Electronic Communications of the EASST
Volume 63 (2014)

Proceedings of the Eighth International Workshop on Software Clones (IWSC 2014)

Active Clones: Source Code Clones at Runtime

Mohammad Asif A. Khan, Chanchal K. Roy and Kevin A. Schneider

18 pages

Guest Editors: Nils Göde, Yoshiki Higo, Rainer Koschke
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Active Clones: Source Code Clones at Runtime

Mohammad Asif A. Khan[1], Chanchal K. Roy[2] and Kevin A. Schneider[3]

[1] m.khan@usask.ca, http://mdakhan.weebly.com/index.html
[2] croy@cs.usask.ca, http://www.cs.usask.ca/~croy/
[3] kevin.schneider@usask.ca, http://www.cs.usask.ca/~kas/Welcome.html
University of Saskatchewan, Canada

Abstract: Code cloning is a common programming practice, and there have been a considerable amount of research that investigated the implications of code clones on software maintenance using static analysis. However, little has been done to investigate the runtime implications of code cloning. In this paper we investigate source code clones at runtime, referring to clones as 'active clones' if they are invoked when a software system is in use. For example, if a particular use u of a system results in a clone c being invoked, we say that clone c is active with respect to use u. From this definition and given a set of uses $\{u_1, u_2, ...\}$ and clones $\{c_1, c_2, ...\}$ we are able to identify the extent clones are active at runtime and analyze active clone resource use (e.g., CPU time) and define and calculate a set of active clone metrics to provide insights into source code cloning implications at runtime. We developed a hybrid static and dynamic analysis technique for detecting and analysing active clones, and conducted an empirical study on five software systems (HSQLDB, JHotDraw, RText, jEdit and UniCentaoPOS) to validate our approach. We found a small portion of clones are active during a typical use of a software system, and that active clones have the potential for guiding a software developer's code inspection activity during a software maintenance task.

Keywords: clone detection, active clones, dynamic analysis

1 Introduction

Code duplication is a common programming practice. Programmers use code duplication to increase the speed of the software development process by using similar code to implement common functionality. Although cloning is an inexpensive and often productive practice, it introduces complications during software maintenance [JDHW09, KG08]. A code fragment may contain a bug, and when copied elsewhere, the bug is also propagated. As well, a bug may emerge when a clone is used in a new context. Fixing a bug in one clone may require reviewing and perhaps applying a similar fix to its clones. Any inconsistency in making a change to a set of clones may introduce new bugs. For large software systems and depending on the extent of the cloning, such maintenance effort may be expensive. Despite the ongoing debate as to whether code clones are harmful or not [JDHW09, KG08, LW08, Roy09, RC07], it is believed that clones are unavoidable, and not all clones can or should be removed [KG08, KSNM05], which calls for software clone management.

Clone management is an active reverse engineering research area and there have been several attempts to aid in the management and maintenance of software [XXJ11, YCY+13, HKKI05,

TG12, MLPB13]. In this paper, we also aim to aid towards clone management. However, unlike the previous approaches, we exploit the runtime implications of source code cloning for clone management. In particular, we investigate source code clones at runtime, referring to clones as 'active clones' if they are invoked when a software system is in use. For example, if a particular use u of a system results in a clone c being invoked, we say that clone c is active with respect to use u. From this definition and given a set of uses $\{u_1, u_2, ...\}$ and clones $\{c_1, c_2, ...\}$ we are able to identify the extent clones are active at runtime, analyze active clone resource use (e.g., CPU time) and define and calculate a set of active clone metrics. Associated with a particular use u of a system, is an execution trace, which we denote as t_u. An execution trace t_u is a sequence of method calls $\{< 1, m_1 >, < 2, m_2 >, ...\}$ and so $codomain(t_u)$ is the set of method calls invoked during the use (i.e., $codomain(t_u) = \{m_1, m_2, ...\}$). Our focus is on method clones, and so if method clone $c \in codomain(t_u)$, then c is an active clone with respect to use u. Execution traces may be collected in the field, however for our study we used a set of tests to collect execution traces. We consider a test to represent a typical use of a software system by an end-user, or a typical use of a software system by a developer in support of a software development or maintenance task. Software developers often use testing to execute a software system with the intent of locating errors, detecting faults and verifying or validating a system's qualities and functionality. If a developer runs the system or designs a test suite for the task at hand, they can use our approach to identify active clones related to the tests and consequently to the task at hand.

It is clear that the choice of tests effects the extent active clones are discovered. For example, a test suite may be specifically designed for full coverage and so in such a situation one would expect all clones to be active. Our interest, however, is in tests that correspond to a particular use of the system. For example, tests that are specifically designed to exercise a single feature, requirement or use case scenario. Although a test suite may exercise the entire system, a single test typically is designed to exercise a subset of the system. A clone, then, is active with respect to a use, a set of uses, a set of tests, or a test suite. It is determining the method clones that are active when a single test or a small set of tests are run that we focus on in this paper. As a result, we are interested in addressing the following three research questions:

RQ1: *To what extent are clones active during runtime?*

RQ2: *How active are the active clones?*

RQ3: *Does active clone identification support software maintenance activities?*

To address these questions we developed a hybrid dynamic and static analysis technique for detecting active clones and conducted an empirical study of five open source software systems downloaded from Sourceforge [Sou]. Our study provides insights into the software systems by showing detailed information concerning active clones, their coverage in the codebase, the frequency of their use and resource utilization.

The remainder of the paper is organized as follows. Section 2 explains the testing framework. Section 3 describes the active clone detection framework. Section 4 presents the defined metrics. Section 5 describes the study approach while Section 6 presents the findings. Threats to validity and related work is discussed in Section 7 and Section 8 with conclusion and future work in Section 9.

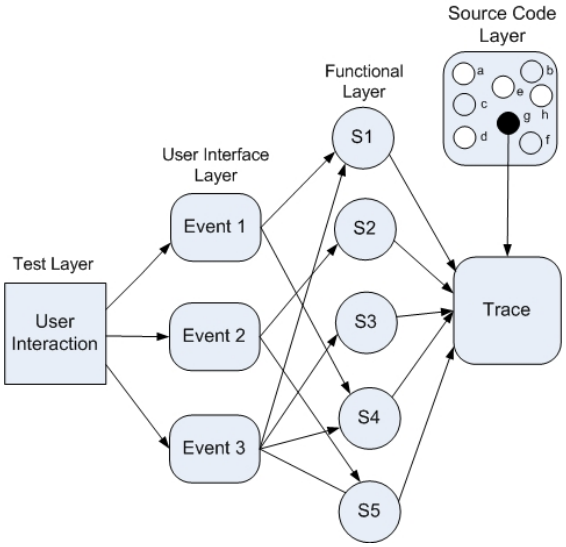

Figure 1: Testing Framework

2 Testing Framework

Each of the systems in our study are interactive systems with graphical user interfaces (GUIs), and so we developed a testing framework Figure 1 that accommodates tests corresponding to a user interacting with a system. Four layers are identified in the framework: the test layer, the user interface layer, the functional layer and the source code layer. The framework connects user activities defined in test suites and tests to the source code. In a drawing application, creating a rectangle, circle, or triangle may be three individual tests in a single drawing test suite. Mouse actions, keyboard actions, display updates and so on are categorized as user interface (UI) events. Such UI events are provided by the software user interface and are denoted in Figure 1 as *Event i* where $i = 1, 2, .., n$. Each individual event in turn corresponds to sub-programs that handle the event, denoted in Figure 1 as *Si*. A sub-program may be executed by more than one tests.

A trace is a sequence of method invocations executed during a particular test. The methods are invoked in response to the user interaction. An execution trace can be further processed to mine meaningful resource information, such as CPU and memory usage. We are also able to segment the execution trace by user interface event to help manage the size and complexity of the trace data.

In our study we consider only method clones. If a method in an execution trace is a clone, then the clone is an active clone. A clone class that includes at least one active clone is referred to as an active clone class. For example, in Figure 1, there are three clone classes: $CC_1 = \{a, b, c\}$; $CC_2 = \{d, e\}$; and, $CC_3 = \{f, g, h\}$. Let us assume that using our approach, we found that only g is an active clone (i.e., the corresponding method was invoked in the trace). Because $g \in CC_3$, clone class CC_3 with clones $\{f, g, h\}$ will be the only active clone class. For a particular clone related maintenance task, this knowledge may help guide a developer's focus.

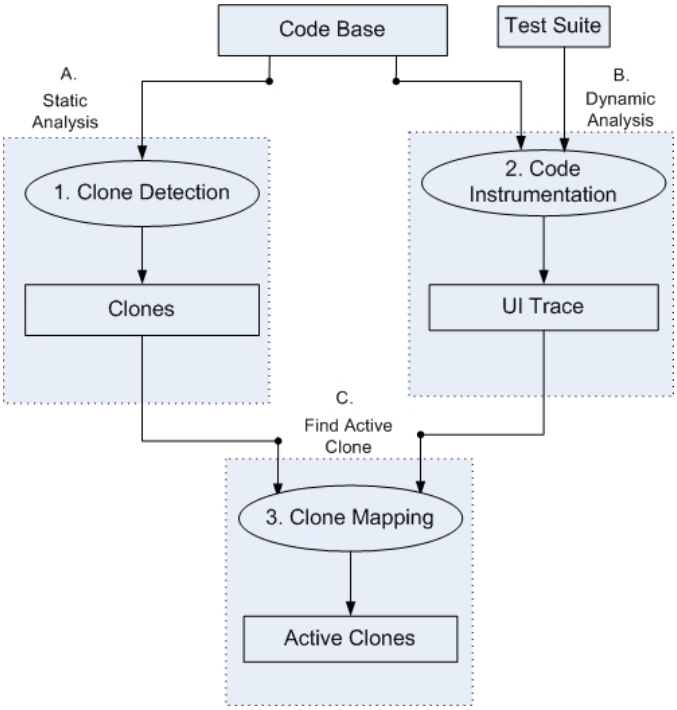

Figure 2: Active Clone Detection Framework

3 Active Clone Detection Framework

Our overall goal is to determine whether software maintenance could be more efficient or effective by identifying the active clones and active clone classes that correspond to a maintenance task by using runtime information. The maintenance effort of a system containing large amounts of code clones may be reduced if test suites specific to the maintenance task can be used to identify just those clones that are relevant.

We developed a hybrid approach, incorporating both static and dynamic analysis techniques. We use dynamic analysis (using aspect-oriented programming to instrument the software) to capture execution trace information and static analysis to capture clone information. To conduct a study a series of steps is required, from obtaining the software to mapping clones to execution traces. For this we designed and developed a framework for the experiment, which is depicted in Figure 2. The framework consists of three major phases: a) static analysis, b) dynamic analysis, and c) finding active clones.

3.1 Static Analysis

Static analysis is useful for mining specific information from a codebase without executing the software and is commonly used for detecting clones. An advantage of using a static technique is that we can often obtain information from the source code even if there is incomplete code. Furthermore, it is neither necessary to run the software nor does the software need to be executable.

We use a static analysis tool, the NiCad clone detector [CR11, RC08], to detect clones in the codebase. NiCad has been shown to be highly accurate with respect to both precision and recall [RC09, RC10] in the detection of copy-and-pasted near-miss clones. We used a dissimilarity threshold of 30% with blind renaming option of identifiers of NiCad.

3.2 Dynamic Analysis

Dynamic analysis [CZD+09] is the technique of analysing the data collected from a running program. The advantage of using dynamic analysis is that it exposes the system's actual behaviour. However, the drawback of it is that only a partial picture of a system is revealed and the obtained information is dependent on the inputs used to exercise the system's functionality. Dynamic analysis is used in a variety of applications such as software testing, performance analysis, and program comprehension via reverse engineering. We are interested in collecting profile information in the form of execution traces. This can be done by instrumenting the code, monitoring the execution environment, or as is the case in our approach, by dynamically weaving aspects to log runtime information. A challenge is to have an adequate set of tests or recorded user interaction to collect relevant behavioural information.

The first part of our dynamic analysis approach is to instrument the code for collecting the execution traces. We use aspect-oriented programming [KLM+97] and dynamic loading to log method calls. This allows us to generate a sequence of execution traces that will later enable us to identify active clones and their behaviour. To capture the execution traces, the system is run using a test suite corresponding to the activity we wish to analyze. By focusing on a single maintenance task, the test corresponds directly to the user interaction that is being investigated. If the entire system and clone execution frequency analysis is to be performed, a comprehensive test suite with adequate coverage is warranted which is out of the scope of this study.

As part of the code instrumentation step of the dynamic analysis phase, we use an AspectJ [KHH+01] implementation to describe and capture the interface actions that occur in a target application. We have written a tracing aspect that weaves with the program and records when any method in the target application is invoked. The advantage of using an AspectJ implementation is that it does not turn off the just-in-time compiler, and enables access to any component (method, parameter, constructor, etc.) in a software system.

To aid in program comprehension we also segmented the execution trace into meaningful segments called user interface (UI) traces [SS09]. An execution contains UI traces for program initialization, display updates, mouse events and so on which are often pertinent to the tasks at hand. This segmentation process allows a developer to associate the cloning concentration with the semantics of code. In this way, they can also learn in which part of the system cloning happens most often (e.g., UI or business logic code).

3.3 Finding Active Clones

The final step maps the trace information to the codebase. For each method in the trace, if it contains a clone in the codebase, the corresponding clone and its clone class are marked as 'active'. This step determines which clone classes are active for that particular test case. First, using dynamic analysis we record execution traces for each individual test case. Each trace

Table 1: Subject Software Systems

Domain	System	No. of Java Files	Total Size	Version
Text Editor	RText	319	5.76M	2.0.0
Graphics	JHotDraw	309	20M	7.0.6
Database Server	HSQLDB	513	14M	2.2.6
Point of Sale	UnicentaoPOS	486	30.4M	2.5.0
Text Editor	jEdit	571	25.5M	4.4.2

record in a trace file consists of the full class name, the class type, the method name, and the parameter types. Second, we obtain all the clones of the subject system detected by the NiCad clone detector (Section 3.1). Third, we determine whether the trace related code contains any clones. For clone mapping, we created a script to extract all the method names from the NiCad XML output file. The clones for which we get a match in the trace are called active clones, and then we retrieve the corresponding clone classes to which the active clones belong. These clone classes are treated as active clone classes.

4 Metrics

To explore the nature of traces and the impact of clones for test based execution of a system we use a number of metrics as defined in this section. We use metrics to quantify aspects of both static and dynamic analyses. Some of the metrics we introduce in this paper (e.g., the active clone metrics) and some are adapted from relevant empirical studies [HL05]. We group the metrics into four major categories: system, clone, trace and active clone metrics.

4.1 System Metrics

- L_{TOTAL}: total number of lines of code in a system.
- F_{TOTAL}: total number of files in a system.
- M_{TOTAL}: total number of methods in a system.

4.2 Clone Metrics

- N_{CLONE}: number of clones in a system.
- CC: number of clone classes in a system.
- $\%M_{CLONE}$: percent of methods that contain a clone.
- F_{CLONE}: number of files that contain a clone.
- $\%F_{CLONE}$: percent of files that contain a clone.

- L_{CLONE}: lines of code that are part of a clone.
- $\%L_{CLONE}$: percent of lines that are part of a clone.

4.3 Trace Metrics

- N_p: number of packages invoked in a trace. When a method from a package is invoked the package is considered to be invoked.
- N_m: number of distinct methods invoked in a trace.
- $\%F_{TRACE}$: Percent of files containing methods invoked in a trace.

4.4 Active Clone Metrics

- N_{ACLONE}: number of active clones.
- CC_{ACLONE}: number of clone classes that contain at least one active clone.
- $\%M_{ACLONE}$: percent of invoked methods that contain at least one active clone.
- F_{ACLONE}: number of files that contain at least one active clone.
- $\%F_{ACLONE} = F_{ACLONE}/F_{TOTAL} * 100$
- L_{ACLONE}: lines that are part of an active clone.
- $\%L_{ACLONE} = L_{ACLONE}/L_{TOTAL} * 100$

5 Study Approach

In this section, we provide a brief description of the subject systems being studied, our approach and the tests used. All the tests were run on an Intel(R) Core (TM) i7 CPU @2.93 GHz with 4GB RAM. Tables 1 and 2 provide a summary of the systems and the tests that were part of our experiment respectively. To bring variation to our study we considered systems from four different domains: text editing, graphics, database management and point of sale. We also selected systems that differed in size, ranging from 5.76M to 30.4M bytes of code.

To help ease the exploration and analysis of active clones and their associated traces, we developed a prototype tool [Kha13] that provides us with trace analysis and management capabilities. The prototype incorporates various features such as filtering, searching and grouping (i.e., categorization), that allows us to investigate the behaviour of active clones.

Table 2: Sample Tests

System	Test	User Interaction
HSQLDB	Select	select * from testtable;
	Insert	insert into testtable values (‘Current’, 22, ‘2003-11-10’, 18, 3, ‘my name goes here’), (‘Popular’, 23, ‘2003-11-10’, 18, 3, ‘my name goes here’), (‘New’, 5, ‘2003-11-10’, 18, 3, ‘my name goes here’), (‘Old’, 5, ‘2003-11-10’, 18, 3, ‘my name goes here’);
	Update	update testtable set (astring, firstnum, adate, secondnum, third-num, aname) = (‘Older’, 5, ‘2003-11-10’, 18, 3, ‘my name goes ’here) where astring = ‘Old’
	Delete	delete from testtable;
	Create table	create cached table testtable (aString varchar(256) not null, first-Num integer not null, aDate date not null, secondNum integer not null, thirdNum integer not null, aName varchar(32) not null);
JHotDraw	Circle	User selects Circle and draws the object.
	Rectangle	User selects Rectangle and draws the object.
	Triangle	User selects Triangle and draws the object.
RText	Edit Text	Write in editor, Save a file, Edit text, Replace text and Find text in files
jEdit	Edit Text	Write in editor, Save a file, Edit text, Replace text, Find text in files, Print File, Close Application
uniCentaoPOS	insert	Adding customer data
	update	updating customer data
	delete	delete customer data
	report	generate customer report

6 Findings

In this section, we present the results of our experiment identifying active clones in five different subject systems of varying sizes. The research questions are discussed based on the experimental results shown in Tables 3 to 8.

Table 3 provides metrics for clones, clone classes, files associated with clones, and lines of code for methods. Table 4 provides statistics for each system in terms of active clones and their classes while Table 6 lists the percentage of active clones for each type of user interface event by test and also provides CPU usage for each type of user interface event along with the percentage of time associated with the active clones. Table 5 shows the percentage of active clones for each test, indicating that clones are involved in each test. It also provides lines of code metrics for each system, comparing the percentage of clone lines with the percentage of active clone lines. This helps illustrate that fewer clones need to be considered when only considering active clones. Table 5 also shows file metrics for each system, comparing the percentage of files containing clones with the percentage of files invoked in the traces and the percentage of files

Table 3: Method Clone Metrics

Systems	Files			Lines of Code			CC	N_{CLONE}	M_{TOTAL}	$\%M_{CLONE}$
	F_{TOTAL}	F_{CLONE}	$\%F_{CLONE}$	L_{TOTAL}	L_{CLONE}	$\%L_{CLONE}$				
RText	319	81	25.3	95036	3019	3	80	189	3698	5.5
JHotDraw	309	183	59.22	56420	7565	13.4	231	656	3260	20.1
HSQLDB	513	293	57.11	227545	20467	8.9	503	1323	9825	13.5
jEdit	571	161	28.2	115246	7320	6.4	212	419	7428	5.6
uniCentaoPOS	486	174	35.8	43032	4578	10.6	163	285	4260	6.9

containing active clones. Again, this shows that fewer files need to be considered when focusing on clones that are active during runtime. Table 7 provides the statistics of the frequently executed active clones and their CPU time consumptions while Table 8 shows the breakdown of the active clones' activeness in terms of genealogy based on types. This explains which active clones types change more frequently and which ones not.

Clones are considered harmful for a number of reasons, including: there is the potential for bug propagation during cloning, cloning increases the size of the codebase, and changes in one clone may require changes to other clones during maintenance. These issues increase the potential for a higher maintenance effort as well as may burden the software developer with an additional cognitive load. Dynamic analysis provides a way to aid in our comprehension of clones. Systems containing clones could be segmented based on testing to make the maintenance task more modular. As well, runtime information allows us to consider other clone properties. Knowing which clones are the most CPU intensive during a particular set of system uses, could help identify places where optimizing one clone could be an important improvement to the system as a whole if we make similar changes to the clones in its clone class.

From Tables 3 and 4 we see that for HSQLDB out of 9,825 methods in the system of which 1,323 are clones, we have only 35–40 active clones for each test. For JHotDraw, out of a 3,260 methods in the system of which 656 are clones, we have only 56–60 for each test, for RText, out of 3,698 methods in the system of which 189 are clones, we have only 11 active clones for the tests, for jEdit, out of 7,428 methods of which 419 are clones, we have 220 active clones for the test, and for uniCentaoPOS, out of 4,260 methodds of which 285 are clones, we have 4–17 active clones in the tests. For HSQLDB, the active clones represent 492–545 lines code, out of 20,467 lines of clone code and a system that is 227,545 lines in size. For JHotDraw, the active clones represent 639–654 lines of code, out of 7,565 lines of clone code and a system size of 6,420. For RText, the active clones represent 265 lines of code, out of 3,019 lines of clone code and a 95,036 line system. For jEdit, the active clones represent 400 lines of code, out of 7,320 lines of clone code and system line of code of 115246. For uniCentaoPOS, the active clones represent 55–141 lines of code, out of 4578 lines of code and a system size of 4,3032. With respect to number of files, HSQLDB has 513 files, 293 with method clones and between 22–25 with active clones for the uses represented by the use sets. The system JHotDraw has 309 files in the system, 183 with clones and between 26–28 files with active clones, and RText has 319 files, 81 with clones and 7 with active clones. The system jEdit contains 571 files in the system, 161 with clones and

Table 4: Test Based Metrics for Active Clones

System	Use Set	Test	N_{int}	N_p	Method Clones				
					N_{ACLONE}	L_{ACLONE}	F_{ACLONE}	$\%M_{ACLONE}$	$\%CC_{ACLONE}$
HSQLDB	Search	Select	461	89	40	545	22	8.6	6.8
	Data Entry	Insert	441	87	35	492	22	7.9	6.2
	Admin	Update	441	89	35	492	22	7.9	6.6
		Delete	441	87	35	506	25	7.9	6.6
		Create	442	90	35	492	22	7.9	8.0
JHotDraw	Draw	Circle	798	189	58	654	27	7.2	19.9
		Rectangle	797	189	56	639	26	7	19.5
		Triangle	819	195	60	645	28	7.3	19.5
RText	Compose	Edit text	591	153	11	265	7	1.8	11.25
jEdit	Compose	Edit Text	3220	310	220	400	44	5.2	37.7
uniCentaoPOS	Search	Select	101	37	5	43	5	4.9	3.2
	Data Entry	Insert	373	85	17	141	15	14.9	9.2
	Admin	Update	76	32	4	55	4	5.2	0.8
		Delete	68	28	4	30	4	5.9	0.6

Table 5: Lines of Code and Number of Files Associated With Traces, Clones and Active Clones

Metrics	System				
	HSQLDB	JHotDraw	RText	jEdit	uniCentaoPOS
$\%L_{CLONE}$	3	13.4	8.9	6.4	10.6
$\%L_{ACLONE}$	0.24	1.2	0.31	0.71	0.31
$\%F_{CLONE}$	57.11	59.22	25.3	28.2	35.8
$\%F_{TRACE}$	17.3	61.2	47.9	51.9	17.5
$\%F_{ACLONE}$	4.2	9.1	2.3	7.4	3.1

44 with active clones while uniCentaoPOS contains 486 files, 174 with clones, and 4–15 files with active clones. Also if we compare the percentage coverage of active clones, lines of code, files and clone classes (Table 5) we find that a fewer percentage of clones are active for each individual test.

We can see that although the systems have a high concentration of clones, relatively few of them are executed. Only those clones that are test specific are active and are shown to be used more often. For example, if we consider the results in Table 5 then in the case of JHotDraw the lines of code for method clones is 13.4% (denoted by $\%L_{CLONE}$)) but only 1.2% (denoted by $\%L_{ACLONE}$) of code are containing active clones and likely requiring attention during maintenance. A similar picture can be seen for HSQLDB, RText, jEdit and uniCentaoPOS as well. On the other hand, from Table 5 we can also associate the coverage of clones over files. For instance, in HSQLDB 57% of files are associated with clones where only 17.3% on average are found in the trace (denoted by $\%F_{TRACE}$) while only 4.2% of file have active clones (denoted by

Table 6: Active Clones by Type of User Interface Event for Each Test

System	Test	UI Event Type	UI Traces	MI	%AC	CPU Time (ns)	
						Total	%AC
HSQLDB	Select	mouseClick (JButton)	3	191	6.8%	8.31E+10	14.8
		threadSignal	50	1555	5.2%	2.86E+12	47.6
	Insert	mouseClick (JButton)	2	203	6.4%	7.31E+11	14.8
		threadSignal	36	1076	7.2%	1.86E+12	51.26
	Update	mouseClick (JButton)	2	192	6.8%	3.16E+11	3.12
		threadSignal	28	1160	7.2%	2.93E+12	44.44
	Delete	mouseClick (JButton)	2	204	6.4%	3.76E+11	3.79
		threadSignal	28	885	7.9%	1.93E+10	42.5
	Create	mouseClick (JButton)	3	196	11.6%	2.05E+11	4.86
		threadSignal	51	1164	7%	1.02E+12	3.12
JHotDraw	Circle	mouseClick (mouse event)	17	397	6.8%	1.66E+12	25.8
		mouseClick (Button)	12	89	3.3%	1.34E+10	0.47
		threadSignal	1	490	6.1%	9.60E+11	33.8
	Triangle	mouseClick (mouse event)	13	375	6.9%	6.03E+11	17.02
		mouseClick (Button)	12	89	3.3%	8.60E+10	3.79
		threadSignal	1	490	6.1%	1.29E+12	57.09
	Rectangle	mouseClick (mouse event)	13	399	6.0%	1.73E+11	7.7
		mouseClick (Button)	12	89	3.3%	6.03E+10	3.92
		threadSignal	1	490	6.3%	1.00E+12	67.18
RText	Edit text	mouseClick (JButton)	1	18	2.2%	8.00E-3	0.001
		mouseClick (JMenuItem)	5	18	1.6%	2.50E+6	0.005
		displayUpdate	30	124	3.2%	3.65E+7	0.080
		threadSignal	11	79	2.5%	4.49E+10	15.5
jEdit	Edit text	mouseClick (Timer)	5	18	1.6%	2.39E+12	9.9
		displayUpdate	30	124	0%	4.83E+11	0
		threadSignal	11	79	2.5%	2.22E+13	85.2
uniCentaoPOS	Select	mouseClick (JButton)	2	71	7.0%	7.43E+10	44.8
		mouseClick (Timer)	50	1555	0%	2.86E+12	0
	Insert	mouseClick (JButton)	4	203	6.4%	1.61E+11	43.8
		mouseClick (Timer)	36	1076	7.2%	1.86E+12	11.9
	Update	mouseClick (JButton)	1	51	7.8%	6.50 +9	64.1
		mouseClick (Timer)	23	206	12.2%	2.93E+12	44.44
	Delete	mouseClick (JButton)	1	29	0	1.15E+10	0
		mouseClick (Timer)	53	208	13.9%	1.93E+10	66.2

UI: User Interface **MI**: Method Invocation **AC**: Active Clone

$\%F_{ACLONE}$). For JHotDraw and RText the percentage of files in the trace denoted by $\%F_{TRACE}$ are 61.2% and 47.9% while the percentage of files containing active clones denoted by $\%F_{ACLONE}$ are 9.1% and 2.3% respectively. Similarly, in case of jEdit and uniCentaoPOS $\%F_{TRACE}$ is 51.9% and 17.5% while $\%F_{ACLONE}$ is 7.4% and 3.1% respectively.

In summary, active clone detection has the following three characteristics if we just consider line, file and method metrics. First, tests can be used to identify active clones. Second, identifying active clone classes allows us to identify not only the clones that are active as a result of the test, but also (using static analysis) the clones that are related to the active clones through clone classes and by seeing which clones are involved with which kind of system uses. It is important to consider all the related clones when performing maintenance to help ensure we consistently change them. We note that for the systems and uses we studied, the active clone classes are between 2 and 6 in size. Third, there is evidence that active clones are present in each test case

we exercised. If we observe closely the statistics and comparison of active clones in terms of lines of code, files, and clone classes with active clones we can clearly see that active clones are very few in number involving less lines of code and less file coverage. For instance, in the case of HSQLDB a reduced number of lines of code, files and classes are found active per test. This may help guide a software developer in prioritizing which clones to look at by giving inspection priorities to active clones and their corresponding active clone classes instead of looking for every clone in the system.

RQ1: ***To what extent are clones active during runtime?*** Analysing our subject systems we see that clones are active even for simple tests and that with typical modest uses of a system that they are at least an order of magnitude fewer in number than the number of clones in a system. Active clones could be used to prioritize clone inspection during maintenance tasks by allowing us to focus on those clones related to the task at hand.

We can analyze active clone runtime properties related to the traces, such as frequency of invocation, order of invocation, relation of invocation to typical system uses, and resource utilization (i.e., CPU usage). Table 6 shows the active clone characteristic and the CPU usage of all the systems over each test case based on user interface (UI) events. The column *CPU time* denotes the total time consumed by events within the test and the column *%Active CPU Time* signifies the percent of CPU time consumed by the active clones executed by the corresponding event within the test. The system implementation under test is divided into two groups: GUI (Graphical User Interface) logic code and business logic code identified by the type of UI event. Thread events correspond to business logic functionality, and mouse clicks, keyboard presses and similar events correspond to GUI code. In the case for hsqldb, where mouse clicks on buttons and so on are grouped as GUI code while thread signals comprising the execution of the system's functionality/features are grouped as business logic code. For jhotdraw, the test case includes drawing a figure in an editor and therefore mouse events such as mouse press, mouse drag, mouse release and so on fall into the GUI section while the thread signal indicates business logic execution, as is the case for hsqldb. Similarly, for rtext a new event displayUpdate occurs and indicates GUI code being executed, and events such as mouse click and thread signal are analogous to that of hsqldb and jhotdraw.

From Table 6, we see that concentration of active clones in feature implementation denoted by threadSignal is higher than GUI code with over 7% for hsqldb for three of the tests and around 6.1% for jhotdraw which is less than GUI code and 2.5% for rtext which is higher than mouseClick but less than displayUpdate. Interestingly, if we compare the two mouse click events (buttons and mouse event) in jhotdraw we notice that there are a greater number of active clones associated with creating figures in the editor than clicking a button in the system. From Table 6, we can also observe that clones do execute and occupy a significant amount of CPU time. Thus, CPU time information of the active clones can also be an important parameter to consider during maintenance activity such as code optimization or determining inconsistent changes in clone. For instance, lets consider a class containing three clone fragments and all gets executed during a certain use. It is apparent that all the active clones should spend the same amount of time during execution. Therefore, any discrepancy in the timing value of the active methods could be an indication of certain method spending more time for execution. Possible reason could be a bug that have never been fixed here which made the methods inconsistent with other methods in the same class. In this way CPU time can be utilised to understand the behaviour of the methods

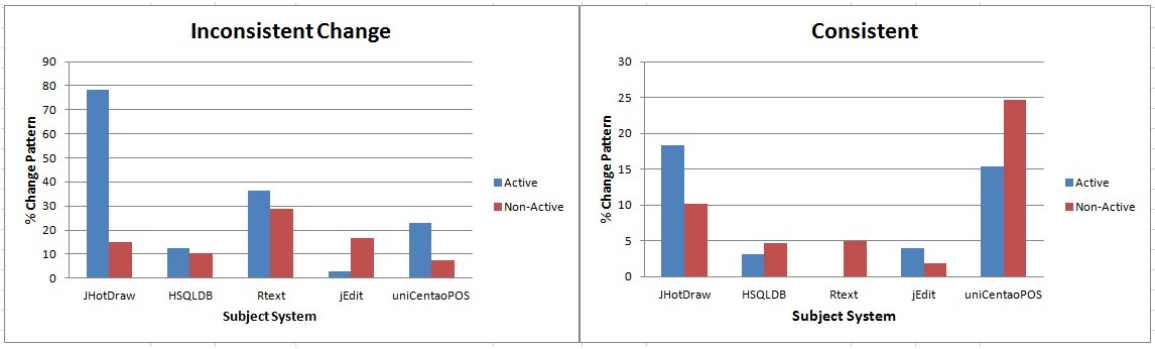

Figure 3: Change Patterns of Active Clones

and be used an an indicator of tracking abnormal behaviour of them in the clone class.

At the same time a segmented active clones can help developers identify which section of the system's code to inspect as a part of the maintenance task. Thus, there is a possibility that effort may be reduced by providing the developer with more detailed, fine grained data on active clones. For instance, if new features in the GUI are being added or modified as part of the maintenance task then the corresponding UI events and the associated active clones can help them in program comprehension. Code inspection may be reduced and the segmented active clones can further provide a way for effective clone management. As an example, we notice that for a number of uses, active clones are taking up a significant portion of the CPU time. For example, 33%–67% of the CPU time is involved with active clones for a small set of traces. When we look at the active clones in more detail we see that clones from an active clone class are involved in multiple trace. If we look at Table 7 we can find that active clone methods gets executed in all the tests with higher number of frequencies and CPU time consumption in all the five systems. Interestingly, we can also observe in case of HSQLDB, RText and uniCentaoPOS that the fragments in the same clone class gets executed with a higher number of invocations. For instance, in HSQLDB methods readByte() and readBoolean() belong to the same clone class 398 and is invoked 2509 and 320 times which is a significant amount. For RText, methods windowGainedFocus() and windowLostFocus() belonging to the same clone class and gets executed six times each while for uniCentaoPOS methods fireDataContentsChanged() and fireDataIntervalAdded() are invoked once.

RQ2: ***How active are the active clones*** The research question RQ1 addressed the dynamic behaviour of the active clones involved during the runtime of the systems. It signified that clones do exercise when a system is used and each feature contains active clones. Since maintenance of a system is a continuous process, changes in the code is done to meet customer requirements, fix bugs or optimize to make the system more efficient. Therefore software evolves with time and during which changes are made to the systems. In our study we are interested to look at how the active clones participate in the maintenance phase when the developers modify the source code. More specifically we wanted to study how the active clones evolve. We have adapted the genealogy extractor, called gCAD [SRS11] to extract the genealogy of the systems over a number of versions. We then post processed the genealogy of the systems to map the active clones and

Table 7: Frequently Executed Active Clones by System

System	Active Clone	Times Invoked	CPU Use (nanosec)	Clone Class	
				Id	Size
HSQLDB	readByte()	2509	1.02E+7	398	3
	getOrAddInteger()	421	2.48E+6	30	2
	writeShort()	43	1.07E+5	464	2
	readBoolean()	320	3.20E+4	398	3
	getInt()	5	4.09E+5	496	4
JHotDraw	basicGetBounds()	8	2.31E+6	207	6
	ensureSorted()	7	1.77E+6	186	2
	updateEnabledState()	6	8.05E+5	140	3
	fireToolDone()	1	9.56E+6	103	3
RText	getRTextScrollPaneAt()	33	5.10E+5	71	2
	createCheckBox()	6	1.51E+6	55	2
	windowGainedFocus()	6	1.04E+3	79	2
	windowLostFocus()	6	6.90E+3	79	2
	setSearchParameters()	1	1.29E+7	25	2
jEdit	getScreenLineStartOffset(int)	911	8.21+E9	196	2
	jj_3R_129()	234	3.12E+10	14	4
	jj_3R_216()	226	3.92E+9	31	4
	EqualityExpression()	90	9.71E+10	8	6
	jj_3R_37()	639	7.58E+10	129	6
uniCentaoPOS	readValues(int)	3	8.21+E9	67	2
	fireDataContentsChanged	1	2.7E+8	28	3
	activate()	1	8.14E+9	97	7
	fireDataIntervalAdded ()	1	7.57E+6	28	3
	getDialog(Component)	1	5.04E+8	69	3

identified their patterns. In our study we have considered two significant types of change patterns, a) consistent, and b) inconsistent change patterns. We then compared the activeness of clones against the active and the non-active code clones. Figure 3 shows the change patterns of the active clones and non-active clones for each systems.

If we look closely at Figure 3 for the consistent change, we can see that for the systems JHotDraw, HSQLDB, jEdit and uniCentaoPOS both the active and the non-active clones change at the same time and the difference in activeness between them is less than 6% with an exception of RText where no active clones participate in the change pattern. Therefore, active clones do play a vital role and have significant impact on the maintenance activity.

Although practice of code cloning speeds up the development process, it also poses threat of bug propagation due to inconsistently changing the clone fragments. If we observe Figure 3 we can also see that active clones are more active than the non-active clones and surpasses the non-active ones. In case of JHotDraw, HSQLDB, RText and uniCentaoPOS the active clones have changed more inconsistently than the non-active clones. In case of JHotDraw almost 80% of the active clones are changed inconsistently compared to non-active clones which is around 15%. Similarly for HSQLDB, RText and uniCentaoPOS the percentage of inconsistent changes of active and non-active clones are around 15%, 35%, 22% and 10%, 29% and 8% respectively. However in case of jEdit even though the non-active clones have changed more inconsistently than the active clones we still get some active clones that have changed inconsistently.

RQ3: ***Does active clone identification support software maintenance activities?*** From our

Table 8: Number of Genealogies of Active and Non-Acitve Clones by Type

System	Type	Active Clones			Non-Active Clones		
		Static	CC	IC	Static	CC	IC
JHotDraw	Type-1	7	2	6	60	43	60
	Type-2	1	2	9	53	47	64
	Type-3	3	6	31	158	66	193
HSQLDB	Type-1	2	0	0	76	3	5
	Type-2	2	1	0	75	6	6
	Type-3	25	0	2	283	15	42
RText	Type-1	0	0	1	9	3	2
	Type-2	3	0	0	37	2	6
	Type-3	4	0	3	75	4	45
jEdit	Type-1	1	0	0	3	0	2
	Type-2	25	0	0	41	2	3
	Type-3	45	3	2	82	1	21
uniCentaoPOS	Type-1	0	0	0	15	2	1
	Type-2	4	0	0	30	12	1
	Type-3	6	2	3	76	30	14

CC: Consistent Change, **IC**: Inconsistent Change

study we see that awareness of active clones and their runtime behaviour may help support maintenance by allowing developer's to focus on similar code fragments whose improvement could benefit the use of the system as a whole and on clone classes that are involved in system uses related to the task at hand. This is well supported by the research questions RQ1 and RQ2. The dynamic properties of active clones such as times of invocations, CPU time, and active clone genealogies can be used to prioritize clone inspection during maintenance activities. Clones that are invoked a greater number of times and that occupy higher percentage of CPU time can be put at the highest priority in the list of clone inspection during clone management.

Previous study [SRSP13] on the change patterns on the genealogies of clones show that Type-3 clones are more inconsistent than the Type-1 and Type-2 clones which require more careful attention during maintenance. With regards to this, we also tried to conduct a similar study based on our framework to reveal the change pattern of the active clone types. Table 8 shows the comparison of activeness of Type-1, Type-2 and Type-3 clones between the active and non-active clones. If we observe closely, we can see that the Type-3 clones have a greater tendency to change when compared to Type-1 and Type-2 clones. In all the change patterns, the Type-3 active clones surpasses that of Type-1 and Type-2 clones. More interestingly, on an average 31.7% of Type-3 active clones change inconsistently while Type-1 changes slightly less than 29%. Similarly, for consistent change pattern 4.2% of active clones are Type-3 with no change for Type-1 and Type-2. On an average for static change, 60.4% of Type-3 clones are static in nature followed by Type-1 which is 49.2% with Type-2 clones standing at 74.6%. We also noticed that Type-3 clones are significantly less stable than the Type-2 clones with an exception of Type-1 clones. This is because of the absence of Type-1 active clone fragments in RText and uniCentaoPOS. In case of non-active clones for all the three change patterns we can see that Type-3 clones possess inconsistent changes with over 50% followed by Type-2 and Type-1. Therefore, the study conducted in the activeness of active clones based on types can also add

benefits to software maintenance, especially when clones need to be managed. We can conclude that further enhancement in clone management activity can be done by prioritizing the clone inspection based on types.

7 Threats to Validity

There are a number of threats to the validity of our analysis. Execution traces are based on the tests we provided. Additional testing would reasonably result in additional methods being executed with the possibility of additional active clones being identified and thus the reduction in clones to be considered may not be as significant. However, our intention of this study is to only focus on test specific clones during the maintenance of that related functionality, and it is obvious that there will be a reduction of active clones (hence the reduction of active clone classes) for maintenance as evident by our results.

8 Related Work

Code clone research traditionally focused on clone detection. Recently, managing code clones has become of interest to the research community. A number of tools supporting management of clones have already been proposed. Tools such as CLONEBOARD [WZD08] tracks changes by dynamically monitoring clipboard activity. CloneTracker [DR08] maintains a model of all clones in the codebase and handles clone changes using linked resolution. There are also a great many efforts towards analyzing and managing clones [XXJ11, YCY+13, HKKI05, TG12, MLPB13, PTK13]. Overall, our work significantly differs from all others in the sense that we use dynamic analysis to get a subset of clones related to a particular maintenance task and thus can prioritize clones of the systems based on the maintenance task at hand. To the best of our knowledge this is the first study to investigate runtime implications of source code clones.

Zibran and Roy [ZR13] proposed a method of ranking the important candidate clones for refactoring. Our work differs from these, as we focused on detecting and tracking only a subset of clones related to the maintenance tasks at hand.

9 Conclusion And Future Work

We presented an approach for assisting a software developer in the maintenance of systems containing clones by focusing on those clones that are active during runtime. We used a hybrid approach to mine a system's source code and capture execution traces to identify active clones using four different categories of metrics. We also used the concept of user interface traces to partition execution traces based on a user's interaction with the system. In this way, we identified active clones, that is, those clones that are part of the methods invoked during a test. We believe that this approach can help focus maintenance effort in an informative way (active clones based on events such as thread signals, display updates and so on) when a maintenance effort is required or where previously documented active clone information can be searched for fixing bugs or other maintenance related updates. Our technique provides guidance to developers

by helping to prioritize their efforts and providing clone awareness.

As future work we plan to further support clone management using the approach and explore the relationships between active clones with more complete and different kinds of testing. We also plan to make the technique available for more general use by integrating it into the Eclipse IDE as a plugin.

Bibliography

[CR11] J. R. Cordy, C. K. Roy. The NiCad Clone Detector. In Proc. ICPC Tool Demo Track, pp. 219–220. 2011.

[CZD+09] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, R. Koschke. A Systematic Survey of Program Comprehension through Dynamic Analysis. IEEE Transactions on Software Engineering 35(5), pp. 684–702. 2009.

[DR08] E. Duala-Ekoko, M. Robillard. Clonetracker: Tool support for code clone management. In Proc. ICSE, pp. 843–846. 2008.

[HKKI05] Y. Higo, T. Kamiya, S. Kusumoto, K. Inoue. ARIES: refactoring support tool for code clone. SIGSOFT Software Engineering Notes 30(4), pp. 1–4. 2005.

[HL05] A. Hamou-Lhadj, T. Lethbridge. Measuring Various Properties of Execution Traces to Help Build Better Trace Analysis Tools. In Proc. ICECCS, pp. 559 – 568. 2005.

[JDHW09] E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner. Do code clones matter? In Proc. ICSE', pp. 485–495. 2009.

[KG08] C. J. Kapser, M. W. Godfrey. Cloning considered harmful" considered harmful: patterns of cloning in software. In Proc. ICSE 13(6), pp. 645–692. 2008.

[Kha13] M. A. Khan. Supporting Source Code Feature Analysis Using Execution Trace Mining. M.Sc. thesis, University of Saskatchewan, p. 113. 2013.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold. An overview of AspectJ. Lecture Notes in Computer Science, pp. 327–353. 2001.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, J. Irwin. Aspect-Oriented Programming. In Proc. ECOOP 1241, pp. 220–242. 1997.

[KSNM05] M. Kim, V. Sazawal, D. Notkin, G. C. Murphy. An empirical study of code clone genealogies. In Proc. ESEC-FSE, pp. 187–196. 2005.

[LW08] A. Lozano, M. Wermelinger. Assessing the effect of clones on changeability. In Proc. ICSM, pp. 227–236. 2008.

[MLPB13] E. Merlo, T. Lavoie, P. Potvin, P. Busnel. Large scale multi-language clone analysis in a telecommunication industrial setting. In Proc. IWSC, pp. 69–75. 2013.

[PTK13] J. R. Pate, R. Tairas, N. A. Kraft. Clone evolution: a systematic review. Journal of Software: Evolution and Process 25(3), pp. 261–283. 2013.

[RC07] C. K. Roy, J. R. Cordy. A survey on software clone detection research. School of Computing Tech Report 2007-541, Queens University, Canada, p. 115. 2007.

[RC08] C. K. Roy, J. R. Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In Proc. ICPC, pp. 172–181. 2008.

[RC09] C. K. Roy, J. R. Cordy. A mutation/injection-based automatic framework for evaluating code clone detection tools. In Proc. ICST Workshop, pp. 172–181. 2009.

[RC10] C. K. Roy, J. R. Cordy. Near-miss function clones in open source software: an empirical study. *Journal of Software: Evolution and Process* 22(3):165–189, 2010.

[Roy09] C. K. Roy. Detection and analysis of near-miss software clones. In Proc. ICSM, pp. 447–450. 2009.

[Sou] Sourceforge. SourceForge: www.sourceforge.net.

[SRS11] R. K. Saha, C. K. Roy, K. A. Schneiderl. An Automatic Framework for Extracting and Classifying Near-Miss Clone Genealogies. In Proc. ICSM, pp. 293–302. 2011.

[SRSP13] R. K. Saha, C. K. Roy, K. A. Schneider, D. E. Perry. Understanding the Evolution of Type-3 Clones: An Exploratory Study. In Proc. MSR, pp. 139–148. 2013.

[SS09] A. Sutherland, K. Schneider. UI Traces: Supporting the Maintenance of Interactive Software. In Proc. ICSM, pp. 563–566. 2009.

[TG12] R. Tairas, J. Gray. Increasing clone maintenance support by unifying clone detection and refactoring activities. Information & Software Technology 54(12), pp. 1297–1307. 2012.

[WZD08] M. de Wit, A. Zaidman, A. van Deursen. Managing Code Clones Using Dynamic Change Tracking and Resolution. In Proc. ICSM, pp. 169–178. 2008.

[XXJ11] Y. Xue, Z. Xing, S. Jarzabek. CloneDiff: semantic differencing of clones. In Proc. IWSC, pp. 83–84. 2011.

[YCY+13] Y. Yamanaka, E. Choi, N. Yoshida, K. Inoue, T. Sano. Applying clone change notification system into an industrial development process. In Proc. ICPC, pp. 199–206. 2013.

[ZR13] M. F. Zibran, C. K. Roy. Conflict-aware Optimal Scheduling of Code Clone Refactoring. IET Software 7(3), pp. 167 – 186. 2013.