hierarchy aggregates a set of program elements, approximating an independent task assignment; in other words, each vertex contains a set of design decisions that can and should be made together, and vertices in the same hierarchy level can be assigned as tasks to be completed in parallel. Edges represent a potential pairwise dependence relation—such that if there is an edge  $u \rightarrow v$  then a change to  $u$  may require a change to  $v$ . Because of the property of independent tasks, we assume that if one program element in a vertex is in our change scope, then all program elements in that vertex are in also our change scope.

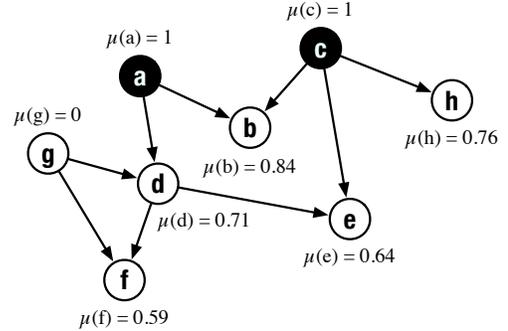


Fig. 2. Example DR Hierarchy Graph

Figure 2 shows an example DR hierarchy graph, which we use to illustrate our design rule-based change scope prediction algorithm. The vertices with shaded background and white text are the change source extracted from the modification request. Starting from these vertices, we assign a weight  $\mu$ , in the range  $[0, 1]$ , to each vertex, in a breadth-first order. The change source vertices are assigned the maximum weight of 1 and added to a set  $S$ , which we call the set of interest. Starting from each vertex in the change source, we examine its neighbors and assign them a weight. We iteratively assign weights to vertices in a breadth-first ordering, and select the elements of highest weight to be in the change scope.

Robillard [24] defines a formula for computing the weight of a vertex:

$$\mu_0 = \left( \frac{1 + |S_{forward} \cap S|}{|S_{forward}|} \cdot \frac{|S_{backward} \cap S|}{|S_{backward}|} \right)^\alpha$$

Consider computing the weight of  $b$ . Here  $S_{forward} = \{b, d\}$  is the set of neighbors, through outgoing edges, of  $a$  and  $S_{backward} = \{a, c\}$  is the set of neighbors, through incoming edges, of  $b$ . The value of  $\mu_0$  is scaled by a constant  $\alpha$  (defined to be 0.25 in both Robillard's work and our evaluation), however the value of this constant does not affect the order of suggested elements. Using this formula, the weight of  $b$  is  $\mu(b) = \mu_0(b) \cdot \mu(a) = \left(\frac{1+0}{2} \cdot \frac{2}{2}\right)^{0.25} \approx 0.84$ . Repeating this, we can assign a weight to each neighbor of  $a$ . Then we compute the weights for the neighbors of  $c$ . In computing the weight of  $e$ , we have  $S_{forward} = \{b, e, h\}$  and  $S_{backward} = \{c, d\}$ . So that  $\mu(e) = \mu_0(e) \cdot \mu(c) = \left(\frac{1+0}{3} \cdot \frac{1}{2}\right)^{0.25} \approx 0.64$ . Since we already assigned a weight to  $b$ , through  $a$ , we do not need to

consider it again. After assigning a weight to the last neighbor  $h$  of  $c$ , we are finished with one iteration of the algorithm. Basically, by using Robillard’s formula, we assign a higher weight to vertices that share more edges with elements in the set of interest  $S$ .

To start the next iteration of our algorithm, we take all the vertices that have just been assigned weights, add them to the set of interest  $S$ , and use them as the starting points for weight assignment. For example, since  $d$  was assigned a weight in the previous iteration, we next compute the weights of its neighbors, of which only  $f$  does not already have a weight. So  $S = \{a, b, c, d, e, h\}$ ,  $S_{forward} = \{e, f\}$ , and  $S_{backward} = \{d, g\}$ , giving us a weight for  $f$  of  $\mu(f) = \mu_0(f) \cdot \mu(d) = (\frac{1+1}{2} \cdot \frac{1}{2})^{0.25} \cdot (\frac{1}{8})^{0.25} \approx 0.59$ . In scaling the weight of  $f$  by the weight of  $d$ , we basically assign lower weights to vertices that are further away from change source.

We repeat this process of iteratively assigning weights to vertices until the new weights fall below a certain threshold. All vertices that were not assigned a weight are considered to have the minimum weight of 0. Figure 2 shows the weights for each vertex after all weights are assigned. The vertices whose weights are above the threshold are then recommended as being in the change scope. Mapping the files in the predicted change scope to their owners, a group of people who should communicate and coordinate to accomplish the modification request can be obtained.

#### IV. EVALUATION

This section describes the procedure we used for evaluating our coordination requirement prediction approach and the results we obtained. We would ideally evaluate our approach by assessing its accuracy as managers and developers use our tool as part of their daily work. However, such an evaluation would require deploying our tool in a non-trivially-sized development team, and convincing a team to adopt a tool without preliminary evaluation of its effectiveness would be difficult. To provide initial effectiveness results, we performed a retroactive study on the open source Hadoop project.

##### A. Subject

Hadoop is an open source map/reduce system for distributed computing, written in the Java programming language. We choose this project because it employs an effective bug tracking system, JIRA,<sup>4</sup> that allows us to extract the comments posted by developers and the files involved in resolving the modification request, for comparison against the predicted coordination requirements. As another motivation for us choosing to study Hadoop, almost all transactions in the revision history included a comment stating who contributed the code. Indication of contributors is important because the person to commit a transaction into revision history is often not the same person who implemented the code to fulfill the modification request. Identifying the contributor allows us to more accurately associate owners to files. Since the

Hadoop developers followed a strict format for specifying this contributor information, it could be easily and automatically extracted to identify ownership.

Table I shows the number of modifications requests we analyzed and the approximate size of each Hadoop release. We categorize each modification request based on the release in which it was resolved (e.g., 19 modification requests were resolved for the 0.4.0 release). With the exception of the first release, each subsequent version of Hadoop was developed and released in approximately one month intervals. The first version was released after two months of development. Hence, our study covered a period of about 13 months.

TABLE I  
HADOOP VERSION STATISTICS

Version	SLOC	# Modification Requests
0.1.0	13,000	25
0.2.0	19,200	14
0.3.0	20,400	14
0.4.0	21,300	19
0.5.0	23,600	11
0.6.0	25,700	15
0.7.0	28,000	14
0.8.0	28,900	9
0.9.1	30,300	17
0.10.0	33,500	18
0.11.0	37,100	23
		<i>Total: 179</i>

During the time period that we studied, there were 188 developers who contributed code or posted comments on the 179 modification requests analyzed. On average, a modification request was closed three weeks after it was opened and contained ten posted comments. For a modification request the number of developers posting comments ranged from 1 to 13, with an average of 5.

##### B. Evaluation Procedure

To assess the accuracy of our CR prediction approach, for each modification request, we compared the predicted coordination requirements (set of developers predicted to communicate) with the set of developers that actually communicated in the bug tracking system. We used the comments posted on the modification requests as actual communication to compare against because, for the most part, this communication is relevant to resolving a MR. For each modification request, we manually identified the program elements that comprise the change source by reading the MR description, as a triager would in using our framework. The change source was then used by the change scope prediction algorithm to identify the files that were likely to be modified and coordination requirements were predicted based on file ownership.

We used precision, recall, and  $F_1$  for assessing the accuracy of CR predictions. Precision is the percentage of the developers that our approach predicted, who actually communicated. Recall is the percentage of the communicating developers that our approach was able to predict. It is well known that there is a trade-off between precision and recall. Predicting more CRs would likely increase recall but decrease precision, since

<sup>4</sup><http://www.atlassian.com/software/jira/>

many of the predicted CRs would be incorrect. On the other hand, predicting few CRs would likely decrease recall but increase precision, since each correctly predicted CR increases precision by a larger percentage. Hence, we use the  $F_1$  score as a single metric that combines both precision and recall. The  $F_1$  score provides a single value for ease of assessment and comparison.

$$Precision = \frac{\# \text{ Correct Predictions}}{\# \text{ Predicted CRs}}$$

$$Recall = \frac{\# \text{ Correct Predictions}}{\# \text{ Actual CRs}}$$

$$F_1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

### Change Scope Prediction

We perform three experiments with multiple change prediction algorithms to see how they affect our framework’s ability to accurately predict coordination requirements. One of our change scope predictions is based on logical dependencies. As Section III describes, this algorithm predicts change scope based on how files change together in the past, using the heuristics of support and confidence. We refer to the experiment using logical dependencies for change scope prediction as the *logical* experiment. In ensuring a realistic setting of using our tool for *predicting* CR, when applying to MRs in version  $n$ , we only use the revision history up to the release of version  $n - 1$ .

Another change scope prediction algorithm we use, as Section III describes, is based on design rule theory [22]. This algorithm predicts change scope based on the topology of dependencies, assign a weight to each program element. We implement this prediction algorithm as a plugin in our modularity analysis framework, Minos [10]. Hence, we refer to the experiment that uses this algorithm for change prediction as the *Minos* experiment. To ensure our tool is used in a predictive setting, when applying to an MR for version  $n$ , we use the dependency structure from the version  $n - 1$  architecture for predicting change scope.

Since both the logical experiment and Minos experiment rely on heuristic threshold values for their change scope prediction, we need to optimally select these threshold values while maintaining a setting that our tool is being used for predictive purposes. That is, we do not want to find the best heuristic thresholds for a version  $n$  and rerun experiment on version  $n$  with these threshold values, since developers cannot time-travel to duplicate such an approach. Instead, we optimize these thresholds based on the previous release’s MRs. That is, we find the best heuristic threshold values for predicting change scope in version  $n - 1$  and use those values for predicting change scope in version  $n$ , adjusting the threshold after each release.

In addition to these two change scope prediction algorithms, we perform an experiment to see how our CR prediction

approach would perform given a “perfect” change scope prediction algorithm. That is, we want to know how accurately our approach can predict CRs if we know the exact set of files that will need to be modified to resolve the modification request. Since this is a retroactive study, we do know, from revision history, the set of files that change to resolve the MR. We refer to this change scope prediction algorithm as *oracle prediction* and we refer to the experiment that uses it CR prediction as the *oracle* experiment.

### Ownership Analysis

Since our framework allows different methods for identifying ownership of files, we use a simple analysis of revision history to derive ownership for our evaluation. Similar with many heuristics-based expertise recommendation approaches, we use *experience* as the measure of a developer’s expertise/ownership on a file. The more often a developer modifies a file, the more we associate the developer with on that file. As described in Section III, we use the heuristics of support and confidence to quantify this experience. Similarly to how we select heuristic thresholds for change scope prediction, we optimize the support and confidence values for ownership based on the previous release’s data. That is, after each release, we determine the support and confidence values that gives us the most accurate CR predictions and use those values for evaluating our approach on the next release.

We developed a plugin for Clio that analyzes the revision history, checking each transaction for who contributed the code. This plugin first checks the commit message for specification of who contributed the code. If no one is specified as having contributed the code, then we assume the developer who committed the files is the contributor. In associating ownership, we disregard transactions with a large number of files (30 files for this evaluation) because they are often branch or merge operations in the repository, and these operations generally do not indicate any expertise on the files. Along the same vein, we also disregard any transaction that explicitly states a branch or merge operation. Additionally, we also disregard transactions that only change the formatting of code, copyright information in comments, etc., as these transactions also do not give indicate of experience or expertise.

Since developers often use multiple aliases when interacting with the revision control system and bug tracking system (e.g., John Doe may have a username of *jdoe* in the revision control system and *john.doe* in the bug tracking system), we need to be able to determine that these aliases are actually the same person in order to accurately predict CRs and assess the predictions. That is, if our approach predicts a coordination requirement with *jdoe* and *john.doe* posts a comment on the MR, then we need know the CR was correctly predicted. Although there are various algorithms that try to automatically perform this de-aliasing (e.g., Top et al. [25]), we manually performed this task since the number of aliases was not too large. We identified 188 unique developers among 315 different aliases in Hadoop.

### C. Results

We ran our experiments on a 2.53GHz Intel Core 2 Duo MacBook Pro with 4GB of RAM. Running our ownership analysis took 15 minutes for all releases. And performing the coordination requirement prediction took about half a minute for each releases.

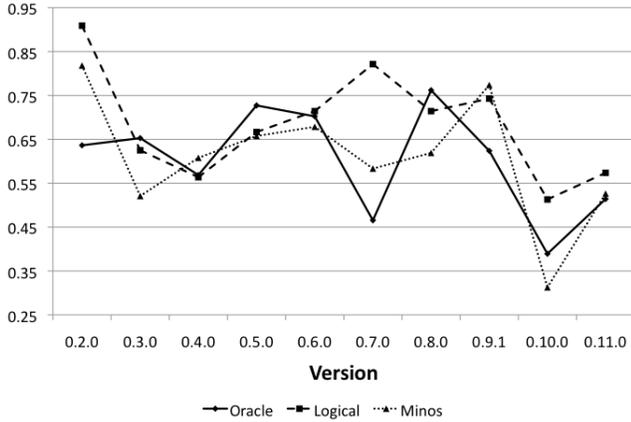


Fig. 3. Precision Comparison

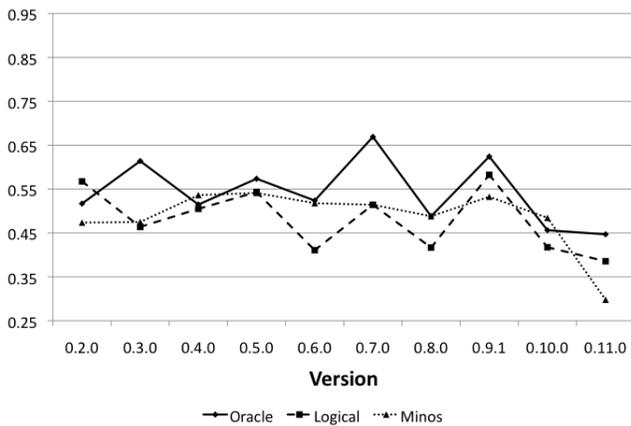


Fig. 4. Recall Comparison

Figure 3 shows the average precision, Figure 4 shows the average recall, and Figure 5 shows the average  $F_1$  score of the different experiments for each release of Hadoop. We do not report the accuracy values for version 0.1.0, because our method for selecting heuristic thresholds is based on the optimizing on the prior release’s data. Since there are no releases prior to 0.1.0 for us to find heuristic thresholds, any data we report for this version would not reflect the accuracy of our approach in a realistic setting. Table II shows the overall average precision, recall, and  $F_1$  values for the three experiments.

Figure 3 shows that, for almost all the releases, all three experiments produced precision values between 55% and 75%. At first, we found it surprising that the precision values for the logical and Minos experiments were higher than the

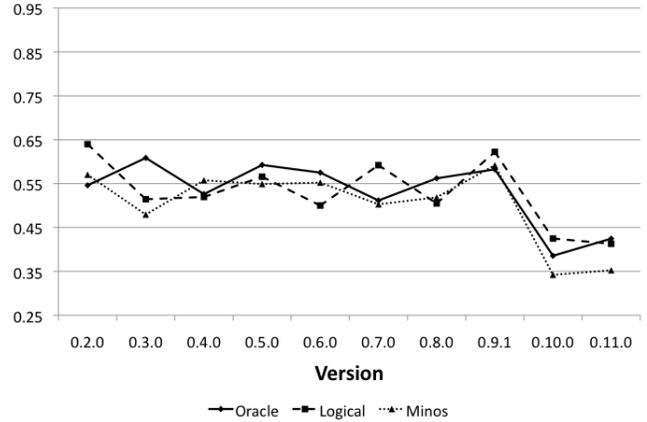


Fig. 5.  $F_1$  Comparison

TABLE II  
AVERAGE PREDICTION ACCURACY

Change Predictor	Precision	Recall	$F_1$
Oracle	0.604	0.543	0.531
Logical	0.684	0.481	0.530
Minos	0.610	0.487	0.502

oracle experiment. However, we found that the logical and Minos experiments generally predicted fewer CRs than the oracle experiment, and hence achieved higher precision. The prediction of fewer CRs is evident from Figure 4—since the logical and Minos experiments predicted fewer CRs, their recall values were generally lower than the oracle experiment.

As we see in Figure 5, the differences in precision and recall between the different experiments averaged out in the  $F_1$  score. All three experiments were comparable in their accuracy of predicting coordination requirements—for example, the average  $F_1$  value for the oracle experiment and the logical experiment, differed by less than  $10^{-3}$ . These results indicate that the use of a change scope prediction algorithm is effective in prediction coordination requirements. Even if we do not know the exact set of files that are going to change to resolve a MR, using a change scope prediction algorithm can give us a reasonable estimate of the CRs for resolving a MR.

### V. DISCUSSION

This section discusses our evaluation results, limitations of our approach, threats to validity, and future work.

A major question about results is whether a precision of 68% and a recall of 48% is good enough to help managers with distributed teams. The only way to know, if our approach is helpful to managers and developers in anticipating problematic coordination requirements, is to perform an empirical study on a moderate-sized, distributed development team where managers are using our tool on a daily basis. We believe the accuracy of prediction reported in this paper is enough to help us convince development teams to try out our tool. Our future work is to study the benefits of using our tool in a live environment.

We only applied our approach to one subject system, so we cannot conclude that our coordination requirement prediction approach generalizes to all software systems. It remains future work to study the accuracy of our CR predictions on other software systems of various sizes and domains.

Since our coordination requirement approach relies on change scope prediction and change scope prediction algorithms use a change source for prediction, the accuracy of our CR prediction heavily depends on the appropriate selection of the change source for each modification request. A poor choice for change source could identify irrelevant files as needing to be modified and thereby identify false CRs. In addition, inappropriate change source selections could miss the identification of certain files and thereby not predict important CRs. We manually identified the change source elements for each modification request. However, not being developers of Hadoop, we cannot guarantee that these elements would be what actual developers would consider as change source for predicting change scope. Evaluating the use of our approach in a live environment with developers selecting change source elements is a future work.

The quality of our coordination requirement prediction depends on accurate ownership information. For our evaluation, we derived ownership from revision history data and requires accurate recording of code contributors. Like many open source projects, the developer who implements the code to resolve a MR is often not the same developer who commits the modified files. Different projects use approaches such as credit files, commit messages, etc. to identify contributors. The technique we used for deriving ownership in our evaluation relies on this contributor information being readily available and accurate. However, our framework allows for different ways of specifying ownership. If contributor information is not available, other techniques can be used. For example, if managers know the owners of certain files, that can be explicitly stated and used as ownership instead of deriving from revision history.

Schuler and Zimmermann [15] recently proposed considering of how often a developer *uses* a method as an indicator of expertise, rather just considering how often a developer *modifies* a method. As Fritz et al. [26] and others have shown, the frequency and recency that developers work on software components do indicate expertise. However, other factors also influence expertise. Our CR prediction framework allows the use of various ownership mapping techniques, which can include other recently proposed factors. Integrating the *uses* experience and other factors into deriving ownership and predicting CRs, remains a future work.

Because we used an *oracle prediction* technique for change scope prediction in one of our experiments, we only considered those modification requests with associated change sets. As Bird et al. [27] showed, the MRs that have associated change sets may not be representative of all MRs in the project. For example, it may have been a coincidence that our approach performs well for the modification requests with associated change sets, but would not have performed well for those

without associated change sets. Applying our approach to additional software systems would reduce the likeliness of our results showing such bias.

In addition, as Aranda and Venolia [28] showed, the coordination requirements automatically extracted from electronic conversations and repositories can be incomplete or inaccurate. That is, using the sets of people who posted comments on the modification requests as the actual set of coordination requirements may not have been completely accurate. Additional developers may have been involved in coordination efforts but their communications were not recorded in the comments. It is our future work to assess the accuracy and effectiveness of our CR prediction approach in a live environment and getting first-hand information from developers on whether the CRs are correct. Performing such an evaluation would potentially reduce any inaccuracy from analyzing electronically recorded communication archives.

There is a natural question of which change scope prediction is better, logic-based or architecture-based. The answer depends on many factors, such as the maturity of the version history and the prediction algorithm in use [11]. In this paper, we contribute a pluggable framework that allows the user to leverage different change scope prediction algorithms. The comparison of these two prediction techniques and selection of an appropriate change scope predictor for a specific software system is out of the scope of this paper.

## VI. CONCLUSION

In this paper, we contributed a framework for predicting coordination requirements of modification tasks, using file ownership and change scope prediction. Our framework uses change scope prediction algorithms to predict a set of files that are likely to be modified to resolve the modification request, and predicts the owners of those files as coordination requirements. We implemented our framework as a tool, *Clio*, with a flexible architecture that allows the swapping of different change scope prediction and ownership identification algorithms to suit specific projects. For example, if extensive revision history is not available for predicting change scope from logical dependencies, an algorithm based on architectural dependencies can be used instead.

To evaluate the accuracy of our CR prediction approach, we applied it to the modification requests in eleven releases of the open-source Hadoop project, comparing predicted CRs to actual communications in the bug reporting system, using different change scope prediction algorithms based on logical dependencies and architectural dependencies. We obtained an average precision of 68% and recall of 48% in predicting coordination requirements, showing that this approach has the potential to assist managers in anticipating problematic coordination when performing software modifications. As a baseline for evaluation, we compared how the different change scope prediction performed in predicting CRs compared to an oracle prediction approach, which uses the exact set of modified files for resolving the modification request. Our results show that both the logical and architecture prediction achieve similar

accuracy as that of oracle prediction—indicating that even if we do not know the exact set of files that are going to change to resolve a MR, using a change scope prediction algorithm can give us a reasonable estimate of the coordination structure for resolving a MR.

## VII. ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation under grants CCF-0916891 and DUE-0837665.

## REFERENCES

- [1] D. W. McDonald and M. S. Ackerman, "Expertise recommender: A flexible recommendation system and architecture," in *Proc. ACM Conference on Computer Supported Cooperative Work*, Dec. 2000, pp. 231–240.
- [2] S. Minto and G. C. Murphy, "Recommending emergent teams," in *Proc. 4th International Workshop on Mining Software Repositories*, May 2007, p. 5.
- [3] M. Cataldo, P. Wagstrom, J. D. Herbsleb, and K. M. Carley, "Identification of coordination requirements: Implications for the design of collaboration and awareness tools," in *Proc. ACM Conference on Computer Supported Cooperative Work*, Nov. 2006, pp. 353–362.
- [4] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb, "Software dependencies, work dependencies, and their impact on failures," *IEEE Transactions on Software Engineering*, vol. 35, no. 6, pp. 864–878, Jul. 2009.
- [5] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proc. 28th International Conference on Software Engineering*, May 2006, pp. 361–370.
- [6] J. Anvik and G. C. Murphy, "Determining implementation expertise from bug reports," in *Proc. 4th International Workshop on Mining Software Repositories*, May 2007, p. 2.
- [7] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Proc. 14th IEEE International Conference on Software Maintenance*, Nov. 1998, pp. 190–197.
- [8] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE Transactions on Software Engineering*, vol. 30, no. 9, pp. 574–586, Sep. 2004.
- [9] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *Proc. 26th International Conference on Software Engineering*, May 2004, pp. 563–572.
- [10] S. Wong, Y. Cai, G. Valetto, G. Simeonov, and K. Sethi, "Design rule hierarchies and parallelism in software development tasks," in *Proc. 24th IEEE/ACM International Conference on Automated Software Engineering*, Nov. 2009, pp. 197–208.
- [11] S. Wong and Y. Cai, "Predicting change impact from logical models," in *Proc. 25th IEEE International Conference on Software Maintenance*, Sep. 2009, pp. 467–470.
- [12] S. A. Bohner, "Software change impacts - an evolving perspective," in *Proc. 18th IEEE International Conference on Software Maintenance*, Oct. 2002, pp. 263–272.
- [13] M. D. Morelli, S. D. Eppinger, and R. K. Gulati, "Predicting technical communication in product development organizations," *IEEE Transactions on Engineering Management*, vol. 42, no. 3, pp. 215–222, Aug. 1995.
- [14] A. Mockus and J. D. Herbsleb, "Expertise browser: A quantitative approach to identifying expertise," in *Proc. 24th International Conference on Software Engineering*, May 2002, pp. 503–512.
- [15] D. Schuler and T. Zimmermann, "Mining usage expertise from version archive," in *Proc. 5th International Workshop on Mining Software Repositories*, May 2008, pp. 121–124.
- [16] R. Sangüesa and J. M. Pujol, "NetExpert: A multiagent system for expertise location," in *Proc. International Joint Conference on Artificial Intelligence*, Aug. 2001, pp. 85–93.
- [17] I. T. Bowman and R. C. Holt, "Reconstructing ownership architectures to help understand software systems," in *Proc. 7th International Workshop on Program Comprehension*, May 1999, pp. 28–37.
- [18] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Extracting structural information from bug reports," in *Proc. 5th International Workshop on Mining Software Repositories*, May 2008, pp. 27–30.
- [19] E. Hill, L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of nl-queries for software maintenance and reuse," in *Proc. 31st International Conference on Software Engineering*, May 2009, pp. 232–242.
- [20] L. C. Briand, Y. Labiche, and L. O'Sullivan, "Impact analysis and change management of UML models," in *Proc. 19th IEEE International Conference on Software Maintenance*, Sep. 2003, pp. 256–265.
- [21] K. B. Gallagher and J. R. Lyle, "Using program slicing in software maintenance," *IEEE Transactions on Software Engineering*, vol. 17, no. 8, pp. 751–761, 1991.
- [22] C. Y. Baldwin and K. B. Clark, *Design Rules, Vol. 1: The Power of Modularity*. MIT Press, 2000.
- [23] S. Wong, Y. Cai, M. Kim, and M. Dalton, "Detecting design rule violations," Drexel University, Tech. Rep. DU-CS-10-01, Apr. 2010, <https://www.cs.drexel.edu/node/15301>.
- [24] M. P. Robillard, "Topology analysis of software dependencies," *ACM Transactions on Software Engineering and Methodology*, vol. 17, no. 4, pp. 18:1–18:36, Aug. 2008.
- [25] P. Top, F. Dowla, and J. Gansemer, "A dynamic programming algorithm for name matching," in *Proc. IEEE Symposium on Computational Intelligence and Data Mining*, Apr. 2007, pp. 547–551.
- [26] T. Fritz, G. C. Murphy, and E. Hill, "Does a programmer's activity indicate knowledge of code?" in *Proc. 11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Sep. 2007, pp. 341–350.
- [27] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and balanced? bias in bug-fix datasets," in *Proc. 17th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Aug. 2009, pp. 121–130.
- [28] J. Aranda and G. Venolia, "The secrete life of bugs: Going past the errors and omissions in software repositories," in *Proc. 31st International Conference on Software Engineering*, May 2009, pp. 298–308.