Impact Analysis of Change Requests on Source Code Based on Interaction and Commit Histories

Motahareh Bahrami Zanjani, George Swartzendruber, Huzefa Kagdi
Department of Electrical Engineering. and Computer Science
Wichita State University
Wichita, Kansas 6760, USA
{mxbahramizanjani, gnswartzendrube, huzefa.kagdi}@wichita.edu

ABSTRACT

The paper presents an approach to perform impact analysis (IA) of an incoming change request on source code. The approach is based on a combination of interaction (e.g., *Mylyn*) and commit (e.g., CVS) histories. The source code entities (i.e., files and methods) that were interacted or changed in the resolution of past change requests (e.g., bug fixes) were used. Information retrieval, machine learning, and lightweight source code analysis techniques were employed to form a corpus from these source code entities. Additionally, the corpus was augmented with the textual descriptions of the previously resolved change requests and their associated commit messages. Given a textual description of a change request, this corpus is queried to obtain a ranked list of relevant source code entities that are most likely change prone. Such an approach that combines information from interactions and commits for IA at the change request level was not previously investigated. Furthermore, the approach requires only the entities that were interacted and/or committed in the past, which differs from the previous solutions that require indexing of a complete snapshot (e.g., a release).

An empirical study on 3272 interactions and 5093 commits from *Mylyn*, an open source task management tool, was conducted. The results show that the combined approach outperforms an individual approach based on commits. Moreover, it also outperformed an approach based on indexing a single, complete snapshot of a software system.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Human Factors, Management, Measurement

Keywords

Impact Analysis, Interaction History, Information Retrieval, Machine Learning

MSR'14, May 31 – June 1, 2014, Hyderabad, India
ACM 978-1-4503-2863-0/14/05
http://dx.doi.org/10.1145/2597073.2597096

1. INTRODUCTION

Software-change impact analysis, or simply impact analysis (IA), is an important activity during the software maintenance and evolution phase. IA aims at estimating the potentially impacted entities of a system due to a proposed change [1]. The applications of IA include cost estimation, resource planning, testing, change propagation, managing ripple effects, and traceability [16, 4, 11, 14, 20, 18].

Change requests are typically specified in natural language (e.g., English). They include bug reports submitted by programmers or end users during the post-delivery maintenance of a product and managed with issue tracking systems (e.g., Bugzilla). One variant of IA that is widely investigated in the literature is to identify the potential source code entities that are change prone on account of a given change request. A number of approaches to address this task have been presented in the literature for performing IA [7]. They range from traditional static and dynamic analysis techniques to modern methods based on Information Retrieval (IR) and Mining Software Repositories (MSR). For example, an IR technique is used to index all the source code entities in a single snapshot and then use the incoming change request to query for the top relevant source code entities. MSR techniques use the historic information from bug and source code repositories to predict the change prone source code. Although ample progress has been made, there remains quite a bit work left in improving the effectiveness of IA.

We propose a new approach, namely *InComIA*, for IA that is centered on the developer interaction and commit histories of source code entities that were involved in the resolution of previous change requests. Developers may interact (e.g., navigate, view, and modify) entities within an Integrated Development Environment (IDE) that may not be eventually committed to the code repository. These interactions could have contributed in locating and/or verifying the entities that were changed due to a change request, which potentially makes them candidates for future changes. Tools such as *Mylyn* capture and store such interaction histories. Our approach combines the source code entities (i.e., files and methods) that were interacted or changed in the resolution of past change requests (e.g., bug fixes). Information retrieval, machine learning, and lightweight source code analysis techniques are then used to form a corpus from these source code entities. Additionally, the corpus is augmented with the textual descriptions of the previously resolved change requests and their associated commit messages. Given a textual description of a change request, this corpus is queried to obtain a ranked list of relevant source

code entities that are most likely change prone (i.e., the estimated impact set). In our previous work [2], we used combinations of interaction and commit histories for source code to source code IA. In this paper, we investigate IA on the onset of an incoming change request to source code.

To evaluate the accuracy of our technique, we conducted an empirical study on the open source system *Mylyn*. Precision and recall metric values on a number of bug reports sampled from this system for IA are presented. That is, how effective our approach is at recommending the actual source code entities (files and methods) that are changed to fix these bugs. Additionally, we empirically compared our approach to those using commits and the entire source code from a single snapsot. The results show that the proposed approach outperformed the baseline competitors: Lowest recall gains of 28% and 44%, highest recall gains of 225% and 350%, lowest precision gains of 28% and 33%, and highest precision gains of 250% and 350% were recorded.

Our paper makes the following noteworthy contributions in the context of performing IA on source code due to an incoming change request:

1. To the best of our knowledge, our approach is the first to integrate the developer interaction and commit histories of source code.
2. We performed a comparative study with an approach that makes an exclusive use of source-code commits.
3. We performed a comparative study with an approach that uses the source code in a single snapshot.

The rest of the paper is organized as follows: Our *InComIA* approach is discussed in Section 2. The empirical study on *Mylyn* open-source projects and the results are presented in Section 3. Threats to validity are listed and analyzed in Section 4. Related work is discussed in Section 5. Finally, our conclusions and future work are stated in Section 6.

2. THE *InComIA* APPROACH

Our *InComIA* approach to IA of an incoming change request on source code consists of the following steps:

- The past change requests (e.g., bug reports) from software repositories are analyzed, and all the source code entities that were interacted and committed are extracted (see Section 2.4).
- The source code of each unique entity, from each revision (i.e., in subversion vocabulary) in which it was interacted or committed in the past, is parsed using a developer-defined granularity level (e.g., file and method). For each entity, comments, identifiers and expressions are then extracted from the parsed source code. For each entity all of its commit messages and bug descriptions are also obtained. A corpus is created such that each source code entity has a corresponding document (i.e., in IR vocabulary) in it.
- Given a change request description, a ranked list of relevant source code entities (e.g., files and methods) is recommended for IA based on the K-Nearest Neighbor algorithm and Cosine similarity.

Before, we present the details of these steps, we briefly discuss the two principal sources of information of *InComIA*.

2.1 Interactions and Commits for Change Request Resolution

Interaction is the activity of programmers in an IDE during a development session (e.g., editing a file, or referencing an API documentation). Different tools (such as *Mylyn*) have been developed to model programmers' actions in IDEs [13, 24, 22, 21]. *Mylyn*[1] monitors programmers' activities inside the Eclipse IDE and uses the data to create an Eclipse user interface focused around a task.

The *Mylyn* interaction consists of traces of interaction histories. Each historical record encapsulates a set of interaction events needed to complete a task. Once a task is defined and activated, the *Mylyn* monitor records all the interaction events (the smallest unit of interaction within an IDE) for the active task. For each interaction, the monitor captures about eight different types of data attributes. The structure handle attribute contains a unique identifier for the target element affected by the interaction. For example, the identifier of a Java class contains the names of the package, the file to which the class belongs to and the class. Similarly, the identifier of a Java method contains the names of the package, the file and the class the method belongs to, the method name and the parameter type(s) of the method. Figure 1 shows an example of 4 consecutive *Mylyn* interaction events. For each active task, *Mylyn* creates an XML trace file called *Mylyn*-context.zip. A trace file contains the interaction history of a task. This file is typically attached to the project's issue tracking system (e.g., Bugzilla or JIRA). The trace files for the *Mylyn* project are archived in the Eclipse bug tracking system as attachments to a bug report[2].

A common practice in the open source software development is for developers to include an explicit bug or issue ID in the commit message. The presence of this information establishes the traceability between an issue or bug reported in the bug tracking system and the specific commit(s) performed to address it. A regular-expression based method can be employed to process commits and extract this traceability information. Figure 2 shows the bug description for bug ID #175229, which is extracted from the Eclipse bug tracking system, and the commit message related to this bug ID, which is for revision #5113. The files that were included in this commit are listed between the *paths* tags. Now, we describe the details of our *InComIA* approach.

2.2 Extracting Interacted Entities

We first need to identify bug reports that contain mylyn-context.zip attachment(s) because all bug issues may not contain interaction trace(s). To do so, we searched the Eclipse bug-tracking system for bugs containing at least one mylyn-context.zip attachment. Another factor to consider is that all of the interactions to a system may not result in committed changes to a source control system. If a bug issue is not fixed with a resolution, it is unlikely for a corresponding commit history to exist. Thus, we only searched for bug issues with a "Resolved" status and a "Fixed" resolution. We developed a tool to process the search result and performs the major following tasks:

`Downloading trace files`: The tool takes the search result from the Eclipse bug-tracking site as input and automatically downloads all the trace files to a user specified direc-

[1]https://www.eclipse.org/mylyn/

[2]https://bugs.eclipse.org/bugs/query.cgi

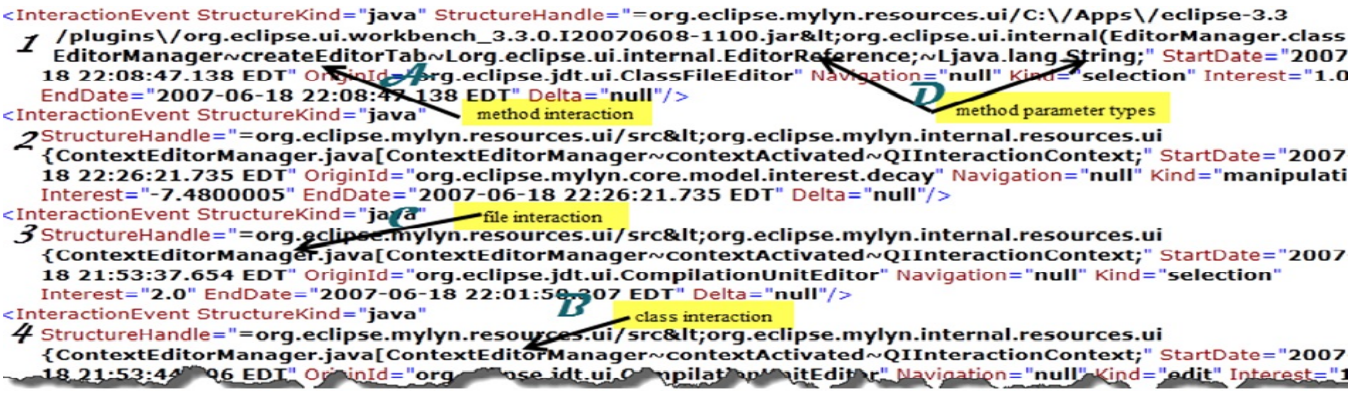

Figure 1: A snippet of 4 interaction events (labelled 1-4) recorded by *Mylyn* for bug issue #175229 with trace ID #71687. In the 1st interaction, the createEditorTab method is selected. In the 2nd, 3rd and 4th interactions, the contextActivated method is indirectly manipulated, then directly selected and finally edited. A) Method name: createEditorTab; B) Class name: ContextEditManager; C) File name: ContextEditorManager.java; D) Parameter types for createEditorTab method: Lorg.eclipse.ui.internal.EditorReference and Ljava.lang.String.

<bug>
<bug_id>175229</bug_id>
<creation_ts>2007-02-23 03:34:00 </creation_ts>
<short_desc>
Should be able to open editor automatically when a task is activated
</short_desc>
</bug>
.....
<logentry revision="5113">
<author>mkersten </author>
<date>2007-06-19T02:27:07 </date>
<paths>
.....
</paths>
<msg>RESOLVED - bug 175229: Should be able to open editor automatically when a task is activated.
https : *//bugs.eclipse.org/bugs/show*$_b$*ug.cgi?id* = 175229
</msg>

Figure 2: Bug ID #175229 from Eclipse bug tracking and commit history.

tory. The trace files have the same name, mylyn-context.zip. The tool renames each file by using the bug ID and attachment ID (separated by an underscore) giving them a unique identifier in the directory they reside in. Internally, the tool identifies the trace file ID(s) for each bug issue. If options are specified to output this result, the tool can save the bug IDs with the corresponding trace IDs in a Java properties file format, the key is the bug id and the values is a comma separated list of trace IDs. It uses the URL pattern https://bugs.eclipse.org/bugs/attachment.cgi?id=X to download each trace file by replacing X with the trace ID.

`Processing trace files`: The tool takes the directory that contains the trace files as input and parses each trace file to identify the list of Java files and methods manipulated by each interaction history. We consider each trace file as an interaction transaction. For each transaction, the tool outputs the issue number together with a tab-separated list of Java files and methods. We need the issue number to create a link between interaction and commit transactions. The targeted files and methods are identified from the structure handle of the interaction event. Figure 1 (A and C) shows a method name and file name from event 1 and 3 respectively. Mylyn can create different types of interaction events (e.g., edit and selection) on the same target in a single interaction history. We consider only the first interaction to an element regardless of the type of interaction event. For further details on this step, please refer to our previous work [2]. We refer to the set of Interacted entities as set $I = \{(i, E_{im})\}$ where i is the revision number and E_{im} is the m^{th} entity E which was interacted in revision i.

2.3 Extracting Committed Entities

Our approach also requires commit data from version archives, such as SVN and CVS, because we need files that have been changed together in a single commit operation. SVN preserves atomicity of commit operations; however, older versions of CVS did not. Subversion assigns a new "revision" number to the entire repository structure after each commit. For a project hosted in an "older" CVS repository, we convert the CVS repository into an SVN repository using the CVS2SVN tool, which has been used in popular projects such as gcc3. Our tool mines file-level commit transactions from the SVN repository. For mining method-level transactions, we used a previously developed tool with some modification to identify the issue number associated with each commit. We refer to the set of committed entities as set $C = \{(i, E_{in})\}$ where i is the revision number and E_{in} is the n^{th} entity E that was committed in the revision i. Figure 3 shows the overall structure of our *InComIA* approach. This section contributes to the left three blocks of Figure 3.

2.4 Obtaining Source Code of Entities from Interacted and Committed Revisions

Our *InComIA* approach needs the source code of all of the entities from all of the revisions in which they were interacted or committed previously in resolving change requests. The extraction step described above only gives the list of entity names and revisions in which they were involved, but not the source code. It is relatively straightforward to determine the source code revision of an entity in which it was committed; however, the precise revision number that was interacted is not available from the interaction history.

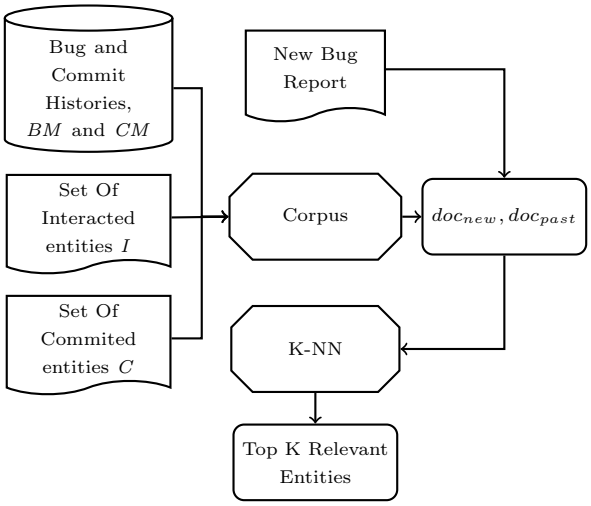

Figure 3: A Schematic diagram of *InComIA*.

Therefore, we consider that the source code in the $n-1^{th}$ revision of an entity should have been interacted if the same entity was committed in the n^{th} revision for an issue.

Source code from each of the revisions in which an entity was interacted is obtained. Similarly, source code from each of the revisions in which an entity was committed is obtained. At the end of this step, we have all the needed source code of each interacted or committed source code entity. We use these source code files to form a corpus.

2.5 Creating a Corpus from Source Code and Textual Descriptions

2.5.1 *Extracting Comments, Identifiers and Expressions*

The source code files from each interacted or committed revision are converted to the srcML representation [3]. This conversion is done for the ease of extraction of identifiers, comments and expressions from the source code. All the comments are extracted from all of the srcML files with an XML query approach. We consider two types of comments in a source code file.

- Header comments that are generally the first comment in a source code file, or leading a source code class and/or source code method, and may contain useful keywords related to the specific role of the related class or method.
- Body comments that sometimes contain important description (e.g., related to a specific change that was done in a specific class or method).

For extracting identifier names, we use the CamelCase splitting technique for compound identifiers. For example, the identifier "TaskHistoryTest" is split into "Task", "History" and "Test" and the identifier "SSLCertificate" is split into "SSL" and "Certificate". Similarly, we consider all of the expressions in a source code file, including all of the output messages, which may contain important keywords.

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

```
public class TaskHistoryTest extends TestCase {
    .....
/**
 * Tests navigation to previous/next tasks that are chosen
 *  from a list rather than being sequentially navigated
 */
public void testArbitraryHistoryNavigation() {

    resetHistory();

    // Simulate activating the tasks by clicking rather
    //than navigating previous/next
    (new TaskActivateAction()).run(task1);
    history.addTask(task1);
```

```
<path
   action="M"
   kind="file">/trunk/.../tasklist/tests/TaskHistoryTest.java</path>
</paths>
<msg>Patch for Bug #110506: provide context for previous/next task
 actions and misc fixes
</msg>
```

Figure 4: A snippet of the file TaskHistoryTest.java. There is a strong textual similarity between the method header and body comments and the commit message.

2.5.2 *Extracting Issue Descriptions*

We add the issue (i.e., bug) descriptions related to each revision that has files that have been interacted or committed to our corpus. Therefore, we need to have a list of all the issue IDs with their textual descriptions. We refer to the set of bug descriptions as set $BM = \{(i, B_{ik})\}$ where i is the revision number and B_{ik} is the k^{th} bug description for revision i. It is possible that for one revision, more than one bug description exists.

2.5.3 *Extracting Commit Messages*

After converting the *Mylyn* CVS repository to an SVN repository, we simply parse the SVN log to find the commit message and bug ID related to each revision number. We refer to the set of commit messages as set $CM = \{(i, C_i)\}$ where i is the revision number and C_i is the commit message for revision i.

After this step, we have a document for each source code entity that was either interacted or committed in the corpus. The dimensions of these documents in the corpus are formed from all of the words extracted from processing the revisions for source code comments and identifiers, issue descriptions and commit messages.

Figure 4 shows that there is a strong text similarity between the header comment of file *TaskHistoryTest.java* "Test navigation to previous/next tasks.... ", the method body comment "....by clicking rather than navigating previous/next" and the commit message from the SVN log "provide context for previous/next task actions ....". After this step, all of the information needed from the left three blocks in Figure 3 is gathered and the initial corpus is formed.

2.6 Indexing and Querying the Corpus

In *InComIA*, we use techniques from Natural Language Processing in order to locate textually relevant files based on comments, identifiers, expressions and bug and commit descriptions. We have the two following documents: a new bug description for which we want to find all of the relevant files (which we will refer to as doc_{new}) and a corpus document made up of all of the information extracted from source code files plus bug descriptions and commit messages (which we will refer to as doc_{past}). Before transforming two

documents to numeric vectors (document representation) we preprocess both of our documents.

2.6.1 Preprocessing Step

The preprocessing step includes two important parts:

`Removing Stop words`: Most common words in English (called stop words) are often removed from documents before any attempt to classify them is made.

`Stemming each word`: Stemming is the process of reducing inflected (or sometimes derived) words to their stem, base or root form. After removing the stop words, we stem each of the words in the bug description.

nltk.stopwords and *nltk.PorterStemmer* from the *Natural Language Toolkit* are used for these two steps[4].

2.6.2 Document Presentation (Including Document Indexing and Term Weighting)

To perform the process of document presentation we use *Gensim*[5] (topic modelling for humans) - a *Python* library.

`Document Indexing`: We produce a dictionary from all of the terms in our document and assign a unique integer ID to each term appearing in it.

`Term Weighting`: We use *tfidf* in our approach. The global importance of a term i is based on its inverse document frequency (idf). idf is the document frequency of term i in the whole document collection. $tf_{i,j}$ is the term frequency of a term i in a document j. Each document d_j is represented as a vector $d_j = (w_{1,j}, ..., w_{i,j}, ..., w_{n,j})$ where n is the total number of terms in our document collection and $w_{i,j}$ is the weight for term i in document j.

2.6.3 Dimensionality Reduction

In *InComIA*, we use the singular value decomposition (SVD) of the feature document matrix representation of our dataset (doc_{past}) [23]. We transform our *tf-idf* corpus into a latent m-D space of a lower dimensionality via the *models.Lsimodel* function in the *Gensim* library. Thus, we have all the processing of corpus creation completed in the "corpus" block in Figure 3.

2.6.4 K-Nearest-Neighbor

The task of multi-label classification is to predict for each data instance, a set of labels that applies to it. Standard classification only assigns one label to each data instance. However, in many settings a data instance can be assigned by more than one label. For IA, each data instance (i.e., a bug report) can be assigned multiple labels (i.e., entities). ML-KNN is a state-of-the-art algorithm in the multi-label classification literature [30]. We employ the K-Nearest-Neighbor (KNN) search with a defined value of K to search the existing corpus (doc_{past}) based on similarities with the new bug description (doc_{new}) (i.e., the "K-NN" block in Figure 3). This search finds the top K similar files or methods. Cosine similarity is used to measure the similarity of the two document vectors (see the three corresponding blocks in Figure 3).

$$f_{sim}\left(doc_{new}, doc_{past}\right) = \frac{doc_{new}.doc_{past}}{|doc_{new}||doc_{past}|} \quad (1)$$

Once a new change request is received, by examining the created corpus, we can recommend the relevant source code entities that need to be fixed to resolve it. In summary, our *InComIA* approach (Formula (2)) is a model based on a union of elements in sets C and I plus the information from sets CM and BM which $Cor_{e_{in}}$ is a created corpora based on entity e_{in}.

$$InComIA = Cor_{e_{in}} \uplus b_i \uplus c_i \quad (2)$$

$(i, e_{i,n}) \in C \cup I, (i, b_i) \in BM, (i, C_i) \in CM$

We use *Mylyn* Interaction data and commits from the SCM for supporting IA. For access to the list of files that are interacted to fix previous bugs, we first extract *Mylyn* interaction files from the bug tracking system and then process them into transactions. Similarly for creating a list of files that are committed to fix previous bugs, we extract commits from the source code repository and then process them into transactions.

Table 1: Predicted files for bug# 201151 by three different approaches

Approach	Predicted files
SIA	—
ComIA	AbstractRepositorySettingsPage.java
InComIA	AbstractRepositorySettingsPage.java BugzillaRepositorySettingsPage.java

2.7 An Example from *Mylyn*

Here, we demonstrate our approach using an example from *Mylyn*. The change request of interest here is the bug# 201151 " *[patch]Bugzilla TaskRepositori Setting let Version Automatic being left when Version not supported*". This bug is fixed in revision# 5595. Our gold set GS, i.e., files changed, to fix this specific bug is *AbstractRepositorySettingsPage.java* and *IBugzillaConstants.java, BugzillaRepositorySettingsPage.java*.

Table 1 shows the results of search, i.e., files predicted by three different approaches. Approaches *SIA* and *ComIA* are discussed in Section 3.1. Only correctly predicted files out of 10 recommended ones are shown. The number of terms for *SIA* is 130607 and the number of documents is 1102. The number of terms for *ComIA* is 5294145 and the number of documents is 2398 and for *InComIA*, the number of terms is 15825476 and the number of documents is 2405. *InComIA* has the largest corpus. *InComIA* has the best performance in comparison with *ComIA* and *SIA*. *ComIA* and *InComIA* have larger corpora than *SIA* because these two approaches consider all the revisions in which files were interacted or committed plus commit messages and bug descriptions.

Tables 2 and 3 show a few interesting commit messages, expressions, and comments for those revisions in which the file *AbstractRepositorySettingsPage.java* was committed and the file *BugzillaRepositorySettingsPage.java* was committed or interacted. This information does not exist in *SIA*, because in *SIA* we only consider a single snapshot of software (in this case the revision 5594 is considered, which is the last). The expressions and comments extracted from source code are shown in tables 2 and 3 do not exist in our gold set source code files in revision 5594.

Our example suggests that comments and expressions from the different revisions of a file plus all of the commit messages and bug descriptions related to those revisions in the corpus could have important words related. If only one revision of software is considered in creating the corpus, we could lose important keywords from other revisions.

[4]http://www.nltk.org

[5]http://radimrehurek.com/gensim/tut3.html

Table 2: Different comments and expressions extracted from file *AbstractRepositorySettingsPage.java* in different revision numbers and associated commit messages

Revision	Comment, expression and commit messages
2526	"Refactor TaskRepository to be extensible "
2872	"add Bugzilla repository templates"
2827	"Trac connector: add repository templates for version support"

Table 3: Different comments and expressions extracted from file *BugzillaRepositorySettingsPage.java* in different revision numbers and associated commit messages

Revision	Comment, expression and commit messages
5189	"Task List Repository preference page allows a negative number of days "
5268	"TaskRepository settings wizards don't persist individual proxy settings"

3. EMPIRICAL EVALUATION

The main purpose of this case study was to investigate how well our combined approach *InComIA* for IA performed in predicting relevant entities for fixing a new incoming change request (e.g., bug). We compared *InComIA* with a previous approach for IA that indexes all the source code in a single snapshot (denoted here as *SIA*). Moreover, we compared *InComIA* with an approach that uses past commits and bug descriptions (denoted here as *ComIA*). This comparison would permit the assessment of the impact of including interactions in *InComIA*.

3.1 Research Questions

We addressed the following research questions (RQs) in our case study:

- **RQ1.** How do *ComIA* (an IA approach trained from entities in only set C) and the *InComIA* (an IA approach trained by entities in combined set C and I) compare in predicting the top K relevant entities for incoming change requests?
- **RQ2.** How do IA approaches *ComIA* and *InComIA* compare with the previous model *SIA* in predicting the top K relevant entities for incoming change requests?

ComIA is a model based on committed entities in the set C plus the information from sets CM and BM. $Cor_{e_{in}}$ is a created corpora based on entity e_{in} $(i, e_{i,n}) \in C, (i, b_i) \in BM, (i, C_i) \in CM$.

$$ComIA = Cor_{e_{in}} \uplus b_i \uplus c_i \quad (3)$$

SIA is a common model that is based on indexing the source code from a single snapshot (e.g., software release).

The purpose of **RQ1** was to assess if incorporating an additional set of interacted entities to our training set improved the accuracy of IA or not. The accuracy comparison between the approaches *InComIA* and *ComIA* would help us assess the level of the past interacted entities' contribution to the accuracy of *InComIA*. That is, *is it really worthwhile to put in the additional work of extracting entities interacted with in the resolution of past change requests, or would the committed entities would suffice?* The purpose of **RQ2** was to assess how do IA approaches based only source code entities that were interacted and/or committed in the past, compare with an approach that uses entire source code body in a single snapshot.

3.2 Experiment Setup

For *ComIA*, we considered all of the bugs that have the associated ID in a commit message. Subversion (SVN) repository commit logs were used to aid in this process. For example, keywords (such as the bug ID) in the commit messages/logs were used as starting points to determine if the commits were in fact associated with the mentioned change request in the issue tracking system. Similarly, we gained the associated revision numbers for each bug ID.

For *InComIA*, we considered all of the bugs that have at least one associated ID either in the commit messages or in the interacted trace files. A list of files (or methods) with the associated revision number was then extracted. The specific versions of the files (or methods) that were committed (in *ComIA*) and either committed or interacted (in *InComIA*) for each bug in the testing set were excluded from the training set. All of the necessary information (comments, identifiers and expressions) was extracted from each revision of the entity plus all of the commit messages and bug descriptions for each of the revisions in order to create the final training corpus. We used a dimensionality of 500, which is recommended for a large number of terms [3]. The experiment was run for two K values: K@10 and K@20.

3.3 Subject Software System

We focused our evaluation on the *Mylyn* project, which contains about 4 years of interaction data. It is an Eclipse Foundation project with the largest number of interaction history attachments. It is mandatory for *Mylyn* project committers to use the *Mylyn* plug-in. Commit history started 2 years prior to that of interaction, and commits to the *Mylyn* CVS repository terminated on July 01, 2011. To get both interaction and commit histories within the same period, we considered the history between June 18, 2007 (the first day of interaction history attachment) and July 01, 2011 (the last day of a commit to the *Mylyn* CVS repository).

3.4 Dataset

The *Mylyn* project consists of 2275 bug issues containing 3272 trace files. After preprocessing the traces and filtering out noises, 2357 file level and 2174 method level transactions were identified. Table 4 provides information about the file and method levels of interaction transactions for the *Mylyn* project. There were more file level interaction transactions than method level interaction transactions. This difference may be due to the fact that *Mylyn* propagates lower-level interaction events into their parents. The *Mylyn* project contained 5093 revision histories. Out of 5093 change sets, 3727 revisions contained a change to at least one Java file and 2058 revisions contained a change to at least one Java method. About 3572 (96%) of file level changes and 1947 (95%) of method level changes were associated with issues.

Table 4: *Mylyn* project interaction and commit histories from June 18, 2007 to July 01,2011.

System	Interaction		Commit	
Mylyn	3272 traces		5093 revisions	
	File	Method	File	Method
	2357	2174	3727	2058

3.5 Training and Testing Sets

Both interaction and commit datasets were split into two groups: training and testing sets. The training sets were used to train our machine learning technique, and the testing sets were used to measure the effectiveness of three different models for impact analysis. We used the first 90% of the bugs for the training set and the next 10% of the bugs for the testing set. We had three different models and two levels of granularity. Therefore, we had a total of 6 training sets.

3.6 Performance Metrics

To evaluate the accuracy of the three models, we used two popular measures: precision and recall. Suppose that there are m bug reports. For each bug report b_i let the set of actual entities that were changed be F_i. We recommend the top K entities E_i for b_i with each of the approaches. Recall @K and precision @K for the m bug reports are given by:

$$Recall@K = \frac{1}{m}\sum_{i=1}^{m}\frac{\mid E_i \cap F_i \mid}{\mid F_i \mid} \tag{4}$$

$$Precision@K = \frac{1}{m}\sum_{i=1}^{m}\frac{\mid E_i \cap F_i \mid}{\mid E_i \mid} \tag{5}$$

These metrics were computed for a recommendation list of entities with different sizes (i.e, K=10, K=20).

3.7 Hypotheses Testing

We derived testable hypotheses to evaluate our research questions. We only list the null hypotheses because one can easily derive the alternative hypotheses from them. SSD indicates statistically significant difference.

H_{01}: There is no SSD between the precision/recall values of *InComIA* and *ComIA* for the file-level.

H_{02}: There is no SSD between the precision/recall values of *InComIA* and *ComIA* for the method-level.

H_{03}: There is no SSD between the precision/recall values of *ComIA* and *SIA* for the file-level.

H_{04}: There is no SSD between the precision/recall values of *ComIA* and *SIA* for the method-level

H_{05}: There is no SSD between the precision/recall values of *InComIA* and *SIA* for the file-level.

H_{06}: There is no SSD between the precision/recall values of *InComIA* and *SIA* for the method-level.

Table 5: A heat-map summarizing hypotheses test results across all the three models for two different values of k. Significant difference: Cells colored black indicate that it exists for both method and file levels; for dark-gray it exists for only the file level; for light-gray it exists for only the method level; for white there is none for both levels.

H	K@10		K@20	
	Precision	Recall	Precision	Recall
H_{01} H_{02}				
H_{03} H_{04}				
H_{05} H_{06}				

We performed the analysis of variance (parametric ANOVA) test with $\alpha = 0.05$ to validate whether there is a statistically significant difference between the models. Table 5 is a heat-map summarizing the hypotheses test results across all three different models for two values of K.

3.8 Case Study Results

In this section, we compare the results of three different models: *ComIA*, *InComIA* and *SIA*. For each change request in the benchmark, we parametrized three different models to predict the top ten and top twenty relevant entities. These recommendations were compared with the actual entities that were changed to fix the considered change request in order to compute the precision and recall values. Table 6 compares recall@10 and recall@20 at both the file and method levels of granularity. As expected, the recall value generally increases with the increase in the K value for each model, and the precision value decreases with the increase in the K value for each model. For example, for the *InComIA* model at the file level granularity, the recall@10 and recall@20 values are 0.26 and 0.32, and the precision@10 and precision@20 values are 0.15 and 0.12.

To answer the research question RQ1, we compare recall and precision values of the *InComIA* and *ComIA* models for different values of K. We computed the precision gain of *InComIA* over *ComIA*, which is computed using the formula:

$$GainP@k_{InComIA-ComIA} = \frac{prec@k_{InComIA} - prec@k_{ComIA}}{prec@k_{ComIA}} \times 100 \tag{6}$$

The two **GainP *InComIA-ComIA*** columns in Table 6 show the precision gain of *InComIA* over *ComIA* for different K values at both file and method levels. As can been seen, *InComIA* clearly outperforms *ComIA* at both the file and method levels for all K values. The precision gain at the file level ranges from 33.33% to 36% and for method level ranges from 28.57% to 75%. Similarly, we computed the recall gain of *InComIA* over *ComIA*, which is computed using the formula:

$$GainR@k_{InComIA-ComIA} = \frac{rec@k_{InComIA} - rec@k_{ComIA}}{rec@k_{ComIA}} \times 100 \tag{7}$$

The two **GainR *InComIA-ComIA*** columns in Table 6 show the recall gain of *InComIA* over *ComIA* for the different K values at both file and method levels. As can been seen, *InComIA* clearly outperforms *ComIA* at both the file and method levels for all K values. The recall gain at the file level ranges from 28% to 44% and for the method level ranges from 58.82% to 80%. In summary, the overall results suggest that *InComIA* would generally perform better than *ComIA* in terms of recall and precision.

To answer the research question RQ2 we compare the recall and precision values of *InComIA* and *ComIA* with those of *SIA* for different values of K. We computed the precision gain of *InComIA* over *SIA* and the precision gain of *ComIA* over *SIA*, which is computed similarly with formula (6). The precision gain at the file level for *ComIA* over *SIA* ranges from 120% to 125% and for the method level it ranges from 100% to 250%. Therefore *ComIA* outperforms *SIA* at both the file and method levels for all different values of K. The precision gain at the file level for *InComIA* over *SIA* is 200% and for the method level it ranges from 250% to 350%. Therefore, *InComIA* outperforms *SIA* at both the file and method levels in terms of precision.

The recall gain of *InComIA* and *ComIA* over *SIA* is computed similarly with formula (7). From Table 6, the two columns labelled GainR *ComIA-SIA* show that at both the

Table 6: Recall@10 and 20 and Precision@10 and 20 of three models *InComIA*, *ComIA*, and *SIA*

	Recall@10					
Granularity	*InComIA*	*ComIA*	*SIA*	GainR *InComIA-ComIA*	GainR *ComIA-SIA*	GainR *InComIA- SIA*
File	0.26	0.18	0.08	44.00%	125.00%	225.00%
Method	0.18	0.1	0.04	80.00%	150.00%	350.00%

	Precision@10					
Granularity	*InComIA*	*ComIA*	*SIA*	GainP *InComIA-ComIA*	GainP *ComIA-SIA*	GainP *InComIA- SIA*
File	0.15	0.11	0.05	36.00%	120.00%	200.00%
Method	0.09	0.07	0.02	28.57%	250.00%	350.00%

	Recall@20					
Granularity	*InComIA*	*ComIA*	*SIA*	GainR *InComIA-ComIA*	GainR *ComIA-SIA*	GainR *InComIA-SIA*
File	0.32	0.25	0.12	28.00%	108.33%	166.66 %
Method	0.27	0.17	0.06	58.82%	183.00%	350.00 %

	Precision@20					
Granularity	*InComIA*	*ComIA*	*SIA*	GainP *InComIA-ComIA*	GainP *ComIA-SIA*	GainP *InComIA-SIA*
File	0.12	0.09	0.04	33.33%	125.00%	200.00%
Method	0.07	0.04	0.02	75.00 %	100.00 %	250.00 %

file and method levels *ComIA* outperforms *SIA* for all of the different values of K. The recall gain value for *ComIA* over *SIA* ranges from 108.33% to 125% at the file level and ranges from 150% to 183% at the method level. The recall gain value for *InComIA* over *SIA* ranges from 166.66% to 225% at the file level and is 350% at the method level. Therefore, *InComIA* outperforms *SIA* at both the file and method levels for recall. In summary, the overall results suggest that both *InComIA* and *ComIA* would generally perform better than *SIA* in terms of recall and precision.

To test our hypotheses, we applied the One Way ANOVA test on the recall and precision values of all three approaches with different values of K at both the file and method levels. From Table 5, we can see that there is a statistically significant difference between the precision and recall values of *InComIA* and *ComIA* at the file level for two different values of K, so we reject H_{01}. However, for the precision value of K@10 at the method level there is no statistically significant difference, so we accept H_{02}. For *ComIA* and *SIA* there is a statistically significant difference for the values of precision and recall at the file level. However, for the precision value of K@20 at the method level there is no statistically significant difference, Therefore we reject H_{03} and we accept H_{04}. Results of the ANOVA test show a statistically significant difference between the precision and recall values of *InComIA* and *SIA* at both the file and method levels, so we reject both H_{05} and H_{06}. In summary, the overall results from Table 5 suggest that *InComIA* would generally perform better than *ComIA* and *SIA* in terms of recall and precision.

4. THREATS TO VALIDITY

We discuss internal, construct, and external threats to validity of the results of our empirical study.

Incomplete or Missing Interaction History: Although, a common period was considered for extracting the interaction and commit datasets in the Mylyn dataset, the number of commit transactions is significantly higher than the number of interaction transactions. This difference may not be the result of a single task getting defined for multiple commits because there are many cases in which committed files were never part of one of the corresponding interactions.

CVS to SVN Conversion: We do not know the error rate of CVS2SVN when grouping individual CVS files. It may erroneously split a commit into multiple, or group multiple commits into one. There are 1366 more commits than the number of interaction traces for the same period. This difference could be due to errors from CVS2SVN.

Explicit Bug ID Linkage: We considered interactions and commits to be related if there was an explicit bug id mentioned in them. Implicit relationships were not considered.

Training and Testing Set Split: We considered only a 90%:10% split between training and testing sets. It is possible that a different split point could produce different results.

Single Period of History: We considered only the history between June 18, 2007 and July 01, 2011. It is possible that this history is not reflective of the optimum results for all the models. A different history period might produce different results in terms of their relative performance.

Accuracy and Practicality: The accuracies of the two standalone techniques, however low in certain cases to raise a practicality concern, are comparable to other previous results (Zimmermann et al. 2005). Our work shows how to improve accuracy by forming effective combinations.

Only One System Considered: Due to the lack of adequate Mylyn interaction histories for open source projects, our validation study was performed only on a single system written in Java. It was the one with the largest available dataset within Eclipse Foundation. It had over 2600 fixed bug reports that contained at least one interaction trace attachment. The second and third largest projects (the Eclipse Platform and Modeling) had about 700 and 450 such bugs. Nonetheless, this fact may limit the generality of our results.

5. RELATED WORK

We briefly discuss the related work in impact analysis. Our intent is not to exhaustively discuss every single work, but to briefly discuss a few representatives.

5.1 Interaction History

Researchers have been developing IDE plug-ins to capture programmers' interactions during programming activities [6, 24, 21]. NavTracks was used to mine Interaction coupling at the file-level granularity [24]. Team Track [6] was used to provide navigation support to programmers un-

familiar with the code base. In HeatMaps [21], the interestingness of a programming element is determined by computing a Degree-of-Interest (DOI) value based on the historical selection and modification of it. If an artifact is found interesting, it is decorated with colors to indicate its importance to the task. Zou et al. [31] used the interaction history to identify evolutionary information about a development process (such as restructuring is more costly than other maintenance activities). Rastkar et al. [17] report on an investigation which considered how bug reports considered similar based on changeset information compare to bug reports considered similar based on interaction information. Robbes et al. [19] developed an incremental change based repository by retrieving the program information from an IDE (which includes more information about the evolution of a system than traditional SCM) to identify a refactoring event. Parnin and Gorg [15] identified relevant methods for the current task by using programmers' interactions with an IDE. Kobayashi et al. [10] presented a Change Guide Graph (CGC) based on interaction information to guide programmers to the location of the next change. Each node in the graph presents a changed artifact and each edge presents a relation between consecutive changes. Following the CGC graph, the next target in the change sequence can be identified. Schneider et al. [22] presented a visual tool for mining local interaction histories to help address some of the awareness problems experienced in distributed software development projects. Both interaction history and static dependencies were used to provide a set of potentially interesting elements to a programming task.

5.2 Software Change IA via IR

Information Retrieval (IR) methods were proposed and used successfully to address tasks of extracting and analyzing textual information in software artifacts, including change impact analysis in source code [5, 8, 16]. Existing approaches to IA using IR operate at two levels of abstraction: change request [5] and source code [8, 16]. In the first case, the technique relies on mining and indexing the history of change requests (e.g., bug reports). In particular, this IA method utilizes IR to link an incoming change request description to similar past change requests and the file revisions that were modified to address them [5, 27]. While this technique has been shown to be relatively robust in certain settings, it is entirely dependent on the history of prior change requests. In cases where textually similar change requests cannot be identified (or simply do not exist), the technique may not be able to identify relevant impact sets. Also, these works show that a sizeable change request history must exist to make this approach operational in practice, which may limit the effectiveness. The other set of techniques to IA that use IR operates at the source code level and requires a starting point (e.g., a source code method that is likely to be modified in response to an incoming change request) [8, 16]. This approach is based on the hypothesis that modules (or classes) in software systems are related in multiple ways. The evident and most explored set of relationships is based on data and control dependencies; however, the classes can be also related conceptually (or textually), as they may contribute to the implementation of similar domain concepts. In our previous work [8, 7], given a textual change request (e.g., a bug report), a single snapshot (i.e., release) of source code, indexed using Latent semantic indexing, is used to estimate the impact set and to derive conceptual couplings from the source code. Machine learning techniques and information retrieval are used in different areas such as bug triaging, impact analysis, feature location, bug severity and classifying software changes [25, 9, 26]. Different machine learning techniques such as ML-KNN, SVM, LSI or decision tree are also used for recommending developers to resolve a reported bug request [28, 29, 12].

6. CONCLUSIONS AND FUTURE WORK

We presented an approach based on a combination of interaction (e.g., *Mylyn*) and commit (e.g., CVS) histories to perform impact analysis (IA) of an incoming change request on source code. The approach requires only the entities that were interacted and/or committed in the past, which differs from the previous solutions that require indexing of a complete snapshot (e.g., a release) or past commits alone. We conducted an empirical study on the open source system *Mylyn*. We empirically compared our approach to those using commits and the entire source code from a single snapsot. The results show that the proposed approach outperformed the baseline competitors: Lowest recall gains of 28% and 44%, highest recall gains of 225% and 350%, lowest precision gains of 28% and 33%, and highest precision gains of 250% and 350% were recorded. In the future, we plan to include additional information in our combined approach, including the use of static and dynamic information.

7. ACKNOWLEDGMENTS

This work is supported in part by the NSF CCF-1156401 grant. Any opinions, findings and conclusions expressed herein are those of the authors and do not necessarily reflect those of the sponsors

8. REFERENCES

[1] R. S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[2] F. Bantelay, M. Zanjani, and H. Kagdi. Comparing and combining evolutionary couplings from interactions and commits. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*, pages 311–320, Oct 2013.

[3] R. B. Bradford. An empirical study of required dimensionality for large-scale latent semantic indexing applications. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management*, CIKM '08, pages 153–162, New York, NY, USA, 2008. ACM.

[4] L. Briand, J. Wust, and H. Lounis. Using coupling measurement for impact analysis in object-oriented systems. In *Proceedings of the IEEE International Conference on Software Maintenance, 1999. (ICSM '99)*, pages 475–482, 1999.

[5] G. Canfora and L. Cerulo. Fine grained indexing of software repositories to support impact analysis. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 105–111, New York, NY, USA, 2006. ACM.

[6] R. DeLine, M. Czerwinski, and G. Robertson. Easing program comprehension by sharing navigation data. In

Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, pages 241–248, Sept 2005.

[7] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk. Integrated impact analysis for managing software changes. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 430–440, Piscataway, NJ, USA, 2012. IEEE Press.

[8] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. Collard. Blending conceptual and evolutionary couplings to support change impact analysis in source code. In *Proceedings of the 17th Working Conference on Reverse Engineering (WCRE)*, pages 119–128, Oct 2010.

[9] S. Kim, E. Whitehead, and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, March 2008.

[10] T. Kobayashi, N. Kato, and K. Agusa. Interaction histories mining for software change guide. In *Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering (RSSE)*, pages 73–77, June 2012.

[11] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the 25th International Conference on Software Engineering*, pages 308–318, May 2003.

[12] M. Linares-Vasquez, K. Hossen, H. Dang, H. Kagdi, M. Gethers, and D. Poshyvanyk. Triaging incoming change requests: Bug or commit history, or code authorship? In *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 451–460, Sept 2012.

[13] G. Murphy, M. Kersten, and L. Findlater. How are java software developers using the elipse ide? *IEEE Transactions on Software Engineering*, 23(4):76–83, July 2006.

[14] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M. J. Harrold. An empirical comparison of dynamic impact analysis algorithms. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 491–500, Washington, DC, USA, 2004. IEEE Computer Society.

[15] C. Parnin and C. Gorg. Building usage contexts during program comprehension. In *Proceedings of the 14th IEEE International Conference on Program Comprehension, ICPC*, pages 13–22, 2006.

[16] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy. Using information retrieval based coupling measures for impact analysis. *Empirical Softw. Engg.*, 14(1):5–32, Feb. 2009.

[17] S. Rastkar and G. Murphy. On what basis to recommend: Changesets or interactions? In *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories, MSR '09*, pages 155–158, May 2009.

[18] X. Ren, B. Ryder, M. Stoerzer, and F. Tip. Chianti: a change impact analysis tool for java programs. In *Proceedings of the 27th International Conference on Software Engineering,ICSE*, pages 664–665, May 2005.

[19] R. Robbes. Mining a change-based software repository. In *Proceedings of the Fourth International Workshop on Mining Software Repositories, 2007. ICSE Workshops MSR '07.*, pages 15–15, May 2007.

[20] M. P. Robillard. Automatic generation of suggestions for program investigation. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 11–20, New York, NY, USA, 2005. ACM.

[21] D. Rothlisberger, O. Nierstrasz, S. Ducasse, D. Pollet, and R. Robbes. Supporting task-oriented navigation in ides with configurable heatmaps. In *Proceedings of the IEEE 17th International Conference on Program Comprehension,ICPC '09*, pages 253–257, May 2009.

[22] K. Schneider, C. Gutwin, R. Penner, and D. Paquette. Mining a software developer's local interaction history. In *Proceedings of the IEEE International Conference on Software Engineering Workshop on Mining Software Repositories,*, 2004.

[23] M. Shafiei, S. Wang, R. Zhang, E. Milios, B. Tang, J. Tougas, and R. Spiteri. Document representation and dimension reduction for text clustering. In *Proceedings of the IEEE 23rd International Conference on Data Engineering Workshop*, pages 770–779, 2007.

[24] J. Singer, R. Elves, and M. Storey. Navtracks: supporting navigation in software maintenance. In *Proceedings of the 21st IEEE International Conference on Software Maintenance, ICSM'05*, pages 325–334, Sept 2005.

[25] F. Thung, S. Wang, D. Lo, and L. Jiang. An empirical study of bugs in machine learning systems. In *Proceedings of the IEEE 23rd International Symposium on Software Reliability Engineering (ISSRE)*, pages 271–280, Nov 2012.

[26] Y. Tian, D. Lo, and C. Sun. Information retrieval based nearest neighbor classification for fine-grained bug severity prediction. In *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE)*, pages 215–224, Oct 2012.

[27] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 1–, Washington, DC, USA, 2007. IEEE Computer Society.

[28] W. Wu, W. Zhang, Y. Yang, and Q. Wang. Drex: Developer recommendation with k-nearest-neighbor search and expertise ranking. In *Proceedings of the 18th Asia Pacific Software Engineering Conference (APSEC)*, pages 389–396, Dec 2011.

[29] X. Xia, D. Lo, X. Wang, and B. Zhou. Accurate developer recommendation for bug resolution. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*, pages 72–81, Oct 2013.

[30] M.-L. Zhang and Z.-H. Zhou. Ml-knn: A lazy learning approach to multi-label learning. *Pattern Recogn.*, 40(7):2038–2048, July 2007.

[31] L. Zou, M. Godfrey, and A. Hassan. Detecting interaction coupling from task interaction histories. In *Proceedings of the 15th IEEE International Conference on Program Comprehension,ICPC '07*, pages 135–144, June 2007.