

A Change-Type Based Empirical Study on the Stability of Cloned Code

Md Saidur Rahman Chanchal K. Roy

Department of Computer Science, University of Saskatchewan, Canada
{saeed.cs, chanchal.roy}@usask.ca

Abstract—Clones are the duplicate or similar code blocks in software systems. A large number of studies concerning the impacts of clones on software systems mainly focus on the frequency of changes to evaluate stability, consistency in evolution and introduction of bugs. Although it is obvious that not each type of changes has equal impact on software systems, none of the existing studies take the types of changes and their significance into account during comparative evaluation of stability of cloned and non-cloned code. This paper presents an empirical study on the comparative stability of cloned and non-cloned code from the perspective of different change types. Changes from successive revisions are extracted and classified using *ChangeDistiller* which employs Abstract Syntax Tree (AST) differencing of the successive revisions of source code and assigns the corresponding level of significance to each of the classified changes. We detect exact (Type-1) and near-miss (Type-2 and Type-3) clones using the hybrid clone detection tool NiCad. Extracted and classified changes and clone information are then analyzed to compare the stability of cloned and non-cloned code from three different perspectives: types of clones, types of changes with respect to the significance of changes, and size and extent of evolution of the systems. Our study on seven open-source Java systems with diversity in their size, length of evolution and application domain shows that changes are more frequent in cloned code than in non-cloned code and Type-1 clones are comparatively more vulnerable to the stability of the systems. Therefore, cloned code is less stable than non-cloned code suggesting that cloned code is likely to pose more maintenance challenges than non-cloned code.

Keywords—Code Clones, Clone Evolution, Change Significance, Change Frequency, Stability

I. INTRODUCTION

Code reuse by copy-paste is a common practice in software development, and as a result software systems often have sections of code that are identical or similar, called software clones or code clones. Clones constitute a significant fraction of code (between 7% and 23% [1] or sometimes even 50% [2]). Regardless of the intentional and unintentional reasons behind code cloning, the impact of clones on software maintenance has been a great concern [3].

Although it is believed that code cloning speeds up software development and facilitates the reuse of mature and tested code, clones are often accused of introducing maintenance challenges by making consistent changes more difficult leading to introduction of bugs [4], propagation of existing bugs and thus resulting in increasing maintenance efforts [5]. Having both positive and negative impacts on software maintenance, code cloning has been under significant research focus towards the evaluation of the impacts of clone on software maintenance and evolution. Some of the existing studies concluded in favour of clones suggesting that clones are not harmful [6], [7], [8],

[9], rather, cloning can be beneficial to software development [10]. A good number of studies on the other hand concluded that clones have negative impacts on software maintenance and evolution [4], [5], [7], [11].

One of the well-studied perspectives of the evaluation of the comparative impacts of cloned and non-cloned code on software maintenance and evolution is the *stability*. Stability measures the extent to which cloned and non-cloned code regions remain unchanged. Code regions that change less frequently are more stable and thus require less maintenance effort. Krinke [8] measured the comparative stability of clone and non-clone code by measuring the volume of code change, *i.e.*, by counting the number of lines added, deleted and changed to clone and non-clone code. His study concludes that clone code is more stable than non-clone code. Göde and Harder [12] extended Krinke’s study [8] with token-based incremental clone detection tool and experimented with different settings of clone detection parameters, clone types and change operations. Their study agrees with the findings of Krinke’s study. Krinke’s other study [9] measures stability in terms of the comparative age of the cloned and non-cloned code based on the average last change date. This study agrees with his previous findings that cloned code is more stable. Hotta *et al.* [7] measured the modification frequency of cloned and non-cloned code to evaluate their comparative stability. This study concludes that cloned code is less frequently modified than non-cloned code, *i.e.*, cloned code is more stable. Lozano and Wermelinger [5], [11] conducted studies on assessing the impacts of clones and they concluded that clones are harmful for software maintenance because they often increase the maintenance efforts and also they are vulnerable to the stability of the software systems. Their findings disagree with the findings of Hotta *et al.* [7], Krinke [8] and Göde and Harder [12].

Because of these contradictory findings in earlier studies Mondal *et al.* [13], [14] carried out a comprehensive stability analysis within a uniform evaluation framework considering the existing metrics (mostly taken from [7], [8], [9]) and their proposed metrics to measure stability. Their study concluded that a firm conclusion cannot be drawn on the stability of cloned and non-cloned code, rather, the comparative stability varies with programming languages, types of clones and overall system development strategies. However, none of the metrics used or proposed in their study consider actual syntactic change types and the different levels of the significance of changes.

As it is obvious that different types of changes have different impacts on change propagation, it is important to consider the distinct change types in evaluating the impacts of clones

on maintenance. Considering the differences in the impacts of different change types, this study carries out an empirical evaluation of the impacts of clones on software maintenance. In particular, we investigate the frequency of changes in cloned and non-cloned code considering fine-grained change types and their levels of significance. In our empirical study, we consider a comprehensive taxonomy of source code change proposed by Fluri and Gall [15]. This taxonomy assigns a level of significance to each type of source code change where *significance* is defined as how strong impact a change may have on other source code entities. In this taxonomy there are four different levels of change significance: *low*, *medium*, *high*, *crucial* where *low* is the lowest level and *crucial* is the highest level of significance respectively.

We extract fine-grained changes from all successive revisions of software systems using *ChangeDistiller* [15] which employs Abstract Syntax Tree (AST) differencing between two consecutive revisions of source code files. One important advantage of the AST-based code differencing over UNIX *diff* is that AST-based differencing takes syntactic context of the code change into account while *diff* considers code merely as text. In addition, the results of *diff* might be sensitive to the formatting of the source code. AST-based code differencing on the other hand gives more fine grained changes of source code artifacts [15]. Extracted and classified changes from *ChangeDistiller* are mapped to the source code entities and stored for further analysis. We detect both exact (Type-1) and near-miss (Type-2 and Type-3) clones of all the revisions of the systems using the hybrid clone detection tool NiCad [16], [17]. The classified changes are then mapped to the cloned and non-cloned code. We then measure the frequency metrics which we define for assessing the comparative impacts of clones on software maintenance with respect to the levels of significance of changes to cloned and non-cloned code.

In particular, we evaluate the comparative stability of cloned and non-cloned code with respect to different levels of significance of changes by answering the following research questions:

- RQ1** *To what extent different types of clones exhibit different stability scenarios?*
- RQ2** *Do the changes of different levels of significance show different stability scenarios?*
- RQ3** *Is the stability of cloned and non-cloned code system dependent?*

Changes of higher level of significance are likely to have higher change impact. Therefore, the higher the frequency of changes of a particular level of significance in the code, the higher maintenance challenges it is likely to pose during evolution.

From this empirical study considering fine-grained change types and their different levels of significance we have the following findings:

(i) Cloned code is less stable than non-cloned code and Type-1 clones are comparatively more vulnerable to the stability of the system. (ii) Stability of cloned code is mostly affected by the changes of lower (*low*, *medium*) levels of significance. (iii) System size and length of evolution do not have significant effects on stability. Moreover, our fine-grained analysis gives important insights into the better management of clones.

The rest of the papers is organized as follows: Section II outlines the important motivation of the study. Section III briefly describes the taxonomy of changes used in this study. Section IV represents the experimental settings and steps used in the study including the subject systems used, change extraction and classification procedure, clone detection and the metrics we measure. Section V represents the experimental results and analysis. Threats to the validity of the study is represented in section Section VI. Section VII discusses related works followed by the conclusion in Section VIII.

II. MOTIVATION

This study is inspired by two previous studies by Krinke [8] and Hotta *et al.* [7] both of which measure the comparative stability of cloned and non-cloned code. Krinke defines *instability* of cloned and non-cloned code in terms of the number of added, deleted and changed lines with respect to total number of lines in cloned and non-clone code respectively. Higher value of instability indicates that the code region is less stable *i.e.*, changes more frequently. Although his method considers the volume of change in terms of lines of code modification to measure instability, it does not consider the actual syntactic change types.

Hotta *et al.* [7], on the other hand, measure the stability of cloned and non-cloned code in terms of *modification frequency*. Their approach counts the number of regions (blocks of consecutive lines of code) modified in cloned and non-cloned code. The average modification count per revision in cloned and non-cloned code are then used to measure the modification frequency by scaling the modification count to the ratio of cloned and non-cloned LOC (lines of code) to the total LOC. The less the modification frequency, the more stable the code region is. One of the major limitations of this approach is that it ignores the volume of change (as opposed to Krinke [8]), neither does it consider the types of actual syntactic changes. This is likely to affect the accuracy and reliability of the stability results. Thus, the measure of the stability of the code should consider the volume of change. And it is reasonable to assume that this stability measured in terms of frequency of changes represents the impact of changes on maintenance more precisely only when the actual syntactic types of changes are taken into consideration. This is because different types of syntactic changes have different impact (likelihood of affecting other entities) from the perspective of change propagation [15].

To have a deeper understanding of the aforementioned problems let us consider the two example change scenarios as shown in Figure 1 and Figure 2 as observed in revision 43 of DNSJava. In the first case, the method `fromString()` in file `DNS/Record.java` has two changes from revision 42 to revision 43. The first change is a *parameter type change* (`StringTokenizer` to `MyStringTokenizer`) which is of *crucial* significance because all the callers of the method need the corresponding *parameter type change* to be propagated. The second change is of type *statement update* with *low* significance as the effect of the change is local to the method.

By investigating the clone results and the source code, we see that this method has three Type-1 (exact) clones in files `org/xbill/DNS/dnsRecord.java`, `org/xbill/DNS/Record.java` and `DNS/dnsRecord.java`. All of the three clones have been updated consistently with the

```

135 public static dnsRecord fromString(StringTokenizer st, dnsName name, int ttl, short type, short dclass) throws IOException {
136 dnsRecord rec;
137 try {
138 Class rclass;
139 Constructor m;
140 String s = dns.typeString(type);
141 rclass = Class.forName("dns" + s + "Record");
142 m = rclass.getConstructor(new Class [] {
143 dnsName.class, java.lang.Short.TYPE, java.lang.Integer.TYPE, StringTokenizer.class });
144 rec = (dnsRecord) m.newInstance(new Object [] {
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159 return null;
160 }
161 )

```

Revision-42



```

135 public static dnsRecord fromString(MyStringTokenizer st, dnsName name, int ttl, short type, short dclass) throws IOException {
136 dnsRecord rec;
137 try {
138 Class rclass;
139 Constructor m;
140 String s = dns.typeString(type);
141 rclass = Class.forName("dns" + s + "Record");
142 m = rclass.getConstructor(new Class [] {
143 dnsName.class, java.lang.Short.TYPE, java.lang.Integer.TYPE, MyStringTokenizer.class });
144 rec = (dnsRecord) m.newInstance(new Object [] {
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159 return null;
160 }
161 )

```

Revision-43

Fig. 1: Changes in method fromString() in file DNS/Record.java in revision-43 of DNSJava

<pre> 116 StringBuffer toStringNoData() { 117 StringBuffer sb = new StringBuffer(); 118 sb.append(name); 119 sb.append("\t"); 120 sb.append(ttl); 121 sb.append("\t"); 122 sb.append(dns.typeString(type)); 123 if (dclass != dns.IN) { 124 sb.append("\t"); 125 sb.append(dns.classString(dclass)); 126 } 127 sb.append("\t"); 128 return sb; 129 } ... </pre> <p style="text-align: right;">Revision-42</p>		<pre> 116 StringBuffer toStringNoData() { 117 StringBuffer sb = new StringBuffer(); 118 sb.append(name); 119 sb.append("\t"); 120 sb.append(ttl); 121 if (dclass != dns.IN) { 122 sb.append("\t"); 123 sb.append(dns.classString(dclass)); 124 } 125 sb.append("\t"); 126 sb.append(dns.typeString(type)); 127 sb.append("\t"); 128 return sb; 129 } </pre> <p style="text-align: right;">Revision-43</p>
---	---	---

Fig. 2: Changes in method toStringNoData() in file DNS/Record.java in revision-43 of DNSJava

two above changes. Again, the method fromString() is called by the method doAdd (dnsMessage query, StringTokenizer st) in update.java and the method doAdd() is again called by the method main() in update.java. So, the first change introduces another *parameter type change* in doAdd() which is of *crucial* significance and also it requires one *statement update* in the method doAdd(). Therefore, the first change introduces three changes (*crucial*) to the three clones and one parameter change (*crucial*) and one statement update to the caller doAdd(), and one statement update change (*low*) to the main() method. So, the total number of changes triggered by the first change is six (3+2+1) while the second change introduces only three (1+1+1) changes of low significance to the three clones. Based on the depth of method call chain, the impact of changes can be even more in some other cases. This example makes it evident that not all changes pose equal challenges in the maintenance process.

Again, in Figure 2 method toStringNoData() in DNS/Record.java has three changes in revision 43. All these changes are *statement reordering* in the method body with *low* significance because the impact of this type of change is local to the changed method. However, by investigating clones and source code we see that these changes have been consistently propagated to three cloned methods of the method toStringNoData() in source files org/xbill/DNS/Record.java, DNS/dnsRecord.java and org/xbill/DNS/dnsRecord.java. Thus, each of the changes

in toStringNoData() introduces one corresponding change in the three Type-1 clone fragments. Now, we see that although the method toStringNoData() has more changes (three) than the number of changes (two) in fromString() in the first example, method fromString() poses higher maintenance cost than toStringNoData() as it introduces more changes due to having the change with higher levels of significance. Thus, from the above examples we could draw the following conclusions:

- (i) As different change types have different levels of significance, *i.e.*, different degrees of likelihood of affecting other entities, the significance levels of different types of code changes should be taken into account while comparing the impacts of changes in cloned and non-cloned code on software maintenance.
- (ii) Because of having different levels of significance, changes of different types should be considered as separate as possible while measuring the frequency of changes in code (*i.e.*, *stability*) in order to have more precise results of comparative evaluation of the stability of cloned and non-cloned code.

Although the existing studies give important insights regarding the impacts of clones on software maintenance, one important limitation is that none of the existing studies addressed the two important points mentioned above. Thus, the existing stability metrics [7], [8] have limitations from the perspectives of reliability and precision.

Our proposed study incorporates the actual change types and their levels of significance in measuring the frequency of changes in cloned and non-cloned code. Our study considers the type-wise relative volume of changes and their frequencies in cloned and non-cloned code for the comparative stability analysis. This enables us to comparatively evaluate the impacts of clones on maintenance in terms of stability from a more reliable and precise point of views.

III. TAXONOMY OF SOFTWARE CHANGE AT METHOD LEVEL

Software systems evolve through different kinds of changes. Each type of change refers to a particular syntactic context of the change. Again, each type of changes affects the software systems from different functional and structural contexts. For our study, we consider the taxonomy of change proposed by Fluri and Gall [15] which is a comprehensive taxonomy for fine-grained source code change and it defines the significance levels of changes. As our clone analysis is at method level granularity, we consider only the method level changes in the taxonomy proposed by Fluri and Gall. The taxonomy of changes used in this empirical study is presented in Table I.

Changes to individual methods are referred to as method-level changes. Changes to methods are further divided into two groups, changes to method declaration and changes to method body, based on where the changes occur in the methods. Changes to method declaration part (*method signature*) include changes in accessibility, overridability, method renaming, parameter change and changes to return type. Changes to parameters include addition, deletion, renaming, reordering and parameter type change. Changes to *method body* comprise changes to statements and structure statements (e.g., loop, branching). Statements might be added, deleted, modified and reordered. Each of these fine-grained change types is assigned a level of significance based on their likelihood of affecting other code entities and the extent they modify the functionality of the system as proposed by Fluri and Gall [15].

IV. EXPERIMENTAL SETUP

This section outlines the experimental settings for different components and steps of our empirical study including preprocessing of subject systems, clone detection, change extraction and classification and the metrics measured.

A. Subject Systems

This study is based on seven open source systems implemented in Java with diversified size, evolution history and application domain. Table II briefly represents the features the software systems including the application domain, length of evolution history, size in lines of code (LOC) and the total number of revisions. The size of the systems represents the lines of code (LOC) in the last revision of the systems counted after removal of comments and pretty-printing. In selecting subject systems we include some systems used in previous studies (Openymsg, Squirrel) by Hotta *et al.* [7] and Krinke [8] (ArgoUML) to have a better comparative analysis of the findings.

B. Metrics Measured

To compare the stability of cloned and non-cloned code, we measure the frequency of changes to cloned and non-cloned code with respect to different levels of significance of changes

and types of clones. The frequency metrics represent how frequently the cloned and non-cloned code encounter changes. To define the frequency metrics we assume that:

- $R = \{r_1, r_2, r_3, \dots, r_n\}$ is the set of revisions,
- $S = \{low, medium, high, crucial\}$ is the set of possible levels of change significance and

Now, we define the number of changes in cloned and non-cloned code with significance level s in revision r as $CC_{c_s}(r)$ and $CC_{n_s}(r)$ respectively.

We then define the following two frequency metrics in terms of the average number of changes to cloned and non-cloned code with a particular level of change significance as follows:

(i) The frequency of changes to cloned code (CF_c) is measured as

$$CF_c = \frac{\sum_{r \in R, s \in S} CC_{c_s}(r)}{|R|} \times \frac{\sum_{r \in R} LOC(r)}{\sum_{r \in R} LOC_c(r)} \quad (1)$$

(ii) The frequency of changes to non-cloned code (CF_n) is measured as

$$CF_n = \frac{\sum_{r \in R, s \in S} CC_{n_s}(r)}{|R|} \times \frac{\sum_{r \in R} LOC(r)}{\sum_{r \in R} LOC_n(r)} \quad (2)$$

In equations (1) and (2),

- $LOC_c(r)$, $LOC_n(r)$ and $LOC(r)$ represent the cloned, non-cloned and total lines of code in revision r respectively.
- the terms $\frac{\sum_{r \in R, s \in S} CC_{c_s}(r)}{|R|}$ and $\frac{\sum_{r \in R, s \in S} CC_{n_s}(r)}{|R|}$ represent the average number of changes per revision with significance s to cloned and non-cloned code respectively.

As the frequencies of changes to cloned and non-cloned code are sensitive to $LOC_c(r)$ and $LOC_n(r)$, we normalize the change frequencies by multiplying with the ratio of $LOC(r)$ to $LOC_c(r)$ and $LOC_n(r)$ for the cloned and non-cloned code respectively in equations (1) and (2).

Our proposed metrics are similar to the metrics proposed by Hotta *et al.* [7], but different from the following two points:

- Metrics proposed by Hotta *et al.* [7] count changes in a range of consecutive lines of code as a single change whereas we count each fine-grained change as a single change, and
- We consider the significance levels of changes and measure metrics for four levels of significance of changes separately.

C. Experimental Steps

We analyze the frequency of changes to cloned and non-cloned code by the following six steps:

1) *Preprocessing*: For the proposed analysis of the frequencies of changes in cloned and non-cloned code, we first extract all the revisions of the subject systems from their corresponding SVN repository. As our goal is to analyze the changes to source code we remove comments from the source files. Pretty-printing of the the source files are then carried out to eliminate the formatting differences using the tool *ArtisticStyle*¹.

¹<http://astyle.sourceforge.net/>

TABLE I: TAXONOMY OF METHOD-LEVEL CHANGES WITH SIGNIFICANCE (adapted from [15])

Change Level	Changed Part	Change Group	Change Type	Significance
Method	Declaration	Accessibility Change (MAC)	Accessibility Increase (MAI) Accessibility Decrease (MAD)	Medium High
		Overridability Change (MOC)	Add Method Overridability <i>final</i> (AMO) Delete Method Overridability (DMO)	Crucial Low
		Parameter Change (MPC)	Parameter Insert (PI) Parameter Delete (PD) Parameter Ordering (PO) Parameter Renaming (PR) Parameter Type Change (PTC)	Crucial Crucial Crucial Medium Crucial
		Method Identifier Change (MIC)	Method Renaming (MR)	High
		Return Type Change (RTC)	Return Type Insert (RTI) Return Type Delete (RTD) Return Data Type Change (RDC)	Crucial Crucial Crucial
	Body	Statement Change (SC)	Statement Insert (SI) Statement Delete (SD) Statement Update (SU) Statement Re-ordering (SO)	Medium Medium Low Low
		Structure Statement Change (SSC)	Condition Expression Change (CEC) Statement Parent Change (SPC) Alternative- part (<i>else</i>) Insert (API) Alternative- part (<i>else</i>) Delete (APD)	Medium Medium Medium Medium

TABLE II: SUBJECT SYSTEMS

Systems	Type	Size (LOC)	Evolution Period	#Revision
JabRef	Bibliography Manager	153952	OCT 2003 - NOV 2011	3718
DNSJava	DNS Protocol	20831	SEP 1998 - FEB 2013	1679
OpenYMSG	Open Messenger	8821	MAR 2007 - MAR 2013	233
Ant-Contrib	Web Server	79434	JUL 2006 - MAR 2009	177
Carol	Driver Application	13213	JUN 2002 - SEP 2013	2237
ArgoUML	UML Modeling Tool	157573	JAN 1998 - JUN 2014	19915
Squirrel	SQL Client	332635	JUN 2001 - JAN 2013	6737

We extract the file modification history using *SVN diff* to list added, modified and deleted files in successive revisions. This information is used to exclude the unchanged files during change analysis to speed up the process.

2) *Method Extraction and Origin Analysis*: To analyze the changes to methods throughout all the revisions, we extract method information from the successive revisions of the source code. We store the method information in a database to use for mapping changes to the corresponding methods. Again, SVN keeps track of the files that are added, deleted or modified and the history of changes to individual file content are preserved as modification of lines. This line-level change information is not sufficient to describe the evolution of source code entities at higher granularity levels such as classes or methods. As a result, to map changes to methods throughout the development cycle, we need to map the methods across the revisions. Therefore, we carry out origin analysis [5] of the methods on the revisions of the systems to gather mapping information for methods and preserve in a database. This information is used to map the classified changes and cloning information back to the corresponding methods to measure the frequency of changes.

3) *Change Extraction and Classification*: The change analysis system in this study is implemented in Java based on the change extraction and classification core of *ChangeDistiller* [15]. The system imports copies of changed files from successive versions and uses JDT API of Eclipse for the extraction of methods and extracting the differences between the copies of each of the files in any two successive revisions. The details

of change extraction and classification are as follows:

- *Change Extraction*: Code changes are extracted using *ChangeDistiller* classifier. *ChangeDistiller* extracts changes by taking differences between two versions of ASTs of the same file and are stored as a sequence of tree-edit operations. The generic operations contain insert, delete, move and update operations on the nodes in the AST. The tree-edit operations encoded as edit scripts are then processed by *ChangeDistiller* to classify extracted changes to fine-grained change types. We have customized the *ChangeDistiller* classifier to suit for analyzing local repository exported from SVN. To extract source code changes, two successive versions of the same file are selected from the source repository and then are passed to the differencing engine of the change classifier. The extracted changes are then passed to the classifier for classification. The process is repeated for all changed files (identified by *SVN diff*) and for all the revisions of the subject systems.
- *Change Classification*: Changes extracted by AST-differencing of two successive revisions of source code files are classified into fine-grained changes to source code entities. Changes are classified according to the defined taxonomy and are assigned the corresponding levels of significance. For the analysis, classified changes are mapped to the corresponding source code entities based on the information extracted during the origin analysis.

TABLE III: NiCad SETTINGS FOR THE STUDY

Parameters	Values
Minimum Size	5 lines
Maximum Size	500 lines
Granularity	Method Level
Threshold	0% (Type-1, Type-2), 30% (Type-3)
Identifier Renaming	blindrename (Type-2, Type-3)

4) *Mapping Change Data*: After classification, the classified changes are mapped to their corresponding source code entities (methods) with the help of extracted origin mapping information for the associated entities. We preserve the extracted, classified and mapped changes into a database to measure metrics for frequency of changes at the method level granularity.

5) *Clone Detection*: For this study we use the hybrid clone detection tool NiCad [16]. NiCad is reported to have higher level of precision and recall [18] and supports the detection of Type-1 (exact copies), Type-2 (syntactically exact with identifier naming differences) and, Type-3 clones (in addition to Type-2 differences lines are added, deleted or modified) clones. We run NiCad on all revisions of the subject systems to detect clones at method level granularity. The clone detection results are then processed to store the clone information in the database. Table III lists the parameter settings for NiCad used for this study.

6) *Measurement of Metrics and Comparison*: We calculate the defined metrics from the extracted changes and clone data stored in the database using equations (1) and (2). For all the subject systems, we measure the frequency metrics for all types of clones (Type-1, Type-2 and Type-3) and for all the four levels of significance of changes. Higher values of frequency metrics refer to lower stability of the corresponding subject system. The metrics are then analyzed for the comparative evaluation of the impacts of clones on maintenance in terms of the frequency of changes to cloned and non-cloned code.

V. RESULTS AND ANALYSIS

This section represents the results of our empirical study by answering three research questions we defined in Section I. We evaluate the stability of clones by analyzing the values of the metrics for the frequency of changes in cloned and non-cloned code. We calculated the values of the metrics for all seven subject systems, for each of the three clone types (Type-1, Type-2 and Type-3) and for each of the four levels of significance (*low*, *medium*, *high* and *crucial*) of changes.

Table IV represents the comparative frequencies of changes in cloned and non-cloned code for Type-1 clones in the seven subject systems considered. Values for the metrics calculated for the frequency of changes in cloned and non-cloned code considering each of the levels of change significance are represented in the corresponding columns in the table. Similarly, Table V and Table VI represent the comparative frequencies of changes in cloned and non-cloned code for Type-2 and Type-3 clones respectively for all seven subject systems.

Table VII represents the summary of the stability scenarios of cloned and non-cloned code. This table is derived from Table IV, Table V, and Table VI. The symbol '⊗' in Table VII

TABLE IV: COMPARATIVE FREQUENCIES OF CHANGES OF DIFFERENT LEVELS OF SIGNIFICANCE FOR TYPE-1 CLONES.

Change Significance→	Low		Medium		High		Crucial	
	CF_c	CF_n	CF_c	CF_n	CF_c	CF_n	CF_c	CF_n
Systems ↓								
OpenYmsg	197.16	6.34	96.74	4.86	2.95	0.15	0.74	0.28
DNSJava	203.60	25.47	183.13	25.15	10.98	3.07	8.26	2.46
JabRef	36.78	12.59	18.78	10.09	0.06	0.14	0.12	0.74
Carol	10.73	14.47	14.44	11.13	0.80	0.64	1.20	0.81
Ant-Contrib	1.13	0.30	0.93	0.21	0.02	0.02	0.01	0.01
ArgoUML	130.74	25.91	81.13	16.78	0.96	0.61	1.80	1.13
Squirrel	14.30	6.94	10.78	4.24	0.30	0.26	0.54	0.49

CF_c =Frequency of Changes in Cloned Code

CF_n =Frequency of Changes in Non-cloned Code

TABLE V: COMPARATIVE FREQUENCIES OF CHANGES OF DIFFERENT LEVELS OF SIGNIFICANCE FOR TYPE-2 CLONES.

Change Significance→	Low		Medium		High		Crucial	
	CF_c	CF_n	CF_c	CF_n	CF_c	CF_n	CF_c	CF_n
Systems ↓								
OpenYmsg	16.64	7.30	12.74	5.28	0.26	0.17	0.26	0.28
DNSJava	44.44	43.32	41.33	41.00	5.61	3.82	2.15	3.07
JabRef	19.05	15.92	11.04	11.31	0.06	0.13	1.66	0.63
Carol	45.39	13.13	36.59	10.16	0.66	0.64	0.42	0.84
Ant-Contrib	0.37	0.71	0.47	0.55	0.00	0.02	0.00	0.01
ArgoUML	68.23	26.42	47.00	16.87	0.49	0.63	2.84	1.06
Squirrel	10.28	7.47	7.25	4.70	0.16	0.27	0.90	0.47

CF_c =Frequency of Changes in Cloned Code

CF_n =Frequency of Changes in Non-cloned Code

indicates that the frequency of changes in cloned code is higher than the frequency of changes in non-cloned code ($CF_c > CF_n$). This implies that cloned code is less stable as compared to non-cloned code for the corresponding subject system, type of clones and the level of significance of changes. A '⊗' symbol on the other hand refers to a case where the frequency of changes in cloned code is less than the frequency of changes in non-cloned code ($CF_c < CF_n$). This indicates that cloned code is more stable as compared to non-cloned code for the corresponding subject system, type of clones and the level of significance of changes. This table represents the 84 (7x3x4) decisions points regarding comparative stabilities of cloned and non-cloned code for seven subject systems, three types of clones and four different levels of the significance of changes. Based on the analysis of these decision points we answer the research questions in the following sections.

TABLE VI: COMPARATIVE FREQUENCIES OF CHANGES OF DIFFERENT LEVELS OF SIGNIFICANCE FOR TYPE-3 CLONES.

Change Significance→	Low		Medium		High		Crucial	
	CF_c	CF_n	CF_c	CF_n	CF_c	CF_n	CF_c	CF_n
Systems ↓								
OpenYmsg	10.39	7.19	9.17	5.07	0.00	0.18	0.00	0.30
DNSJava	52.03	42.48	48.38	40.27	10.26	3.23	2.73	3.07
JabRef	36.16	14.31	21.96	10.42	0.17	0.13	1.40	0.59
Carol	33.24	11.70	29.31	8.60	0.47	0.67	0.97	0.80
Ant-Contrib	2.43	0.35	2.31	0.21	0.03	0.02	0.01	0.01
ArgoUML	79.70	20.56	53.34	12.95	0.42	0.65	1.77	1.05
Squirrel	12.08	6.44	8.07	4.00	0.21	0.28	0.62	0.47

CF_c =Frequency of Changes in Cloned Code

CF_n =Frequency of Changes in Non-cloned Code

TABLE VII: COMPARATIVE STABILITY OF CLONED AND NON-CLONED CODE

Change Significance →	Low			Medium			High			Crucial		
	Systems ↓	Type-1	Type-2	Type-3	Type-1	Type-2	Type-3	Type-1	Type-2	Type-3	Type-1	Type-2
OpenYmsg	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊖	⊗	⊖	⊖
DNSJava	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊖	⊖
JabRef	⊗	⊗	⊗	⊗	⊖	⊗	⊖	⊖	⊗	⊖	⊗	⊗
Carol	⊖	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊖	⊗	⊖	⊗
Ant-Contrib	⊗	⊖	⊗	⊗	⊖	⊗	⊖	⊖	⊗	⊖	⊖	⊖
ArgoUML	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊖	⊖	⊗	⊗	⊗
Squirrel	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊖	⊖	⊗	⊗	⊗

⊗=cases where frequency of changes in cloned code is higher than that in non-cloned code ($CF_c > CF_n$)

⊖=cases where frequency of changes in cloned code is less than that in non-cloned code ($CF_c < CF_n$)

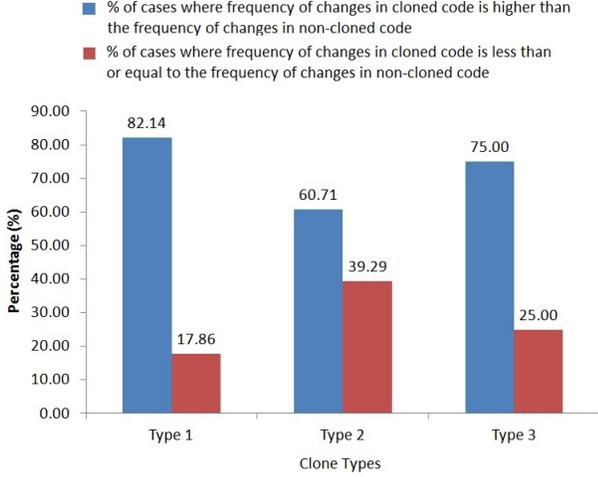


Fig. 3: Comparison of change frequency of cloned and non-cloned code based on clone types

A. Answer to RQ1: Analysis from the perspective of clone types

In this analysis, we evaluate the comparative stability scenarios of the subject systems with respect to three different types of clones and we try to answer the first research question (RQ1). From Table VII, we have 28 (7 systems x 4 levels of change significance) decision points for each of the three types of clones in columns labeled with corresponding clone types (Type-1, Type-2 and Type-3). Based on the summary in Table VII, we represent the comparative stabilities of cloned and non-cloned code from the perspective of the types of clones in Figure 3. Now, for Type 1 clones there are 82.14% (23/28) of cases (marked with '⊗' in Table VII) where the frequency of changes in cloned code is higher than the frequency changes in non-cloned code ($CF_c > CF_n$). For the remaining 17.86% (5/28) cases (marked with '⊖' in Table VII) the frequency of changes in cloned code is less than that of non-cloned code ($CF_c < CF_n$).

Again, for Type-2 clones there are 60.71% (17/28) cases where the frequency of changes in cloned code is higher than the frequency changes in non-cloned code. For Type-3 clones, on the other hand, 75% (21/28) cases show that the frequency of changes in cloned code is higher than that of non-cloned code. Here, we observe that for all clone types (Type-1, Type-2 and Type-3) the frequency of changes in cloned code is higher than the frequency of changes in non-cloned code suggesting that cloned code is modified more frequently than non-cloned code. This is an indication that cloned code is less stable as compared to non-cloned code.

To evaluate how statistically significant the comparative frequencies of cloned and non-cloned code are with respect to different types of clones, we carry out Mann-Whitney Wilcoxon (MWW) test [19]. We consider the corresponding frequencies of changes for cloned and non-cloned code for a particular clone type and for all the systems. We observe that for Type-1 clones the p value for *low* and *medium* significance is 0.03102 (< 0.05 , two-tailed test) which implies that "the difference between the two samples is marginally significant". However, for the *high* and *crucial* levels of significance of changes for Type-1 clones and for all levels of significance for Type-2 and Type-3 clones p values are > 0.05 indicating that values for frequencies for cloned and non-cloned code are not significantly different. However, we observe that the frequency of changes to cloned code is higher than that of non-cloned code in higher percentages of cases (Figure 3).

Summary- According to our findings from the perspective of the types of clones, the *stability* of non-cloned code is higher than that of cloned code meaning that cloned code poses higher maintenance challenges than non-cloned code during maintenance and evolution. Also, Type-1 clones are the most vulnerable to the stability of the software systems.

B. Answer to RQ2: Analysis from the perspective of change significance

Here, we analyze the comparative frequencies of changes to cloned and non-cloned from the perspective of the four different levels (*low*, *medium*, *high*, *crucial*) of significance of changes. For each of the four levels of significance, we have 21 decision points (7 systems x 3 clone types) in Table VII in the column groups labeled with the corresponding levels of significance to answer the second research question (RQ2). The comparative stability scenarios with respect to the levels of change significance are presented in Figure 4.

As shown in Figure 4, 90.48% (19/21) of cases cloned code has higher frequency of changes than the frequency of changes in non-cloned code in both *low* and *medium* levels of significance. For more significant (*high*, *crucial*) changes, the percentages of cases where the frequencies of changes is higher in cloned code than in non-cloned code is comparatively closer. However, for both *high* and *crucial* levels of significance the frequencies of changes in cloned code is higher than that of non-cloned code. In Figure 4, we observe that cloned code is less stable than non-cloned code and this is mostly dominated by the changes of *low* to *medium* levels of significance.

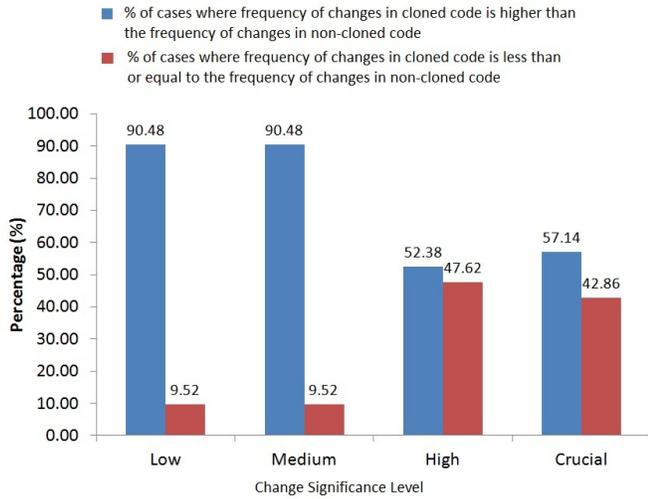


Fig. 4: Comparison of change frequency of cloned and non-cloned code based on different levels of significance of changes

To evaluate whether there are statistically significant differences among the values for the metrics we obtained with respect to different levels of significance of changes, we carry out Mann-Whitney Wilcoxon (MWW) test [19]. We consider the corresponding frequencies of changes for cloned and non-cloned code for a particular level of significance of change and for all the systems and all types of clones. We observe that for changes with *low* and *medium* significance the p values are 0.0207 and 0.0209 respectively and both of the values are <0.05 (two-tailed test) which imply "the difference between the two samples is marginally significant". However, for the high and crucial significance levels p values are >0.05 indicating that values for frequencies for cloned and non-cloned code are not significantly different. So, we see that higher frequency of changes in cloned code as compared to non-cloned code is mostly influenced by changes of lower significance levels (*low*, *medium*) whereas for changes of higher significance levels (*high*, *crucial*) the differences among the frequencies are not statistically significant. However, from the perspective of change significance we observe the frequency of changes to cloned code is higher than that of non-cloned code in higher percentages of cases.

Summary- Our findings from the analysis regarding the perspective of the different levels of significance of changes suggest that cloned code is less stable than non-cloned code and this stability is influenced mostly by the changes of *low* to *medium* significance.

C. Answer to RQ3: Analysis from the perspective of systems

We carry out analysis from the perspective of individual systems to answer the third research question (RQ3). We consider the 12 (3 clone types x 4 levels of significance) decision points for each of the subject systems from the corresponding rows in Table VII. The system centric comparative stability scenarios for cloned and non-cloned code is represented in Figure 5.

As shown in Figure 5, our study shows that for six out of seven subject systems the frequency of changes in cloned code

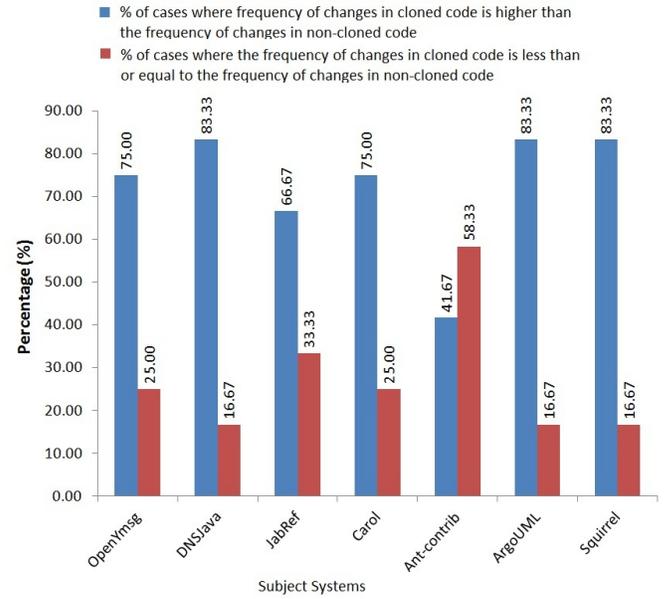


Fig. 5: Comparison of change frequency of cloned and non-cloned code from the perspective of the systems

is higher than that of non-cloned code considering all types of clones and all the four levels of the significance of changes. Also, we observe that for larger systems with comparatively large number of revisions and with longer period of evolution (Jabref, Argouml and Squirrel in Table II) tend to have higher percentages of decision points supporting the higher frequency of changes in cloned code as compared to non-cloned code. This suggests that clones in larger systems might be more change prone and comparatively more vulnerable to the stability of the system.

One important point is that for the system Openmsg, our findings agree with the findings from Hotta *et al.* [7] but our findings disagree with their findings for the system Squirrel. This disagreement might be due the differences in how we count changes. Hotta *et al.* count changes in consecutive lines of code as a single change whereas we count individual fine-grained changes to program entities. Considering changes at finer granularity level is highly likely to increase the change count and thus the increase in the values of the change frequency. For comparatively smaller systems with smaller number of revisions like Openmsg, this difference might not be significant. However, for larger systems with higher number of revisions like Squirrel, changes at finer and coarser granularity may make significant differences in the metrics values calculated which might have affected in having differences in the results. In addition, the relative proportion of cloned and non-cloned code in the software systems might also have influence on the measurement of stability metrics. Moreover, we consider individual levels of significance of changes and clone types which might have effects on decisions regarding comparative stability of cloned and non-cloned code.

Summary- Our system centric analysis suggests that cloned code is less stable in general and software systems with comparatively larger code base and higher number of revisions tend to have higher probability of having higher frequency of changes in cloned code as compared to non-cloned code.

This study re-investigates the comparative stability of cloned and non-cloned code. Stability is a widely focused measure to evaluate the impacts of clones on software maintenance. But our study quantifies stability from a new perspective considering the fine-grained types of changes and their levels of significance. From the study we conclude that although the magnitude of stability may vary with different types of clones and the changes of different significance levels, cloned code generally tend to have less stability. In addition, Type-1 clones are more vulnerable to the stability of the system and thus, need more attention during the evolution.

We also observe that the stability of cloned code is influenced mostly by the changes of lower (*low, medium*) levels of significance. However, for changes with higher (*high, crucial*) levels of significance the comparative frequency of changes in cloned code is still higher than that of non-cloned code. Although our fine-grained analysis disagrees with the stability decisions of some of the related existing studies [7], [8], [12], our methodology gives us the confidence that our stability decisions are more precise and reliable as those take into account the actual impacts of changes. Again, the way we analyze the stability gives us an insight into the detail evolution of how individual clones change throughout their period of evolution. The information regarding the frequency of changes with distinct levels of significance might be valuable in ranking and prioritizing clones by the developers during clone management. The evolution information might also be used as constraints for automatic scheduler for clone refactoring [20].

VI. THREATS TO VALIDITY

The change analyzer in this study is based on the change extraction and classification engine of the *ChangeDistiller*. Although *ChangeDistiller* is reported to have good performance, the validity of the outcomes of the study is dependent upon the accuracy of the core classifier used.

Another source of potential threats is the clone detection tool used. We used NiCad, a recently introduced hybrid clone detection tool which detects both exact (Type-1) and near-miss (Type-2, and Type-3) clones with high precision and recall. Again, different settings for clone detection tools might result in different stability scenarios because of the variations in clone detection results. This is termed as *confounding configuration choice problem* [21]. However, the NiCad settings we used are considered standard [1], [22] and are close enough to the optimal configuration settings identified in recent study [21] for NiCad to detect clones in Java systems. This is likely to mitigate the potential adverse effects of the configuration settings on our findings.

We selected subject systems with diversified size, number of revisions, length of evolution and application domain to avoid potential biasing. However, due to the limitation of existing change classifier our study is limited to Java systems only. Although Java is considered to be a widely used language with a comprehensive set of language features for software development, inclusion of subject systems of other languages might help in more generalization of the findings.

There are large body of research work investigating the impacts of clones on software maintenance and evolution. The primary focus of these studies are evaluating the impacts of clones in terms of the degree of consistency in comparative evolution, measuring stability to study how frequently a code region is modified and the likelihood of introducing bugs during evolution.

Krinke [8] proposed a measure for comparative stability of cloned and non-cloned code. His approach counts the modification of code in terms of the number of lines added, deleted and modified in cloned and non-cloned code. The more the number of changes to a code region, the less stable the code is. His study concluded that clone code is more stable than non-cloned code. Krinke's other study [9] measures the average age of the cloned and non-cloned code. This study concludes that clone code is older meaning more stable than non-cloned code. Göde and Harder [12] extended Krinke's [8] approach with token-based incremental clone detection tool and experimented with different settings of clone detection parameters, clone types and their changes. This study agrees with the findings of Krinke [8] that cloned code is more stable than non-cloned code but this does not hold in case of deletion.

Hotta *et al.* [7], on the other hand, measure the stability of cloned and non-cloned code in terms of *modification frequency*. Their study concludes that cloned code has lower modification frequency and thus more stable than non-cloned code. Their approach counts the number of blocks (consecutive lines) modified in cloned and non-cloned code. The average modification count per revision in cloned and non-cloned code are then used to measure the modification frequency. The less the modification frequency, the more stable the code region is. One of the key limitations of this approach is that it ignores the volume of change and also the types of actual syntactic changes. This is likely to affect the accuracy and reliability of the stability of the code measured. Although this study considers fine-grained changes because of the change analysis at token level, it does not differentiate between the types of syntactic changes and their levels of significance.

Mondal *et al.* [13], [14] carried out a comprehensive stability analysis within a uniform evaluation framework considering existing (taken from [7], [8], [9]) and their proposed metrics to measure stability. Their study concluded that there is no firm conclusion on the stability of cloned and non-cloned code, rather, the comparative stability varies with programming languages, clone types and overall system development strategies. However, none of the metrics used or proposed in this study consider actual syntactic change types.

Lozano *et al.* [23] investigated whether clones are harmful or not. Their study concludes that cloned code tend to be more frequently modified than non-cloned code. To assess the impacts of clones on software maintenance Lozano and Wermelinger [5] also investigated the impacts of clones on software maintenance. Their study shows that although the *likelihood* (the ratio of the number of change to an entity to the total number of changes in the system) in cloned code is not very different than in non-cloned code, in some cases the *impact* (percentage of the system affected by a change) of cloned methods is greater than

that of non-cloned methods. Their study also suggests that the presence of clones may decrease the changeability of the code entity containing clones.

Our proposed study to measure the comparative stability of cloned and non-cloned code is kind of related to the study proposed by Hotta *et al.* [7]. However, our study is different from their study because we count each of the fine-grained changes whereas Hotta *et al.* consider a range of consecutive lines of modified code as a single change. Another difference is that we consider the different significance levels of changes and we measure the frequency of changes based on the levels of significance of the changes. Our study differs from Krinke's study [8] because we consider every revision of the subject systems whereas Krinke's study considers revisions on weekly intervals. Again, Krinke's study [8] differentiates between the change types as the addition deletion and update to lines of code but does not consider the actual syntactic change types as in our study. Although there are some similarities, our study is different from all others in the sense that we study the stability in the fine-grained levels considering the different change types and their levels of significance. Our objective is to focus on how cloned and non-cloned code are evolving through different changes and the impact of clones on software maintenance in terms of the frequency of changes of different levels of significance, which none of the existing studies above consider.

VIII. CONCLUSION AND FUTURE WORK

This study measures the comparative frequency of fine-grained code changes to cloned and non-cloned code considering the levels of significance of changes. To the best of our knowledge, this is the first ever approach to the comparative evaluation of the stability of cloned and non-cloned code that considers actual syntactic change types and their corresponding levels of significance.

We use the hybrid clone detection tool NiCad to detect exact and near-miss clones. By analyzing the changes to the cloned and non-cloned code of seven software systems, we observe that the frequency of changes is higher in cloned code than that in non-cloned code in most cases. Again, from the perspective of clone types our study shows that in most of the cases the frequency of changes in cloned code is higher than that in non-cloned code and Type-1 clones are more change prone affecting the stability of the software systems. Another, important point is that the comparatively higher instability of clones is mostly influenced by the changes of lower (*low, medium*) significance. For changes of higher significance (*high, crucial*) the stability of cloned and non-cloned code are closer unlike the stabilities for changes of lower significance. However, for all levels of significance of changes the stability of non-cloned code is higher than the cloned code. Thus, cloned code is likely to pose higher challenges during the evolution and maintenance of the software systems.

Moreover, in this study, our analysis at the fine-grained change types considering their levels of significance facilitates to have a deeper insight into how the clones evolve with changes of different levels of significance. The history regarding the evolution of the clones representing how individual clones are changed over time might be an important source of information to rank the clones based on the types and volume of changes they evolve through. This might help in better managing clones

by tracking and/or refactoring clones to minimize the negative impacts of clones. We plan to extend our study to a larger number of large-scale projects from diversified application domain and development history to have a more generalized conclusions on the stability of clones.

REFERENCES

- [1] C. K. Roy and J. R. Cordy, "Near-miss function clones in open source software: An empirical study," *Journal of Soft. Maintenance and Evolution: Research and Practice*, vol. 22, no. 3, pp. 165–189, 2010.
- [2] M. Rieger, S. Ducasse, and M. Lanza, "Insights into system-wide code duplication," in *Proc. WCRE*, 2004, pp. 100 – 109.
- [3] C. Roy, M. Zibran, and R. Koschke, "The vision of software clone management: Past, present, and future (keynote paper)," in *Proc. CSMR-WCRE*, 2014, pp. 18–33.
- [4] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *Proc. ICSE*, 2009, pp. 485–495.
- [5] A. Lozano and M. Wermelinger, "Assessing the effect of clones on changeability," in *Proc. ICSM*, 2008, pp. 227–236.
- [6] L. Aversano, L. Cerulo, and M. Di Penta, "How clones are maintained: An empirical study," in *Proc. CSMR*, 2007, pp. 81–90.
- [7] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto, "Is duplicate code more frequently modified than non-duplicate code in software evolution?: An empirical study on open source software," in *Proc. IWSPSE*, 2010, pp. 73–82.
- [8] J. Krinke, "Is cloned code more stable than non-cloned code?" in *Proc. SCAM*, 2008, pp. 57–66.
- [9] —, "Is cloned code older than non-cloned code?" in *Proc. IWSC*, 2011, pp. 28–33.
- [10] C. Kapsner and M. W. Godfrey, "'Cloning considered harmful' considered harmful: Patterns of cloning in software," *Empirical Soft. Engg.*, vol. 13, no. 6, pp. 645–692, 2008.
- [11] A. Lozano and M. Wermelinger, "Tracking clones' imprint," in *Proc. IWSC*, 2010, pp. 65–72.
- [12] N. Go'de and J. Harder, "Clone stability," in *Proc. CSMR*, 2011, pp. 65–74.
- [13] M. Mondal, C. K. Roy, and K. A. Schneider, "An empirical study on clone stability," *SIGAPP App. Comp. Rev.*, vol. 12, no. 3, pp. 20–36, 2012.
- [14] M. Mondal, C. K. Roy, M. S. Rahman, R. K. Saha, J. Krinke, and K. A. Schneider, "Comparative stability of cloned and non-cloned code: An empirical study," in *Proc. ACM SAC*, 2012, pp. 1227–1234.
- [15] B. Fluri and H. C. Gall, "Classifying change types for qualifying change couplings," in *Proc. ICPC*, 2006, pp. 35–45.
- [16] C. Roy and J. Cordy, "NiCad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *In Proc. ICPC*, 2008, pp. 172–181.
- [17] J. R. Cordy and C. K. Roy, "The NiCad clone detector," in *Proc. ICPC*, 2011, pp. 219–220.
- [18] C. Roy and J. Cordy, "A mutation/injection-based automatic framework for evaluating code clone detection tools," in *Proc. ICSTW*, 2009, pp. 157–166.
- [19] Mann-Whitney Wilcoxon (MWW) Test. Accessed: 2014-06-30. [Online]. Available: <http://elegans.som.vcu.edu/~leon/stats/utest.html>
- [20] M. Zibran and C. Roy, "A constraint programming approach to conflict-aware optimal scheduling of prioritized code clone refactoring," in *Proc. SCAM*, 2011, pp. 105–114.
- [21] T. Wang, M. Harman, Y. Jia, and J. Krinke, "Searching for better configurations: A rigorous approach to clone evaluation," in *Proc. FSE*, 2013, pp. 455–465.
- [22] R. Saha, C. Roy, and K. Schneider, "An automatic framework for extracting and classifying near-miss clone genealogies," in *Proc. ICSM*, 2011, pp. 293–302.
- [23] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Evaluating the harmfulness of cloning: A change based experiment," in *Proc. MSR*, 2007, pp. 18–21.