Understanding Widespread Changes: A Taxonomic Study

Shaowei Wang, David Lo, and Lingxiao Jiang
School of Information Systems
Singapore Management University, Singapore
{shaoweiwang.2010,davidlo,lxjiang}@smu.edu.sg

***Abstract*—Many active research studies in software engineering, such as detection of recurring bug fixes, detection of copy-and-paste bugs, and automated program transformation tools, are motivated by the assumption that many code changes (e.g., changing an identifier name) in software systems are widespread to many locations and are similar to one another. However, there is no study so far that actually analyzes widespread changes in software systems. Understanding the nature of widespread changes could empirically support the assumption, which provides insight to improve the research studies and related tools.**

Our study in this paper addresses such a need. We propose a semi-automated approach that recovers code changes involving widespread changes in software systems. We further manually analyze more than nine hundred widespread changes recovered from eight software systems and categorize them into 11 families. These widespread changes and their associated families help us understand better why these widespread changes are made.

I. Introduction

During software evolution, many similar changes can be made to various locations within a software system together. For example, refactoring code that changes an identifier name may need to change many places in the code that use the name. Also, similar bugs may happen at different locations in a program and fixing them requires changes to all of these locations. We refer to such a phenomenon that similar code change may appear in many locations in a program as *widespread changes*. Many past studies have leveraged widespread changes for various purposes. Work on recurring bug fixes [36], [44] leverages the fact that bug-fixing changes may repeatedly occur in different locations at different time to automatically detect possible bugs and suggest fixes. Studies on copy-paste bug detection [28], [39] investigate bugs occurred in similar pieces of code based on the assumption that similar code should evolve in a similar way and inconsistencies among similar code would be an indication of bugs. Many other studies and tools are motivated by this phenomenon as well, such as studies on code refactoring [14], automated code search [41], [55], [59], and program transformation and synthesis [50], and they often aim to detect pieces of code that need to be created or changed in a similar way. Despite the proliferation of studies that rely on such a phenomenon and its impact to existing research, few studies have focused on the understanding of widespread changes themselves.

In this paper, we aim to fill the gap and provide better understanding of the nature of widespread changes by answering questions like how often widespread changes occur in a program and what types of widespread changes there are. We propose an approach that can semi-automatically detect widespread changes within the same program versions, and perform an empirical study on more than nine hundred detected widespread changes to understand possible reasons that cause widespread changes to occur.

To detect these widespread changes, we perform a multi-step semi-automated approach involving: data collection and filtering, clone mining, and manual identification. In the data collection step, we extract thousands of revisions from various software systems. For each revision, we perform a *diff* operation with the immediate revision prior to it to find the lines of code that are changed. In the filtering step, we remove code changes that contain too many or too few lines of code. In the clone mining step, we rely on code clone detection to recover *syntactically* similar changes in each revision that occur a sufficient number of times and spread across a number of files. In the manual identification step, we inspect the detected syntactically similar changes and flag those that are widespread ones by using the following criterion: the changes should be *semantically* similar or related; in other words, they must be made to achieve a similar or related goal, for example, fixing the same kind of bugs at multiple locations.

To understand the widespread changes, we perform additional analysis to categorize them into various families. Based on possible reasons that cause the changes to occur, we group them into 10 families, including *argument addition/removal*, *argument change*, *method addition*, *method change*, *data structure*, *external change*, *non-functional*, *feature addition*, *algorithm change*, and *bug fix*. For completeness, we also include another family *others* that captures everything else.

We analyze more than 32,452 revisions from eight software systems, including ArgoUML [1], Columba [2], Lightweight Java Game Library [5], JFreeChart [4], MegaMek [6], PDF Split and Merge [7], TightVNC [8], and JEdit [3]. The systems are small to large open source software ranging from 30,427 lines of code (LOC) to 382,740 LOC. Out of these 32,452 revisions, we detect more than 965 widespread changes and categorize them into families. We find that *non-functional* widespread changes are the most common followed by *feature addition*, *algorithm change*, and *bug fix*.

Our contributions are as follows:

1) We highlight the importance of understanding widespread changes and its impact to many software

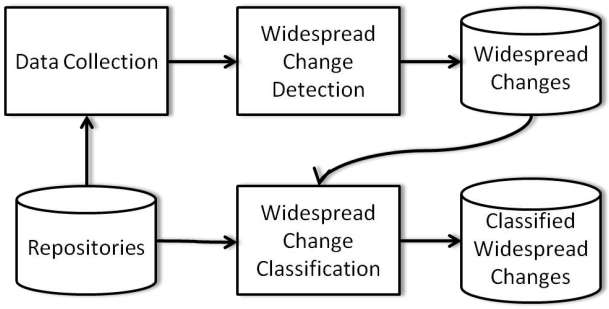


Fig. 1. Overview of Our Approach

engineering studies.

2) We propose a semi-automated approach to recover many widespread changes from software systems.
3) We manually investigate 965 detected widespread changes from eight software systems and classify them into 11 categories. The categorization gives insights on the reasons that cause these widespread changes to occur.

This paper is structured as follows. We discuss our overall methodology in Section II and zoom into major steps of our methodology in Sections III, IV, and V. We present the results of our empirical study in Section VI. We discuss related work in Section VII. Section VIII concludes with future work.

II. Approach Overview

In order to gain insights into widespread changes, we propose an approach to collect many widespread changes from software repositories. Based on the intuitive definition of widespread changes, this section introduces our approach for collecting raw data about code change histories, detecting widespread changes, and classifying the changes.

Our rationale for the semi-automated approach is based on the intuitive definition of widespread changes: *widespread changes are the code changes that are (syntactically and semantically) similar to one another and occur at different locations in a program.* As the software version control systems often record all of the code changes made to a program, it is thus possible to recover widespread changes from the software repositories by searching for similar code changes in the code change history.

The major steps in our approach are shown in Figure 1. The process first accesses the repository of a given program, and extracts all of the commits made to the repository. Then, a semi-automated step based on code clone detection is used to search for similar changes among the commits. Heuristics are also applied to filter out changes that are unlikely to be widespread changes. After the semi-automated step on detection of widespread changes, we carry out further manual investigation of the detected widespread changes and classify them into various categories to facilitate understanding.

The following Sections III, IV, and V describe each of these steps in more details.

III. Collection of Raw Data

The major task for this step is to retrieve the code change histories from the revision control systems (e.g., git[1] and subversion[2]) of desired programs. Since we mainly look for *changes*, it is sufficient for our approach to collect just the *diff*[3] between every two revisions that may contain the desired changes.

In this study, we look at the changes between every *consecutive* revisions. This ensures that the overall process includes fine-grained changes. In cases when changes at a larger granularity are desired (e.g., the changes between major releases of a program which may contain many changes from many revisions), our approach may be adjusted to take the diff between the major releases as input and possibly detect less but larger widespread changes.

The collection of change history from a repository is often straightforward by using the commands available from the revision control systems and a set of scripts to automate the collection process. For example, for subversion repositories, we use `svn checkout svn://somepath@r working-directory` to retrieve a particular revision, and `svn diff -r r1:r2` to get the diff between two revisions. For each diff, it is often comprised of a set of *change hunks*[4] each of which represents a region that is changed in a file in a project as a set of added lines and a set of deleted lines. Each change hunk is often shown together with some number (three by default) of unchanged lines above and below the added and deleted lines; these unchanged lines are also often referred as the surrounding contexts of the change hunk. Each change hunk, together with its surrounding contexts, is called a *hunk*.

While the general term *change* refers to changes at any granularity (e.g., byte changes, token changes, etc.), hunks usually use *lines* as the granularity for representing changes. In this paper, we view hunks as sequences of token-level changes. One hunk may contain the same changes as the other, in which case we say the hunk is the same as the other; or only a subset of all changes contained in the hunk is the same as another subset of all changes contained in another hunk, in which case we say the hunk is partially the same as or similar to the other hunk. Such similarity measurements among hunks are the basis for our detection of *widespread changes* in program revisions (cf. Section IV).

The actual programs and their repositories that we have collected for our study in this paper are detailed in Section VI-A.

IV. Detection of Widespread Changes

Based on the intuitive definition of widespread changes (cf. Section II), we propose four criteria for a set of hunks to be considered as a widespread change:

(1)

[1]http://git-scm.com/
[2]http://subversion.apache.org/
[3]GNU Diffutils: http://www.gnu.org/software/diffutils/
[4]http://en.wikipedia.org/wiki/Diff, 16^{th} October, 2012

1) Each hunk should be of a relatively small size considering that many common real-world widespread changes (e.g., changing an identifer name, changing a method interface) are small.
2) The code involved in the hunks should be *syntactically similar* to one another; i.e., the hunks are meant to contain similar changes made in a similar way.
3) The code involved in the hunks should spread over multiple locations and files; i.e., the changes made in the hunks should repeatedly appear in various places.
4) The code involved in the hunks should be *semantically similar* to one another; i.e., the hunks are meant to address the same or similar issues in a program.

The criteria enable us to design a semi-automated technique to detect many widespread changes as follows so as to scale up our study. The flowchart describing the detection process is shown in Figure 2. Given the diff files containing the difference between every two consecutive revisions (cf. Section III), our detection component would eventually output the widespread changes, if any, existing in the diff files. During the process, (i) our technique first extracts the hunks, including the changed code *and* the code surrounding the change hunk (i.e., the code three lines before or after every line of changed code), from each diff; (ii) then the extracted code is fed to a code clone detection tool to find syntactically similar code fragments, which could be considered as candidates for widespread changes; (iii) and filter out changes that do not occur in a sufficient number of places; (iv) finally we manually examine the filtered outputs from the clone detection tool and remove changes that are semantically irrelevant.

The step (i) takes the criterion (1) into consideration and ignores hunks that contain more than $maxHunkSize$ lines of code. In this paper, by default we set the parameters $maxHunkSize$ to be 20 lines. Since as our observation, very rare widely changes appear in the hunk which is more than 20 lines.

The step (ii) takes the criterion (2) into consideration and only keeps code hunks that are syntactically similar to others (i.e., the hunks contain similar changes to others). This step involves measuring similarity among code fragments and can be naturally achieved by a code clone detection tool.

Code clones (i.e., pieces of code with similar appearance) are believed to exist ubiquitously in software systems. Many code clone detection techniques and tools have been developed [9], [10], [15], [25], [29], [40]. A code clone detection tool in general works as follows: it converts a given set of pieces of code into certain intermediate representations (e.g., a sequence of tokens, a sequence of hash values, a set of feature vectors capturing the essential code characteristics, etc.), looks for similar intermediate representations with various techniques, and thus find similar code. Based on the intermediate representations used, the tools can be classified into string-based [9], token-based [29], tree-based [10], [25], graph-based [15], etc. Tree-based or graph-based tools often require complete and compilable code, while the code considered in our setting is the hunks from *diffs* that may not be syntactically complete. Token-based approaches are often more robust than string-based approaches against non-essential code differences such as formatting and spacing changes. Thus, in our study, we choose to use a token-based clone detection tool, CCFinder [29], to detect clones (i.e., syntactically similar code changes in our setting) among hunks. In particular, when using CCFinder, we set the minimum length of the detected code clones to be 10 tokens, and set the minimum number of kinds of tokens in input code fragments to be 6. At the end of the step (ii), CCFinder reports groups of syntactically similar code fragments (a.k.a. *clone groups*) from different hunks. These groups of code fragments are thus the candidates for widespread changes. Since CCFinder may report a partial hunk in a clone group, we can see that a widespread change detected by our approach is a set of hunks or partial hunks.

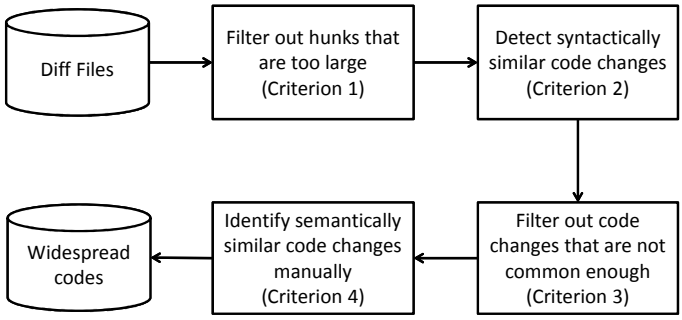


Fig. 2. Detection of Widespread Changes.

In the step (iii), we take the criterion (3) into consideration: a widespread change should occur in more than $minLoc$ locations across $minFile$ files. Thus, if a group of syntactically similar code changes reported by CCFinder (i.e., a clone group) involves code changes from less than $minLoc$ hunks or less than $minFile$ files, we also remove it. In this paper, by default we set the parameters $minLoc$ and $minFile$ to be 5 and 3 respectively as the setting in [47].

The step (iv) takes the criterion (4) into consideration and only keeps a clone group if the code changes involved in the group are semantically similar or related to each other. Although there are some clone detection tools aiming for detecting semantic or functional similar code [15], [27], their settings may not be suitable for us. To reduce possible falsely identified widespread changes, we choose to manually investigate all of the syntactically similar code changes reported by the step (iii) and use human judgments to decide whether each group of syntactically similar code changes is also semantically similar or related, and only keep the groups of code changes that we can determine to be semantically similar or related to each other by investigating their surrounding codes and understanding their usage context.

Note that in the step (ii), we focus on detecting widespread changes that occur within the same revision. This means that we apply the code clone tool to the hunks from each revision separately from the hunks from other revisions. It will be straightforward to extend our approach to detect widespread

TABLE I
CATEGORIES OF WIDESPREAD CHANGES.

Types of widespread changes	Description
Argument Addition /Removal	Add or drop argument(s) for a method.
Argument Change	Change the value, type, name of the argument(s) for a method.
Method Addition	Extract a piece of codes to be a method, or add methods to get or set variables.
Method Change	Change the name, access control, return type for a method or change deprecated methods.
Data Structure	Change the name, access control or the field of data structure, or change the data structure to be used(e.g., Change map to set).
Feature Addition	An addition/implementation of a new feature.
Algorithm Change	Algorithm or implementation change.
External Change	The change caused by the change of external dependent library or interacting system.
Non Functional	Other kinds of code changes that do not change the code functionalities, documentation change, comment change, or fix warnings.
Bug Fix	Fix a bug. It includes all kinds of code changes that are made for the purpose of fixing a bug.
Others	Except above categories.

changes across multiple revisions by applying the code clone tool to the combined set of hunks from all revisions.

V. CATEGORIZATION OF WIDESPREAD CHANGES

After running the detection component described in Section IV, we obtain many widespread changes. In this step, we would like to facilitate the understanding of the widespread changes by manually classifying them into various categories based on the reasons that cause them to occur.

Firstly, in order to create the categories, one of our authors, who has five years Java programming experience, manually look into all code corresponding to the widespread changes. We also investigate comments in the commit logs which give us additional insights on why those changes are made. After a number of iterations, we come up with 10 categories based on our understanding of the reasons for the code changes: *argument addition/removal*, *argument change*, *method addition*, *method change*, *data structure*, *external change*, *non-functional*, *feature addition*, *algorithm change*, and *bug fix*. Also, we use another category *others* for all of the widespread changes that do not fit any one of the above 10 categories.

The 11 categories are shown in Table I with their descriptions. The first seven categories are related to addition, removal, or change of arguments, methods, data structures, features, or algorithms. The other remaining categories correspond to changes that are due to an external change (e.g., a change in a commonly used API), non-functional changes, bug fixes, and others.

Secondly, after the above categories are created, we tag each of the widespread changes with the categories. For each widespread change, it is possible that more than one category can apply. In such cases, we pick one category that is the best fit for the change based on our human judgement. Section VI-E shows various sample code changes belonging to some categories together with our empirical evaluation results.

VI. EMPIRICAL EVALUATION

In this section, we first describe the datasets and our experimental settings. We then present our research questions and the answers. Finally, we present some threats to validity.

A. Datasets

By using the method described in Section III, we collect code change histories for eight open source projects from Sourceforge[5] including: ArgoUML [1], Columba [2], Lightweight Java Game Library [5], JFreeChart [4], MegaMek [6], PDF Split and Merge [7], TightVNC [8], and JEdit [3]. The projects are listed in Table II. All of them are at least 5 years old and use Subversion[6] as their version control systems. These projects implement various functionalities ranging from email clients to games to editors, etc. Each of them has a relatively large user base.

For each of the eight programs, we collect revision information from their respective code repositories. The number of revisions we collect for each program is listed in Table III. The table also shows the time period during which the revisions are made. The average number of revisions per program is 4,057. The sizes of the revisions that we collect for each program range from 18.5 MB to 1.18GB, and the average size across all programs is 330 MB.

TABLE III
NUMBER OF REVISIONS FOR EACH PROGRAM.

Project	# Revisions	Period
ArgoUML [1]	7170	01/01/2007-09/05/2012
Columba [2]	458	09/07/2006-09/07/2011
Lightweight Java Game Library [5]	3777	24/07/2002-04/06/2012
JFreeChart [4]	2468	19/06/2007-13/06/2012
MegaMek [6]	8806	21/02/2002-09/05/2012
PDF Split and Merge [7]	1173	03/01/2007-06/03/2012
TightVNC [8]	2813	08/10/2004-22/04/2010
JEdit [3]	5797	27/07/2006-07/05/2012

We have implemented our approach in various scripts written in Java, and performed our study on a workstation with an Intel Core i7 3.3GHz CPU installed with 6GB of memory and 700GB of hard disks.

B. Research Questions

With the above collected datasets, we investigate the following research questions in order to get better understanding of widespread changes:

- RQ1 How accurate is our clone detection based approach in recovering widespread changes?
- RQ2 How frequent do widespread changes occur during software development and maintenance?
- RQ3 What are the various types of widespread changes?

[5]http://sourceforge.net/
[6]http://subversion.apache.org/

TABLE II
PROJECTS INVESTIGATED IN THIS STUDY.

Project	Description	Lines of Code in the latest revision	Language
ArgoUML	ArgoUML is a free and feature-rich open source UML modeling tool which support for latest UML standard.	369,427	Java
Columba	Columba is a powerful email management tool which has a friendly graphical interface with wizards.	194,125	Java
Lightweight Java Game Library	A Java Game Library extension which allow developers to easily developer 2D or 3D games by accessing high performance crossplatform libraries such as OpenGL, OpenCL and OpenAL.	138,198	Java
JFreeChart	JFreeChart is a free (LGPL) chart library written in Java.	327,865	Java
MegaMek	MegaMek is a open source, network Classic BattleTech board game.	382,740	Java
PDF Split and Merge	PDF Split and Merge is a tool to allow user to merge and split pdf documents easily.	30,427	Java
TightVNC	TightVNC is a great free remote-desktop tool which allows to remote network access to graphical desktops.	74,468	Java, C, C++
JEdit	jEdit is a mature and powerful programmer's text editor.	183,280	Java

C. RQ1: Accuracy of Clone-Based Approach

In our approach, we have both automated and manual parts. The automated part makes use of a clone detection tool to detect syntactically similar code changes spread at multiple locations in multiple files. These automatically detected changes are then subject to manual inspection to see if they are *semantically* similar or not. In our first research question, we are interested in investigating the accuracy of the automated part of our approach. In effect, we are answering the following question: How many syntactically similar code changes automatically detected from multiple locations in multiple files are also semantically similar?

In this study, the automated part of our approach returns 999 syntactically similar groups of hunks or partial hunks. We manually inspect all of the 999 groups and find that 965 of them are semantically similar and are thus widespread changes. Thus, the accuracy of the automated part of our approach is 96.66%, which is very high. This justifies the use of a code clone detection tool in reducing efforts in the manual parts in our approach. However, note that we limit our clone detection to syntax-based clones of relatively small sizes; it would be interesting future work to investigate the effects of larger clones and tools that could detect semantic clones [15], [27], [33] on detecting widespread changes.

An example of such inaccuracy is shown in Figure 3. The group of clones shown are syntactically similar but they are not semantically similar. For this example, we can see that these clones all contain common terms *Model* and *getFacade* and `if` statements, but they use if statements to check different things which means they are not semantically similar.

```
//@@//UMLMetaClassComboBoxModel-135.java
+             if (Model.getFacade().isATagDefinition(t)) {

//@@//UMLModelElementNamespaceComboBoxModel-172.java
+             if (Model.getFacade().isANamespace(o)) {
+             if (Model.getFacade().isAGeneralization(o)) {

//@@//UMLGeneralizationPowertypeComboBoxModel-150.java
+             if (Model.getFacade().isAClassifier(o)) {

//@@//UMLLinkAssociationComboBoxModel-148.java
+            if (Model.getFacade().isAAssociation(n)) {

//@@//UMLLinkAssociationComboBoxModel-148.java
+            if (Model.getFacade().isALink(o)) {

//@@//UMLMetaClassComboBoxModel-135.java
+             if (Model.getFacade().isATagDefinition(t)) {

//@@//UMLModelElementNamespaceComboBoxModel-172.java
+             if (Model.getFacade().isANamespace(o)) {
```

Fig. 3. Syntactically but not Semantically Similar Changes

D. RQ2: Frequencies of Widespread Changes

We analyze a total of 32,452 revisions from the eight software systems. We detect 965 widespread changes. Thus, not all revisions contain a widespread change, and on average there is one widespread change in every 34 revisions. In total, there are 502 revisions that contain at least one widespread change. Each revision contains at most 11 widespread changes.

The number of widespread changes for each software system is shown in Table IV. We notice that the absolute numbers of widespread changes in different software systems differ a lot, ranging from 2 to 484, and the proportion of revisions containing widespread changes in each system also differs from 0.04% to 10.82%. We see that JFreeChart has the highest number of revisions containing widespread changes, and that TightVNC has the lowest number of revisions containing widespread changes. The pie chart in Figure 4 further illustrates the proportions of widespread changes contained in each program with respect to the total number (965) of widespread changes.

E. RQ3: Types of Widespread Changes

In Table I, we list all types of widespread changes and their corresponding descriptions. We manually investigate each of the 965 widespread changes and assign each of them into one of these categories. Table V shows the numbers of widespread changes belonging to various categories. The pie chart in Figure 5 further shows the percentage of widespread changes for each category. Only 1 (0.1%) of the widespread changes belong to the category of *others*. This shows that our 10 categories are comprehensive enough in capturing many

TABLE IV
NUMBERS OF WIDESPREAD CHANGES PER SOFTWARE SYSTEM

Software System	# Widespread Changes	# Revisions Containing Widespread Changes	% of Revisions Containing Widespread Changes
ArgoUML	127	70	0.98%
Columba	8	4	0.87%
Lightweight Java Game Library	49	26	0.69%
JFreeChart	484	267	10.82%
MegaMek	255	110	1.25%
PDF Split and Merge	24	11	0.94%
TightVNC	2	1	0.04%
JEdit	16	13	0.22%

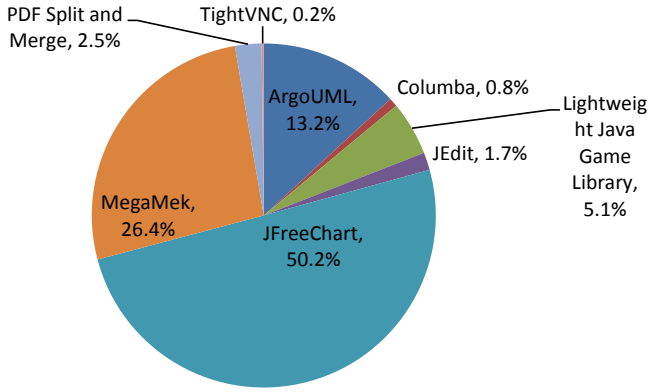


Fig. 4. Proportion of widespread changes contained in each program.

different kinds of widespread changes.

TABLE V
NUMBERS OF WIDESPREAD CHANGES PER CATEGORY.

Types of widespread changes	Numbers of widespread changes
Argument Addition/Removal	34
Argument Change	20
Method Change	25
Data Structure	68
Non Functional	487
Feature Addition	113
Algorithm Change	96
Method Addition	41
External Change	4
Bug Fix	76
Others	1

We find that widespread changes belonging to the *non-functional* category dominate the distribution (50.4%). An example of this kind of widespread change is shown in Figure 6. We just show one of the many changes spread in many files. It shows a change to the comments of classes related to Decorator.

Aside from the *non-functional* category, the next three largest categories of widespread changes in our dataset are *feature addition*, *algorithm change*, and *bug fix*. We show an example of a widespread change involving each of these categories in Figures 7, 8, and 9 respectively.

Figure 7 shows a feature addition widespread change which appears in several locations. The developer of Megamek, a game application, added a new feature (i.e., a new ammo type) and thus needed to add the new conditions into related `if`

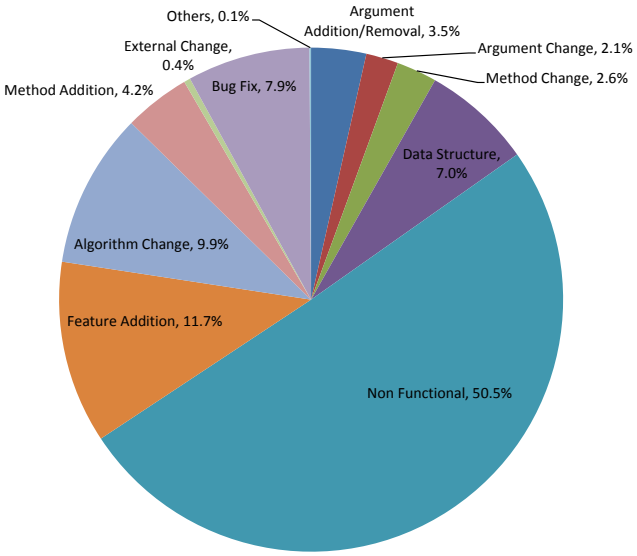


Fig. 5. Proportion of widespread changes in each change category.

```
 /**
+ * This Decorator is responsible for generating commands for any
- * This Decorator is responsible for generating mementos for any
  *
  *
  * @author Linux Tolke
```

Fig. 6. Example of Non-Functional Widespread Change from ArgoUML

statements in multiple places. Figure 8 shows an algorithm change widespread change. It appears in a number of locations and describes an additional else branch being added to change the behavior of the algorithm. Figure 9 shows a fix of a bug occurring in many locations that involves a file not being closed after it is no longer used.

```
        }else if(useAmmo
+               && ((atype.getAmmoType() == AmmoType.T_AC_LBX)
+               || (atype.getAmmoType() == AmmoType.T_AC_LBX_THB))
-               && (atype.getAmmoType() == AmmoType.T_AC_LBX)
```

Fig. 7. Example of Feature Addition Widespread Change from Megamek

Aside from *others*, the least common families of widespread changes are *external change*, *argument change*, and *method change*. We show an example of a widespread change for each of these categories in Figures 10, 11, and 12 respectively.

Figure 10 shows an external change widespread change

```
 if (dataFile.exists("armor_type")) {
            a.setArmorType(dataFile.getDataAsInt("armor_type")[0]);
+        } else {
+            a.setArmorType(EquipmentType.T_ARMOR_STANDARD);
         }
```

Fig. 8. Example of Algorithm Change Widespread Change from Megamek

```
        finally
        {
+               loading = false;
+               try
+               {
+                       if(in != null)
+                               in.close();
+               }
+               catch(IOException io)
+               {
+                       Log.log(Log.ERROR,Registers.class,io);
+               }
```

Fig. 9. Example of Bug Fix Widespread Change from PDF Split and Merge

that is caused by an upgrade in JUnit 4. Many places in the code need to change the called API (`assertEquals`) in the same way. Figure 11 presents one change from a group of changes that add a modifier (`final`) to the arguments of various methods. Figure 12 shows a widespread change caused by the change of a method name. This change needs to be made at various locations that invoked the same method (i.e., getDefaultWorkingDir).

```
        for(int i =0; i < v1.size(); i++)
+               Assert.assertEquals((String) v1.get(i),
                                    (String) v2.get(i));
-               assertEquals((String) v1.get(i),
                                    (String) v2.get(i));
        }
```

Fig. 10. Example of External Change Widespread Change from Columba

```
+       public RotateException(final int exceptionErrorCode,
                  final Throwable e) {
-       public RotateException(int exceptionErrorCode,
                        Throwable e) {
                super(exceptionErrorCode, e);
        }
```

Fig. 11. Example of Argument Change Widespread Change from PDF Split and Merge

```
        config = Configuration.getInstance();
+       fileChooser = new JFileChooser(
                        config.getDefaultWorkingDirectory());
-       fileChooser = new JFileChooser(
                        config.getDefaultWorkingDir());
```

Fig. 12. Example of Method Change Widespread Change from PDF Split and Merge

In Table VI, we list the average number of LOC involved in one widespread change for each category. We notice that, aside from widespread changes from the category *others*, a non-functional widespread change involves the highest number of lines of code on average (i.e., 43.2 lines). Bug fix is the next category with the highest number of lines of code per widespread change (i.e., 31.3 lines). Categories involving the least numbers of lines of code per widespread change are method change (11.7 lines), and external change (8.4 lines).

In Figure 13, we present how much each program contributes to each family of widespread changes. Figure 13(a) shows the numbers of widespread changes from each program

TABLE VI
AVERAGE LINES OF CODE PER WIDESPREAD CHANGE FOR EACH CHANGE CATEGORY.

Types of widespread changes	Avg. Lines of Code
Argument Addition/Removal	19.7
Argument Change	16.4
Method Change	11.7
Data Structure	20.9
Non Functional	43.2
Feature Addition	30.5
Algorithm Change	22.3
Method Addition	31.1
External Change	8.4
Bug Fix	31.3
Others	50

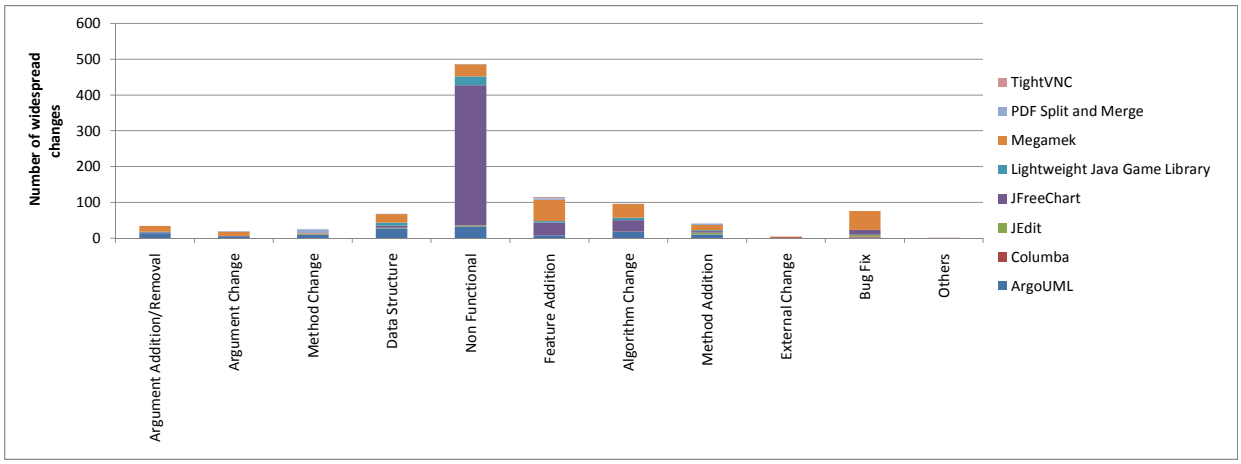
for each family of widespread changes, while Figure 13(b) shows the corresponding percentages.

We can see that most *argument addition/removal* changes are from Megamek and ArgoUML. Most *argument change* changes are from Megamek. Most *method addition/removal* changes are from Megamek. Most *method change* changes are from PDF Split and Merge and ArgoUML. Most *data structure* changes are from ArgoUML and Megamek. Most *non-functional* changes are from JFreeChart. Most *feature addition* changes are from Megamek. Most *algorithm change* changes are from Megamek and JFreeChart. Most *external change* changes are from Columba. Most *bug fix* changes are from Megamek. All *other* changes are from Columba.

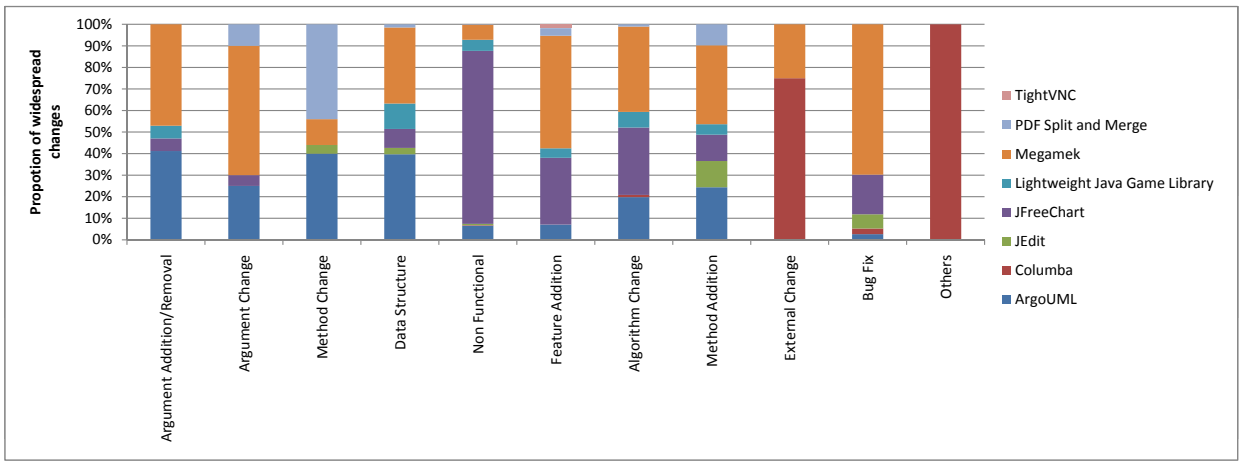
Three software systems deserve further mentioning. Megamek contributes the most to most families of widespread changes with a few exceptions: *method change*, *non-functional*, *external change*, and *other*. PDF Split and Merge dominates *method change*. *JFreeChart* dominates *non-functional* and there are many (386 changes) such changes in *JFreeChart*. Columba dominates *external change* and *other*.

F. Discussion

1) Frequency of Widespread Changes: We note that we could find 965 widespread changes from more than 30,000 revisions. Although there are many widespread changes, many revisions do not include widespread changes. Also, the frequencies of widespread changes occurring in the revisions of various software systems vary widely from 0.04% to 10.82%. Many tools proposed in the literature, such as automatic bug fixing via recurring bug fix detection (e.g., [44]) and bug detection via inconsistent code clone detection (e.g., [26]), rely on the phenomenon of widespread changes to some extent. It is unclear if these tools remain effective on programs with a low number of widespread changes. It would be interesting to investigate the correlation between the numbers of widespread changes and the effectiveness of these tools in the future. It also would be interesting to investigate the correlation between the numbers of widespread changes and various characteristics (e.g., project sizes, numbers of developers involved) of the subject programs as well in the future. We note that we may have not detected all widespread changes in the projects due to the settings of our detection approach; it is also valuable



(a) Numbers of Widespread Changes Per Program For Each Change Category



(b) Proportions of Widespread Changes Per Program For Each Change Category

Fig. 13. Contribution of Each Program to Each Family of Widespread Changes

future work to estimate the recall rates of our approach and further confirm the implications of our results.

2) Prevalence of Non-Functional Widespread Change: We note that there are many widespread changes that are non-functional. These non-functional changes numbered to 487 out of 965 widespread changes that we collect (50.5%). This points to the need to maintain these non-functional changes. We believe not only code clones need to be maintained [12], [23], [45], but these and yet pervasive changes to non-functional code or even non-code (e.g., documents and comments) need to be similarly maintained too. Poor comments could lead to bugs as future developers and system maintainers could make wrong assumptions based on outdated comments. Tan et al. have shown that discrepancies between comments and code could be used to find bugs [53], [54]. Other studies also show that some comments include licensing information and it is important to manage licensing information well [16], [17].

3) Types of Widespread Changes: There are many different types of widespread changes which potentially need different tools to handle. Past studies on program transformations have primarily focused on bug fixes [38] and external changes [46]. It would be interesting to investigate the extent to which these tools are applicable to other types of widespread changes and identify the issues for improvements.

G. Threats to Validity

Threats to internal validity includes experimenter bias. We manually inspect each structurally similar code change and decide if it is semantically similar or not. We might make some mistakes in the process. We also manually categorize each widespread change to one out of the 11 categories. Again, there might be some bias and error in this manual process.

Threats to external validity refers to the generalizability of our findings. We have analyzed 8 software systems. These systems implement a wide range of functionalities. We have analyzed more than 30,000 revisions and manually analyzed nearly one thousand widespread changes. Also, in this study, we focus on detecting widespread changes that occur within the same revision. In the future, we plan to investigate even more revisions from more software systems written in various programming languages.

Several threats to construct validity are related to our definitions of widespread changes and the 11 categories of widespread changes. In this work, we provide one definition

and one approach of detecting widespread changes. In future, we would like to investigate different kinds of definitions and approaches for detecting widespread changes. We use a set of fixed parameter values for our semi-automated widespread change detection approach, e.g., number of surrounding lines, number of lines changed, number of files affected, minimum number of tokens in detected code clones, minimum number of kinds of tokens, etc. Other parameter values could be used and we plan to investigate them in the future. In this work, we have decided 11 different categories of widespread changes. There might be better categorization. We plan to investigate different categorizations considering different granularity levels in the future. One possibility is to consider the different kinds of refactoring patterns[7].

VII. Related Work

In this section, we present closely related studies on code commits, software evolution, and code clones.

A. Studies on Code Commits

Our study investigates the source code (including comments) changed between two continuous versions of programs. It is different from many other studies that investigate different aspects of code commits. Hindle et al. perform a taxonomical study on large commits that involve many files [22] and propose an approach to automatically classify large changes into maintenance categories [21]. Kawrykow and Robillard look for non-essential changes in code commits [31]. Pham et al. [48] search for recurring bugs in software that gets reused. Wu et al. [60] propose an approach to automatically link code changes with corresponding bug reports together. Eyolfson et al. [13] find that the time and day of a code commit may affect the quality of the code. Murphy-Hill et al. [42] use a large scale study on refactoring changes and find that many refactoring changes are mixed with others and may not be clearly stated in commit logs.

B. Studies on Software Evolution

There are many studies on software maintenance and evolution and how they relate to software quality and productivity. Posnett et al. [51] investigate how new features and code improvements affect defects. Zhu et al. [62] propose the use of deviations of software modularity to monitor quality. Hammad et al. [19] use lightweight analysis and syntactic code differencing to determine how a code change would impact the system design. Xing and Stroulia investigate software evolution in Eclipse to analyze refactoring efforts [61]. They find that current tool support is not sufficient for complex refactorings. Kim et al. investigate API refactorings when software evolves over time [34]. They find after refactorings the number of bug fixes increases but the time taken to fix bugs decreases. Padioleau et al. analyze Linux device drivers and detect changes in drivers code that are due to changes in the kernel interfaces (a.k.a. collateral evolutions) [47]. Van Rysselberghe et. al. were among the first to investigate changes due to move operation by using a clone detector on code diffs [57]. Xing et al. present an approach for detecting refactorings based on information of system evolution at the design level. Different from their approach, our approach is based on the information at the source code level.

[7]http://www.refactoring.com/catalog/index.html

C. Studies on Code Clones

As briefly mentioned in Section IV, code clones have been widely studied in the literature. There are various kinds of clone detection techniques, based on similarities among strings, tokens, syntax trees, dependency graphs, and even program memory states and functionalities [9], [10], [29], [32], [33], [37], [40], [49], [56], [58]. With the evolving software, there are also studies aiming to detect clones incrementally and improve the clone detection efficiency when programs change [18], [20], [43].

Clones are traditionally thought as harmful, and techniques have been proposed to reduce clones [24], [52]. On the other hand, some studies show that clones can be useful and necessary [30], [35]. Then, instead of reducing clones, some studies investigate techniques to track and manage code clones [12], [23], [45]. Our work is also related to many studies on clone evolutions. Kim et al. [35] investigate how clones change across program versions. Cai and Kim [11] extend the work by studying long-lived clones and identify factors that may affect the survival time of clones.

Different from the above studies, our work studies clones residing within the *changed code* from one revision, instead of clones residing in the whole code base of a program. We focus on detecting and classifying widespread changes which is a different kind of phenomenon from clones themselves.

VIII. Conclusion and Future Work

Many studies in software engineering including recurring bug fixes, detection of copy-paste bugs, automated program transformations, code search, and many more are motivated by the assumption that many code changes are similar to one another and are spread across many locations. We refer to this phenomenon as widespread changes. In this work, we propose a semi-automated approach to recover widespread changes from program revisions. We also analyze these widespread changes and create a categorization consisting of 11 different families of widespread changes. We group the changes into these families and present the distribution of these families of widespread changes across 8 software projects. In total we analyze more than 30,000 revisions and manually analyze almost a thousand changes manually.

We find that widespread changes occur many times — 965 of them out of the 32,452 revisions that we analyze. Although many, they are a small proportion of the revisions. This highlights the need to investigate the applicability of many techniques that rely on widespread changes on software systems, in which widespread changes are not observed. We also find that the most common family of widespread changes is non-functional changes followed by feature addition, algorithm change, and bug fix. It would be interesting to investigate

the applicability of various program transformation tools in helping with each family of widespread change and design better support for developers in performing the most common widespread changes in the future.

IX. Acknowledgments

We would like to thank anonymous reviewers for providing valuable comments to revise our paper.

References

[1] "ArgoUML," http://argouml.tigris.org/.

[2] "Columba," http://sourceforge.net/projects/columba/.

[3] "JEdit," http://www.jedit.org/.

[4] "JFreeChart," http://www.jfree.org/jfreechart/.

[5] "Lightweight Java Game Library," http://www.lwjgl.org/.

[6] "MegaMek," http://megamek.info/.

[7] "PDF Split and Merge," http://www.pdfsam.org/.

[8] "TightVNC," http://www.tightvnc.com/.

[9] B. S. Baker, "Finding clones with Dup: Analysis of an experiment," *IEEE TSE*, 2007.

[10] I. D. Baxter, C. Pidgeon, and M. Mehlich, "DMS©: Program transformations for practical scalable software evolution," in *ICSE*, 2004.

[11] D. Cai and M. Kim, "An empirical study of long-lived code clones," in *FASE*, 2011, pp. 432–446.

[12] E. Duala-Ekoko and M. P. Robillard, "Tracking code clones in evolving software," in *ICSE*, 2007.

[13] J. Eyolfson, L. Tan, and P. Lam, "Do time of day and developer experience affect commit bugginess?" in *MSR*, 2011, pp. 153–162.

[14] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[15] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *ICSE*, 2008, pp. 321–330.

[16] D. M. Germán, Y. Manabe, and K. Inoue, "A sentence-matching method for automatic license identification of source code files," in *ASE*, 2010, pp. 437–446.

[17] D. M. Germán and M. D. Penta, "A method for open source license compliance of java applications," *IEEE Software*, vol. 29, no. 3, pp. 58–63, 2012.

[18] N. Göde and R. Koschke, "Incremental clone detection," in *CSMR*, 2009, pp. 219–228.

[19] M. Hammad, M. L. Collard, and J. I. Maletic, "Automatically identifying changes that impact code-to-design traceability during evolution," *Software Quality Control*, vol. 19, no. 1, pp. 35–64, Mar 2011.

[20] Y. Higo, Y. Ueda, M. Nishino, and S. Kusumoto, "Incremental code clone detection: A pdg-based approach," in *WCRE*, 2011, pp. 3–12.

[21] A. Hindle, D. M. Germán, M. W. Godfrey, and R. C. Holt, "Automatic classication of large changes into maintenance categories," in *ICPC*, 2009, pp. 30–39.

[22] A. Hindle, D. M. Germán, and R. C. Holt, "What do large commits tell us? a taxonomical study of large commits," in *MSR*, 2008, pp. 99–108.

[23] P. Jablonski and D. Hou, "CReN: A tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE," in *ETX*, 2007.

[24] S. Jarzabek and S. Li, "Eliminating redundancies with a "composition with adaptation" meta-programming technique," in *ESEC/FSE*, 2003.

[25] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *ICSE*, 2007.

[26] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs," in *ESEC/SIGSOFT FSE*, 2007.

[27] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *ISSTA*, 2009.

[28] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs," in *ESEC/SIGSOFT FSE*, 2007, pp. 55–64.

[29] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE TSE*, 2002.

[30] C. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful," in *WCRE*, 2006.

[31] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," in *ICSE*, 2011, pp. 351–360.

[32] I. Keivanloo, J. Rilling, and P. Charland, "Internet-scale real-time code clone search via multi-level indexing," in *WCRE*, 2011, pp. 23–27.

[33] H. Kim, Y. Jung, S. Kim, and K. Yi, "MeCC: memory comparison-based clone detector," in *ICSE*, 2011, pp. 301–310.

[34] M. Kim, D. Cai, and S. Kim, "An empirical investigation into the role of api-level refactorings during software evolution," in *ICSE*, 2011, pp. 151–160.

[35] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in *ESEC/FSE*, 2005.

[36] S. Kim, K. Pan, and E. J. Whitehead, Jr., "Memories of bug fixes," in *FSE*, 2006, pp. 35–45.

[37] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *SAS*, 2001.

[38] J. L. Lawall, B. Laurie, R. R. Hansen, N. Palix, and G. Muller, "Finding error handling bugs in openssl using coccinelle," in *EDCC*, 2010.

[39] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: A tool for finding copy-paste and related bugs in operating system code," in *OSDI*, 2004.

[40] ——, "CP-Miner: A tool for finding copy-paste and related bugs in operating system code," in *OSDI*, 2004.

[41] C. McMillan, N. Hariri, D. Poshyvanyk, J. Cleland-Huang, and B. Mobasher, "Recommending source code for use in rapid software prototypes," in *ICSE*, 2012, pp. 848–858.

[42] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," in *ICSE*, 2009, pp. 287–297.

[43] T. T. Nguyen, H. A. Nguyen, J. M. Al-Kofahi, N. H. Pham, and T. N. Nguyen, "Scalable and incremental clone detection for evolving software," in *ICSM*, 2009, pp. 491–494.

[44] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Recurring bug fixes in object-oriented programs," in *ICSE (1)*, 2010, pp. 315–324.

[45] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Clone-aware configuration management," in *ASE*, 2009.

[46] Y. Padioleau, J. L. Lawall, R. R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in linux device drivers," in *EuroSys*, 2008.

[47] Y. Padioleau, J. L. Lawall, and G. Muller, "Understanding collateral evolution in linux device drivers," in *SIGOPS EuroSys*, 2006, pp. 59–71.

[48] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Detection of recurring software vulnerabilities," in *ASE*, 2010, pp. 447–456.

[49] A. Podgurski and L. Pierce, "Retrieving reusable software by sampling behavior," *ACM TOSEM*, 1993.

[50] D. Poshyvanyk and M. Grechanik, "Creating and evolving software by searching, selecting and synthesizing relevant source code," in *ICSE Companion*, 2009, pp. 283–286.

[51] D. Posnett, A. Hindle, and P. T. Devanbu, "Got issues? do new features and code improvements affect defects?" in *WCRE*, 2011, pp. 211–215.

[52] D. C. Rajapakse and S. Jarzabek, "Using server pages to unify clones in web applications: A trade-off analysis," in *ICSE*, 2007.

[53] L. Tan, Y. Zhou, and Y. Padioleau, "acomment: mining annotations from comments and code to detect interrupt related concurrency bugs," in *ICSE*, 2011, pp. 11–20.

[54] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@tcomment: Testing javadoc comments to detect comment-code inconsistencies," in *ICST*, 2012, pp. 260–269.

[55] S. Thummalapenta and T. Xie, "Parseweb: A programmer assistant for reusing open source code on the web," in *ASE*, 2007, pp. 204–213.

[56] M. S. Uddin, C. K. Roy, K. A. Schneider, and A. Hindle, "On the effectiveness of simhash for detecting near-miss clones in large scale software systems," in *WCRE*, 2011, pp. 13–22.

[57] F. Van Rysselberghe, M. Rieger, and S. Demeyer, "Detecting move operations in versioning information," in *Proceedings of the Conference on Software Maintenance and Reengineering*, ser. CSMR '06, 2006.

[58] V. Wahler, D. Seipel, J. W. von Gudenberg, and G. Fischer, "Clone detection in source code by frequent itemset techniques," in *SCAM*, 2004.

[59] S. Wang, D. Lo, and L. Jiang, "Code search via topic-enriched dependence graph matching," in *WCRE*, 2011, pp. 119–123.

[60] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "Relink: recovering links between bugs and changes," in *ESEC/FSE*, 2011, pp. 15–25.

[61] Z. Xing and E. Stroulia, "Refactoring practice: How it is and how it should be supported - an eclipse case study," in *ICSM*, 2006.

[62] T. Zhu, Y. Wu, X. Peng, Z. Xing, and W. Zhao, "Monitoring software quality evolution by analyzing deviation trends of modularity views," in *WCRE*, 2011, pp. 229–238.