An Empirical Study of Function Clones in Open Source Software

Chanchal K. Roy and James R. Cordy
School of Computing, Queen's University
Kingston, ON, Canada K7L 3N6
{croy, cordy}@cs.queensu.ca

Abstract

The new hybrid clone detection tool NICAD combines the strengths and overcomes the limitations of both text-based and AST-based clone detection techniques to yield highly accurate identification of cloned code in software systems. In this paper, we present a first empirical study of function clones in open source software using NICAD. We examine more than 15 open source C and Java systems, including the entire Linux Kernel and Apache httpd, and analyze their use of cloned code in several different dimensions, including language, clone size, clone location and clone density by proportion of cloned functions. We manually verify all detected clones and provide a complete catalogue of different clones in an online repository in a variety of formats. These validated results can be used as a cloning reference for these systems and as a benchmark for evaluating other clone detection tools.

1. Introduction

Reusing a code fragment by copying and pasting with or without minor modifications is a technique frequently used by programmers, and thus software systems often have duplicate fragments of code in them. Such duplicated fragments are called *code clones* or simply *clones*.

Although cloning is beneficial in some cases [13] and often programmers intentionally use it [17], it can be detrimental to software maintenance [8, 12]. For example, if a bug is detected in a code fragment, all the fragments similar to it should be investigated to check for the same bug [19], and when enhancing or adapting a piece of code, duplicated fragments can multiply the work to be done [12].

Over the past decade several techniques and tools for detecting code clones have been proposed [25]. However, despite a decade of active research there has been a marked lack of in-depth comparative studies of cloning in a variety of systems. There have been many empirical studies on cloning, for example every new technique comes with some sort of empirical validation [25, 24], and empirical studies are used when comparing tools [5, 6]. However, in both cases the focus is on validating or comparing the techniques rather than the clone properties of the subject systems themselves. Particular subject systems have also been analyzed with respect to aspects such as categories [15] or evolution of clones [4], and there has been one in-depth study [20] on exact clones in web applications.

In this paper we provide an in-depth empirical study of function clones in more than 15 open source C and Java systems including Apache *httpd* and the entire Linux Kernel, using our recently introduced hybrid approach, NICAD [23], which combines the strengths and overcomes the limitations of both text-based and AST-based techniques. Our detailed results are in an online repository [22] as well as in this summary paper.

NICAD has been found to be effective in detecting near-miss function clones [23]. However, it was applied to only two small C systems, focusing on its efficacy in detecting copy/pasted near-miss clones by using flexible pretty-printing, code normalization and code filtering. In this paper, we exploit further improvements to NICAD to deal with large systems, using a dynamic clustering technique and distributing the comparison load across multiple processors, and an adaptation to its use on Java systems.

Although NICAD is designed to allow for flexible pretty-printing, code normalization and filtering, in this paper we focus on detecting only exact and near-miss function clones, using only the basic NICAD technique, consisting of standard pretty-printing of code fragments followed by text line comparison at a variety of dissimilarity thresholds. Our study shows that NICAD is capable of detecting clones in very large systems in different languages and that there is a significant proportion of code in these systems that has been reused by copy/paste.

The rest of the paper is organized as follows. Following a short introduction to NICAD in Section 2, we provide the experimental setup of our study in Section 3. Section 4 presents and analyzes our findings, and Section 5 considers other empirical studies and their relation to ours. Finally, Section 6 concludes the paper with directions of future research.

2. NICAD Overview

The clone detector used in this study is a significantly improved and adapted version of NICAD [23], which works in three phases : *Extraction*, in which all potential clones (code fragments of the target kind) are identified and extracted, *Comparison*, in which the potential clones are clustered and compared, and *Reporting*, in which discovered clone pairs and classes are related to original source and presented for human inspection.

NICAD begins by extracting the set of all code fragments of the desired granularity in the system, each to a set of pretty-printed and source-coordinate annotated *potential clones*. Pretty-printing allows us to use efficient text comparison while remaining structure sensitive by preserving the parsed structure in the pretty-printed code.

In this paper we are interested in function clones, so NICAD was asked to extract all function and method definitions with their original source-coordinates. Because we are interested in intentional copy/pasted clones, C macros are not expanded, but *#ifdefs* are resolved using the method of Antoniol et al. [4] to comment out all *#else* parts.

NICAD compares the pretty-printed potential clones line-by-line textwise using a longest common subsequence (LCS) algorithm similar to the Unix *diff* utility. To determine whether two potential clones really are clones of each other, we compare their pretty-printed (and optionally normalized/filtered) text lines and use the number of unique lines in each as a measure of similarity/dissimilarity. In particular, we compute the size-sensitive Unique Percentage of Items (UPI) for each potential clone usng the equation (for details see [23]):

$$UPI = \frac{No.\ of\ Unique\ Items * 100}{Total\ No.\ of\ Items}$$

If the UPI for both line sequences is zero or below a certain threshold, the potential clones are considered to be clones. The number of comparisons is optimized using an exemplar-based technique that builds clone classes directly, by choosing the largest unclassified potential clone as an exemplar and comparing similarly-sized potential clones to it depending on the UPI.

Results from NICAD are reported in both XML database form and as an interactive HTML website to assist hand validation.

3. Experimental Setup

In this experiment we have applied NICAD to find function clones in a number of open source systems. We have then used a set of metrics to analyze the results. This section introduces the systems we have studied and the metrics used, including a brief overview of our definition and methodology for manual verification of the detected clones.

Table 1. Overview of the subject systems

Lang	System	LOC	Method
C	Abyss [1]	4K	148
	Bison [5]	16K	315
	Cook [5]	70K	1362
	Gzip-1.2.4 [9]	8K	117
	Apache-httpd-2.2.8 [2]	275K	4301
	Postgresql [5]	202K	4669
	snns [5]	94K	2201
	Weltab [5]	11K	123
	Wget [5]	17K	219
	Linux-2.6.24.2 [16]	6265K	154977
Java	Eclipse-ant [5]	35K	1754
	EIRC [5]	11K	588
	Netbeans-Javadoc [5]	14K	972
	Eclipse-jdtcore [5]	148K	7383
	JHotDraw 5.4b1 [11]	40K	2399
	Spule [5]	13K	420
	j2sdk-swing [5]	204K	10971

3.1 Subject Systems

In this study we have analyzed ten C and seven Java systems varying in size from 4K LOC to 6265K LOC and including the entire Linux Kernel. In Table 1 we provide a statistical overview of these subject systems (only .c and .java files were considered in the calculations).

Because Bellon's experiment [5] is the most extensive to date, we have chosen all the C and Java systems of his experiment including the systems used in their test run. In addition, we have studied Apache *httpd* [2], *JHotDraw* [11], the entire Linux Kernel [16] and a number of small systems.

3.2 Clone Definition

In this study we have considered all non-empty functions of at least 3 LOC in pretty-printed format (function header and opening bracket on the first line, at least one code line, and ending bracket on the third line). Empty functions, which are common in some systems, have intentionally not been considered. We then use different UPI (difference) thresholds (c.f. Section 2) to find exact and near-miss (copy/paste/modify) function clones. For example, if the UPI threshold is 0.0, we detect only exact clones; if the UPI threshold 0.10, we detect two functions as clones if at least 90% of the pretty-printed text lines are the same (i.e., if they are at most 10% different – see [23]). In this paper we present our results for the representative set of UPI thresholds 0.0, 0.10, 0.20 and 0.30, although we have also tested 0.05, 0.15, 0.25 and 0.35 in our work.

3.3 Validation of Clones

All clones detected in this study were validated by hand. To validate detected clones we use a two-step process. First, we use NICAD's interactive HTML output to give an overall view of the original source of the clone classes. Second, we use the XML output to pairwise compare the original source of the functions in each clone class using Linux *diff* to determine the textual similarity of the original source. We then manually check all code clone pairs that have lower similarity values w.r.t the UPI threshold chosen. Because of our concise interactive HTML view and tool support for comparing original source, manual validation is not time-consuming, and the total time to manually validate all clones in this experiment was less than one man-month.

3.4 Metrics and Visualizations

This subsection descibes the different metrics and visualizations that we have used in this experiment. These metrics are either adapted or reused from previous studies of cloning [20, 3, 15, 26, 21].

Total Cloned Methods (TCM): In this study we focus on function clones, and thus our first metric is related to the number of methods. By *TCM* we mean the total number of cloned methods of a system for a given UPI threshold (after manual verification). *TCMp* is the percentage *TCM* of the total number of methods in a system. A higher value of *TCMp* corresponds to a higher percentage of cloning in the system w.r.t the number of methods. For example, if the *TCMp* of a system is greater than 50% with UPI threshold 0.0, we can say that the system has more exact cloned methods than non-cloned methods. Such systems have a high update anomaly risk; every update to the system has a higher chance of involving a clone than not.

Since methods can be of different sizes and there may be many clones that are quite small, we also consider similar metrics w.r.t the number of lines in the systems. We define *TCLOC* as the total number of cloned lines of a system for a given UPI threshold and *TCLOCp* as the percentage of total number of lines of the system for a given UPI threshold. Since we apply standard pretty-priting before clone detection, which eliminates formatting and layout differences, resolves *#ifdefs* (in C systems) and ignores comments, we can get an accurate percentage of cloned lines. We thus define the similar metrics w.r.t standard pretty-printed lines of code as *TCSppLOC* and *TCSppLOCp* respectively. In practice, there is not much difference between *TCLOCp* and *TCSppLOCp*, but at the same time *TCSppLOCp* gives a more accurate measure. We thus provide our findings w.r.t *TCSppLOCp* rather than both.

File Associated with Clones (FAWC): While the above metrics provide the overall cloning statistics for a subject system, they cannot provide any clue as to whether the clones are from some specific files or scattered all over the system among many files. With *FAWC* we provide these statistics for each system at each UPI threshold. We consider that a file is associated with clones if it has at least one method that forms a clone pair with another method in the same file or in a different file. We define *FAWCp* as the percentage of files associated with clones of a system for a given threshold. For example, "*FAWCp* of a systems x with UPI threshold 0.0 (exact clones) is 50%" means that 50% of the files of x contain at least one exact cloned method. From a software maintenance point of view, a lower value of *FAWCp* is desirable, as in this case clones are localized to certain specific files and thus may be easier to maintain.

Cloned Ratio of File for Methods (CRFM): While *TCM* related metrics provide a good indication of the overall cloning level and *FAWCp* hints at the overall localization of the clones, still one cannot say which files contain the majority of the clones in the system. With *CRFM* we attempt to discover the highly cloned files. In particular, for a file f, $CRFM(f)$ is defined as follows:

$$CRFM(f) = \frac{Total\ number\ of\ cloned\ methods\ in\ file\ f\ *\ 100}{Total\ number\ of\ methods\ in\ file\ f}$$

Where a method is considered to be a *cloned method* if it forms a clone pair/clone class with another method(s) of the same file (e.g., for near-miss clones) or another file (within the same directory or a different directory) and *total number of methods in file f* denotes the number of methods of f that are 3 LOC or more in standard pretty-printed format. Similar metrics are defined w.r.t the lines of code (*CRFLOC*) and standard pretty-printed lines of code (*CRFSppLOC*). These metrics are similar to (but not same as) the *FSA* metric of Rajapakse and Jarzabek [20] and the *RSA* metric of Ueda et al. [26], although they ignore clones that form clone pairs within the same file and we do not.

With *CRFM* we can determine the highly cloned files of a system and possibly can also predict the maintenance difficulty based on the metric values. For example, consider two systems x and y of similar size, both having the same values for the *TCM* related metrics. In x, clones are scattered across the system in such a way that no two files are substantially similar. But in y, clones are well concentrated into a certain set of files. From a clone treatment perspective, system y is more interesting than x because the clones in y might be more easily treatable than those of x.

Qualifying File Count for Methods (QFCM): As in Rajapakse and Jarzabek [20] we define *QFCM(v)* for *CRFM* value v as the number of files for which *CRFM* is not less than v. For example, QFCM(20%) gives the number of files in the system having a *CRFM* value not less than 20%. *QFCMp* is *QFCM* expressed as a percentage of the total number of files in the system. For example, "*QFCMp*(30%+) = 28% for a system x with UPI threshold 0.0" means that 28% of the files of x have 30% or more exact cloned methods. As usual we define similar metrics for source lines of code (*QFCLOC* and *QFCLOCp*) and for standard pretty-printed lines of code (*QFCSppLOC* and *QFCSppLOCp*).

Profiles of Cloning Locality w.r.t Methods (PCLM): Kapser and Godfrey [15] provide three types of function clones based on their location – clone pairs in the same file (category 1), in the same directory (category 2) and in a different directory (category 3). They also provide the reasons, usefulness / harmfulness for each of these categories [15].

In this study we define three metrics, $PCLM(1)$ for category 1, $PCLM(2)$ for category 2 and $PCLM(3)$ for category 3 where $PCLM(i)$ gives the total number of clone pairs for category i. Furthermore, $PCLMp(i)$, the percentage clone pairs for category i is defined as follows:

$$PCLMp(i) = \frac{PCLM(i) * 100}{Total\ number\ of\ clone\ pairs\ in\ the\ system}$$

As usual similar metrics are defined with respect to lines of code (*PCLLOC and PCLLOCp*).

4. Experimental Results

In this section we provide the experimental results of this study starting from overall cloning level in both C and Java systems and then for each individual system in a variety of measures based on the metrics described in Section 3.4. While we provide here only the overall findings and statistical measures, the detailed results and the raw data for each systems for different UPI thresholds can be found in an online repository [22] as XML databases and HTML website.

4.1 Overall Cloning Level

In this section we provide overall cloning level both in terms of number of methods and number of pretty-printed LOC, i.e., the values of *TCM* related metrics of subsection 3.4. Figure 1 summarizes our results on C and Java systems by the proportion of functions (or methods in the case of Java) that are cloned (i.e., *TCMp* over languages). The values for the *TCSppLOCp* metric (i.e., the proportion by number of pretty-printed LOC) by language can be found in the *% Total* rows of Table 2.

The first thing we can notice is that there is significantly more function cloning in our open source Java systems than in C. On average, about 15% (7.2% w.r.t LOC) of the methods in open source Java programs are exact clones - those with no changes at all, whereas only about 2.5% (1.1% w.r.t LOC) of C functions are exact clones. After detecting clones by setting a size range for the function size (see [22] for detailed results) we noticed that this is in large part due to the large number of accessor methods in Java programs that are not present in C.

The second thing we can notice in Figure 1 (and in the *% Total* rows of Table 2 for LOC) is that the effect of increasing the UPI threshold is almost identical in both languages. We can interpret this as meaning that the numbers of small changes made to cloned functions in each of these languages seems to be roughly the same in these systems. This is in some ways surprising - there is no particular reason why the pattern of changes to copied code should be similar across languages.

Figures 2 and 3 (also columns 3 to 6 of Table 2) refine Figure 1 to show a detailed view of the same information for the individual open source C and Java systems respectively. As expected, the overall trends for each language are much like the summaries in Figure 1 (or the *% Total* rows of Table

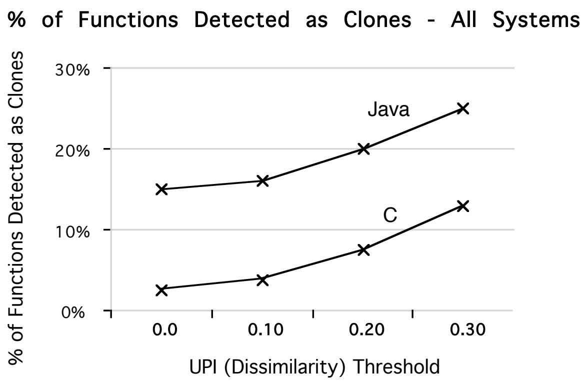

Figure 1. Proportion of functions/methods that are cloned, by language

2), with lower levels of cloning in C than Java. However, we can now see more. For example, we can see that several of the C systems have very little function cloning at all - less than 10%, and to a large extent independently of the system size (e.g., *Postgresql* and Apache *httpd* are very large, whereas *snns*, *cook* and *Weltab* are quite small). Moreover, contrary to popular intuition, the cloning characteristics of the Linux Kernel seem to be not significantly different from any other kind of C system.

Figure 3 (also Table 2 for LOC) is even more interesting, because while the C systems vary, we can see that the Java systems are remarkably consistent in their cloning characteristics. All begin with a relaitively high level of exact method clones (betweeen 8 and 22 percent), and in all cases allowing for changes increases the proportion only modestly. What's interesting is that this seems to be completely independent of system size, and appears to be a characteristic of the language. The only exception to this consistency seems to be *JDTcore*, which has about twice as many clones at the 0.30 dissimilarity level than exact clones. More research will be needed to investigate this phenomenon and compare to other object-oriented languages.

In Table 2 we also provide the number of clone pairs and clone classes for each of the systems for varying UPI thresholds. It is interestng to notice that most systems have significantly fewer clone classes than clone pairs, indicating the fact that there are many pairs of functions in the systems that are similar to each other.

4.2 Clone Associated Files

The *FAWC* and *FAWCp* metrics of Section 3 address the issue of what proportion of the files in a system is associated with clones, that is, contains at least one cloned method. A system with more clones but associated with only a few files is in some sense better than a system with fewer clones scattered over many files from a software maintenance point of view. In this section, we examine the *FAWCp* metrics for each of the systems with varying UPI thresholds. The third

Table 2. Percentage of LOC that are associated with clones with clone pairs and clone classes

		% Cloned LOC				No. of Clone Pairs				No. of Clone Classes			
Lang	System	T=0.0	T=0.1	T=0.2	T=0.3	T=0.0	T=0.1	T=0.2	T=0.3	T=0.0	T=0.1	T=0.2	T=0.3
C	Abyss	0	1.9	1.9	3.2	0	1	1	3	0	1	1	3
	Bison	0.2	0.2	2.0	2.9	1	1	3	8	1	1	3	8
	Cook	0.3	2.0	7.7	13.3	7	18	107	280	5	12	56	98
	Httpd	2.1	4.1	6.2	9.6	183	224	322	711	107	133	195	276
	Postgresql	0.1	1.0	4.3	9.43	7	24	195	530	7	20	89	203
	Snns	3.2	6.2	13.3	18.6	109	157	343	495	63	86	143	191
	Wetlab	21.0	55.2	62.7	72.2	46	105	148	160	8	11	17	20
	Wget	0.0	1.3	1.7	2.4	0	2	4	11	0	2	2	2
	Linux	1.0	2.6	8.3	10.8	5953	7362	13813	25767	1505	2263	4613	7918
% Total		1.1	2.8	8.4	11.0								
Java	Ant	5.1	5.4	6.3	9.7	363	365	374	426	92	94	101	119
	EIRC	7.2	7.2	7.7	10.9	117	117	121	149	35	35	36	47
	Javadoc	10.8	12.6	18.6	24.0	193	197	240	304	80	82	95	110
	Jdtcore	5.1	8.8	16.2	23.7	1427	1553	2126	4378	323	377	518	660
	JHotDraw	7.6	8.28	12.0	19.1	291	295	377	598	137	141	170	208
	Spule	2.0	2.7	3.1	5.9	60	64	68	113	11	13	15	19
	Swing	9.4	11.0	15	19.4	8115	8203	9978	11209	516	558	687	843
% Total		7.2	9.4	14.4	20.0								

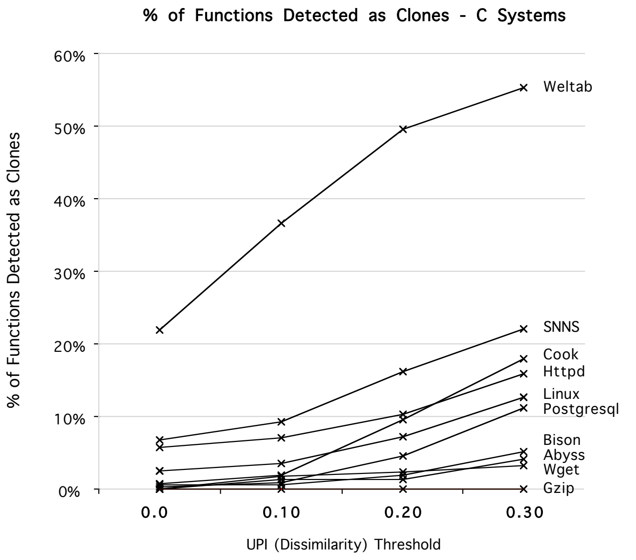

Figure 2. Proportion of function/methods that are cloned for C systems

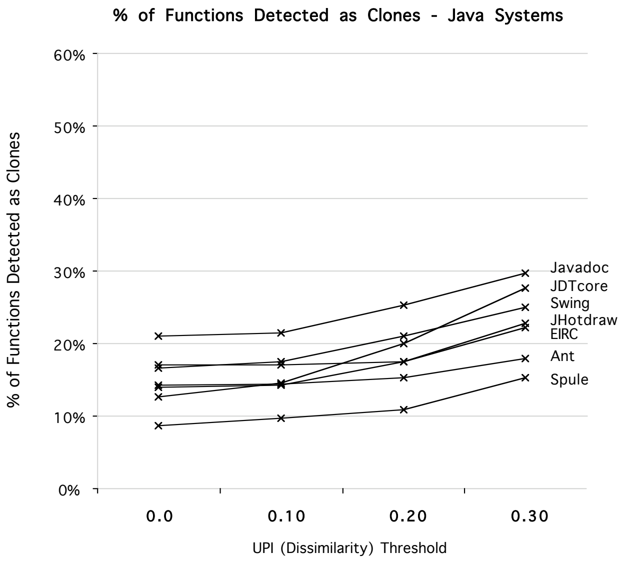

Figure 3. Proportion of function/methods that are cloned for Java systems

column (with column heading *v=0+% Both*) of Table 3 and Table 4 shows the statistics for the C and Java systems respectively. The first column of these tables shows the subject systems and the number of files with at least one method of three or more lines in standard pretty-printed format. The second column shows the different UPI thresholds used while calculating the metric values. The last row shows the average values for each of the language categories.

While we will look at the details in the other columns of these tables in the next subsection, from the average values (last row and third colmn) of the tables, we see that on average 15.1% of the files in the C systems and 45.7% of the files in the Java systems are associated with exact clones (UPI threshold 0.0). The higher percentage of Java systems can be explained by the fact that in Java systems there are many small similar accessor methods. However, in the case of the C system *Weltab*, 51.3% files are associated with exact clones. In case of Java systems, *Swing* shows the highest percentage, 65.9% for exact clones. When we increase the UPI threshold to detect near-miss clones, we see that C systems show a faster growing ratio than the Java systems, indicating the fact that there might be more near-miss clones in the C systems than the Java systems and that the clones are scattered across different files. For example, even for the largest C system, the Linux Kernel, it increases from 14.0% (for UPI threshold 0.0) to 49.6% (for UPI threshold

Table 3. Percentage of files that have clones over a certain percentage for C systems

System(#File)	UPIT	v=0+%	v = 10+%		v = 20+%		v = 30+%		v = 40+%		v = 50+%		v=100%
		Both	Meth	LOC	Meth	LOC	Meth	LOC	Meth	LOC	Meth	LOC	Both
	0.0	3.5	3.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	0.1	3.5	3.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Bison(57)	0.2	8.8	8.8	5.3	5.3	5.3	5.3	5.3	5.3	5.3	5.3	5.3	3.5
	0.3	19.3	19.3	19.3	15.8	15.8	15.8	15.8	15.8	15.8	14.0	14.0	10.5
	0.0	3.8	3.5	1.4	2.8	1.0	1.7	0.7	1.4	0.7	1.4	1.4	0.7
	0.1	8.4	7.7	5.2	6.3	4.5	4.5	3.8	2.8	2.8	2.4	2.4	1.7
Cook(287)	0.2	26.8	25.4	21.6	21.6	17.4	17.1	13.9	12.9	10.1	12.2	12.2	6.3
	0.3	43.6	42.2	35.9	36.9	29.3	31.4	24.7	25.4	22.0	24.7	1.0	2.5
	0.0	18.3	16.7	10.9	12.5	6.7	8.3	4.2	6.5	2.6	4.8	4.8	1.0
	0.1	22.2	20.0	14.5	14.7	9.1	9.9	6.9	8.5	4.2	6.0	6.0	1.0
Httpd(496)	0.2	31.0	28.4	22.4	23.2	16.1	15.9	11.3	12.1	9.1	10.1	10.1	2.0
	0.3	41.7	39.3	31.9	33.5	23.8	23.6	18.1	19.6	14.9	15.5	15.5	3.8
	0.0	1.9	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	0.1	4.5	2.2	0.6	0.0	0.6	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Postgres(314)	0.2	16.9	9.6	8.3	4.1	4.1	2.5	2.5	1.6	1.3	1.3	1.3	0.3
	0.3	30.3	21.3	18.2	12.4	10.8	8.3	8.9	5.7	6.0	3.2	4.5	0.3
	0.0	13.0	8.7	5.8	8.7	5.1	4.3	3.6	3.6	2.9	2.9	2.2	0.0
	0.1	20.3	13.0	8.7	10.1	7.2	6.5	5.1	5.1	3.6	4.3	3.6	0.0
Snns(138)	0.2	36.2	26.1	19.6	19.6	15.2	16.7	11.6	13.8	7.2	10.1	5.1	1.4
	0.3	46.4	39.1	29.0	29.0	21.7	22.5	18.1	20.3	10.1	15.9	8.0	1.4
	0.0	51.3	51.3	41.0	51.3	23.1	48.7	20.5	43.6	20.5	43.6	18.0	0.0
	0.1	51.3	51.3	46.2	51.3	41.0	51.3	41.0	51.3	38.5	51.3	38.5	35.9
Weltab(39)	0.2	66.7	64.1	61.5	64.1	56.4	64.1	53.8	61.5	51.3	59.0	46.2	43.6
	0.3	71.8	69.2	71.8	69.2	66.7	69.2	64.1	64.1	56.4	64.1	56.4	56.4
	0.0	14.0	9.6	6.0	7.0	3.8	5.0	2.9	3.8	2.4	3.3	3.3	0.9
	0.1	19.7	13.1	10.1	9.2	6.4	6.5	4.8	5.0	4.0	4.4	3.4	1.4
Linux(9491)	0.2	34.9	24.0	19.4	16.1	12.6	11.0	9.1	8.1	7.3	6.9	5.9	2.1
	0.3	49.6	38.1	31.0	26.0	21.5	18.0	15.7	13.3	12.1	10.7	9.5	3.1
	0.0	15.1	13.5	9.3	11.8	5.7	9.7	4.6	8.4	4.2	8.0	4.2	0.4
	0.1	18.6	15.8	12.2	13.1	9.8	11.2	8.8	10.4	7.6	9.8	7.7	5.7
Avg.	0.2	31.6	26.6	22.6	22.0	18.2	18.9	15.4	16.5	13.1	15.0	12.3	8.5
	0.3	43.2	38.4	33.9	31.8	27.1	27.0	23.6	23.5	19.6	21.2	15.6	11.1

0.3), whereas for the largest Java system, the *Swing*, it increases from 65.9% (for UPI threshold 0.0) to 76.1% (for UPI threshold 0.3).

4.3 Profiles of Cloning Density

While the subsection above provides an overall view of cloning over the files in a system, one cannot immediately see which files are highly cloned or which files contain the majority of clones. In this section we provide the values for the *CRFM* and *QFCM* related metrics. Tables 3 and 4 provide the data for the C and Java systems respectively. The first and second columns give the subject systems and different UPI thresholds, while the remaining columns show the corresponding *QFCMp(v)* (indicated with column *Meth*) and *QFCLOCp(v)* (indicated with column *LOC*) values. The last row of each table shows the average values of the metrics for each of the languages of the systems. When *v=0+%* or *v=100%* (the third and last columns) both metrics values are same. When *v=0+%* we get the percentage of files that are associated with at least one cloned method.

From the last rows of the tables with *v=10+%* we can see that on average 13.5% (9.3% LOC) of the files for C systems have 10% or more of their content as exact (UPI threshold 0.0) cloned methods. For Java systems this is even higher, at 37.8% (26.2% LOC). When we increase the UPI threshold (say to 0.3) the Java systems are still the winners, at 54.3% (44.9% LOC) compared to 38.4% (33.9% LOC) for the C systems both in terms of methods and LOC. However, for higher values of v, say 50+% with UPI threshold 0.3, both C and Java systems tend to have about the same percentage, at 21.2% (15.6% LOC) for C systems and 22.2% (16.7% LOC) for Java systems. One C system, *Weltab*, even has 64.1% (56.4% LOC) when *v=50+%* and 56.4% (still 56.4% LOC) when *v=100%*, once again indicating its high density of cloned code across different files. These tables show only the overall percentage for the systems. Detailed results for each file of each system can be found in our online respository [22] as an XML database.

We have also looked at the copy/paste patterns of the detected clones. Figure 4 shows examples of copy/paste changes from *Weltab*, *snns* and *Postgresql*. Assuming that cloned methods in high density cloned files have been intentionally copy/pasted, we have also compared copy/paste patterns between the high density files of the systems.

Table 4. Percentage of files that have clones over a certain percentage for Java systems

System(#File)	UPIT	v=0+%	v = 10+%		v = 20+%		v = 30+%		v = 40+%		v = 50+%		v=100%
		Both	Meth	LOC	Meth	LOC	Meth	LOC	Meth	LOC	Meth	LOC	Both
	0.0	45.3	37.3	17.4	26.1	5.5	14.3	3.1	8.1	2.5	4.4	3.0	0.6
	0.1	47.2	37.9	19.3	26.1	6.2	14.3	3.1	8.1	2.5	4.4	2.5	0.6
Ant(161)	0.2	49.7	39.1	20.5	27.2	7.5	15.3	5.0	9.9	3.7	6.2	3.1	0.6
	0.3	57.8	50.9	31.1	37.9	17.4	20.5	10.6	14.3	8.1	9.9	6.8	2.5
	0.0	44.4	42.6	27.8	27.8	20.4	24.1	13.0	18.5	9.3	13.0	5.6	1.9
	0.1	44.4	42.6	27.8	27.8	20.4	24.1	13.0	18.5	9.3	13.0	5.6	1.9
EIRC(54)	0.2	46.3	44.4	29.6	29.6	22.2	25.9	14.8	20.4	11.1	13.0	7.4	1.9
	0.3	51.9	51.9	38.9	38.9	27.8	33.3	20.4	24.1	14.8	14.8	9.3	1.9
	0.0	61.9	52.6	41.2	43.3	30.9	36.1	21.7	23.7	13.4	18.6	10.3	3.1
	0.1	61.9	52.6	42.3	44.3	32.0	36.1	23.7	23.7	15.5	18.6	10.3	3.1
Javadoc(97)	0.2	66.0	57.7	51.6	50.5	34.0	44.3	29.9	36.1	23.7	29.9	19.6	9.3
	0.3	68.0	61.9	56.7	54.6	45.4	48.5	38.1	40.2	30.9	37.1	26.8	13.4
	0.0	51.5	44.2	30.4	32.8	23.0	24.2	16.8	17.7	11.7	14.4	9.8	1.7
	0.1	57.7	50.0	36.4	38.0	28.4	27.1	21.8	22.0	16.3	18.6	14.4	3.3
Jdtcore(582)	0.2	66.7	59.3	48.6	49.3	38.7	36.3	29.9	29.6	24.4	24.4	21.1	4.3
	0.3	74.2	68.2	58.4	57.2	46.7	46.2	37.3	36.1	30.6	30.4	25.1	6.5
	0.0	44.6	36.9	32.2	27.5	18.9	19.3	13.3	14.2	9.4	11.6	6.4	2.1
	0.1	45.5	38.2	33.5	28.3	21.0	21.0	15.0	15.9	11.2	12.9	7.7	2.6
JHotDraw(233)	0.2	51.5	45.9	42.9	36.9	29.6	27.9	22.7	20.2	15.9	17.6	12.0	3.0
	0.3	57.5	53.6	51.1	45.1	39.5	36.9	30.9	27.5	22.7	21.9	17.6	5.6
	0.0	6.0	4.0	2.0	2.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	0.1	16.0	14.0	12.0	12.0	6.0	12.0	4.0	10.0	0.0	10.0	0.0	0.0
Spule(50)	0.2	16.0	14.0	12.0	12.0	8.0	10.0	4.0	10.0	0.0	10.0	0.0	0.0
	0.3	30.0	28.0	24.0	24.0	20.0	20.0	14.0	18.0	8.0	18.0	8.0	2.2
	0.0	65.9	47.3	32.6	32.1	23.4	24.9	17.4	19.1	15.9	17.1	13.8	2.2
	0.1	67.6	49.8	35.5	33.3	25.1	25.6	19.3	19.3	17.6	17.4	15.9	2.2
Swing(414)	0.2	71.5	56.8	43.0	40.1	32.1	29.0	24.6	24.2	21.5	20.5	19.8	2.9
	0.3	76.1	65.9	54.1	51.0	40.3	33.8	31.6	28.3	25.8	23.2	23.2	3.6
	0.0	45.7	37.8	26.2	27.4	17.4	20.4	12.2	14.5	8.9	11.3	7.0	1.7
	0.1	48.6	40.7	29.5	30.0	19.9	22.9	14.3	16.8	10.3	13.6	8.1	2.0
Avg.	0.2	52.5	45.3	35.5	35.1	24.6	27.0	18.7	21.5	14.3	17.4	11.9	3.1
	0.3	59.4	54.3	44.9	44.1	33.9	34.2	26.1	26.9	20.1	22.2	16.7	5.1

From the above discussion it is obvious that one of the interesting systems is *Weltab*. Using the NICAD interface we noticed that although there is no file in *Weltab* that is exactly similar to another, 14 files of its 39 are clones of each other even with UPI threshold 0.05, that is, they are the same but for very minor edits. As an example, in the three files *vedt.c*, *vfix.c* and *xfix.c* (each of which has two functions including one large *main* function) there are only minor differences in the *main* function, and no differences at all in the other (*acknowledge*) function. Most of the changes are in *fprintf* statements and the parameters of the *acknowledge* function when it is called, and there are also lines added/deleted in the *main* function. For other high density files we have noticed similar changes, including changes in *if-statements*, the names of functions, and so on. Java systems show similar behavior but the methods are most cases smaller in size.

4.4 Profiles of Cloning Localization

In this section we provide the $PCLMp$ related metrics. The location of a clone pair is a factor in software maintainence [15, 26]. A code fragment can form a clone pair with another fragment within the same file, or it can form a clone pair with another fragment of a different file located in the same directory or with a code fragment that is located in a different file in a different directory. Kapser and Godfrey [15, 14] provide a categorization of function clones based on such location differences and analyze the causes, usefulness, harmfulness and possible solutions for each kind of cloning. For example, clone pairs that appear in the same file may not be harmful as they are not physically apart and might be easily maintainable. On the other hand, clone pairs that appear in different directories, might be harmful to software maintenance as the similar fragments are hard to find and thus might not easily maintainable. They also provide cloning statistics of such clone types on the Linux Kernel *file-system subsystem version 2.4.19* and *Postgresql 7.4.2*. We further extend the similar study with more than 15 C and Java systems including entire Linux Kernel with our new hybrid clone detection tool.

In Table 5 we provide the percentage clone pairs for each of the different categories for the C systems. Table 6 shows similar values for Java systems. For each of the systems the first row represents *PCLMp* metric (i.e., w.r.t no. of methods) and the second row represents *PCLLOCp* metric (i.e.,

```
---------------Weltab ---------------
 <   fprintf (stderr, "*** Skipping VFIX. Run canceled.\n");
 ---
 >   fprintf (stderr, "*** Skipping XFIX. Run canceled.\n");
 ---------------Weltab ---------------
 <   askchange ("", &change, &quit, FALSE, FALSE);
 ---
 >   askchange ("", &change, &quit, FALSE, TRUE);
 ---------------Weltab ---------------          ---------------Weltab ---------------
 <   lines = LANDSCAPE;                           <  if (lines+2 > LANDSCAPE)
 ---                                            ---
 >   lines = 60;                                  >  if (lines+2 > 60)
 ----------------snns ----------------          -------------Postgresql -------------
 <   yy_is_jam = (yy_current_state == 144);     < char *scanstr (char *s){
 ---                                            ---
 >   yy_is_jam = (yy_current_state == 26);      > char *GUC_scanstr (char *s){
 -------------------------------------          -------------------------------------
```

Figure 4. Some Copy/paste Change Examples

w.r.t LOC). For each of the metrics four different values are shown corresponding to UPI thresholds ranging from 0.0 (exact clone) to 0.30 (relaxed near-miss clone).

From the tables we see that while there are no exact clones (UPI threshold 0.0) within the same file for C systems (except in the Linux Kernel), there are on average 18.7% (17.6% LOC) exact clone pairs within the same files for Java systems. This is particularly surprising in the *Spule* system. Out of 60 exact clone pairs, 96.7% (97.2% LOC) of clone pairs occur within the same files, more specifically in file *spule/src/common/Messages.java*. Cloned methods occur between the static classes in the files. For example, two *write* methods of 11 LOC and 7 LOC appear two times (lines 1659-1669 and 1612-1622) and seven times (lines 1796-1802, 1687-1693, 1533-1539, 1432-1438, 1391-1397, 1349-1355, and 1268-1274) respectively in file *Messages.java*. Similarly, exact clone pairs appear within the same file in other Java systems between original and abstract classes (e.g., *ant/BuildEvent.java* lines 163-165 and 93-95), as inlined functions (e.g., *jdt-core/src/internal/compiler/Compiler.java* lines 148-151 and 84-87) and so on. While most of the clone pairs are smaller in size, exact cloning within the same file might be interesting both from software maintenance and language design points of view. As an exception to the C systems, the Linux Kernel has one file (*/drivers/net/fec.c*) that has six exact clone pairs in it. In fact there are two exact clone classes in this file. In the first class, function fec_get_mac (32 LOC) appears three times (lines 1744-1775, 1595-1626 and 1464-1495) and in the second class, function fec_set_mii (16 LOC) also appears three times (lines 1727-1742, 1577-1593 and 1446-1462) as static inline functions.

However, when we detect near-miss clones by allowing a higher UPI threshold, we see that the metrics values increase to a higher ratio for the C systems than the Java systems. For example, when UPI threshold is 0.3, on average 45.9% (49.0% LOC) of clone pairs of the C systems occur within the same file compared to only 25.3% (29.7% LOC) of the clone pairs in the Java systems. If we have a close look at the individual systems we also see higher ratios for most of the C systems than the Java systems. In the case of *Wget* it actually reaches 90.9% (92.7% LOC). However, *Wget* is a small system, has only 11 clone pairs in 2 classes with UPI threshold 0.3, and only 3 of its 21 files are associated with clones. Taking a closer look, we found that of the 2 clone classes, one contains 5 cloned methods (lines 503-539, 463-499, 389-425, 350-385, and 428-460) and all are from the same file *wget/src/ftp-basic.c*.

However, a reasonably large system, *Postgresql* (530 clone pairs in 203 classes with UPI threshold 0.3), also shows a higher percentage (87.6% methods and 86.1% LOC) of clone pairs within the same files with UPI threshold 0.3. As with *Wget*, it has also higher frequency of clone classes in the same file. For example, in file *postgresql/src/backend/utils/adt/float.c*, there are six similar methods (lines 952-965, 935-948, 900-913, 865-878, 848-861, and 831-844) of 14 LOC differing only in their function names and built-in function calls (e.g., *tan* changes to *sin*, *cos* to *acos*, and so on).

Both C and Java systems tend to have a higher percentage of exact clones (UPI threshold 0.0) within the same directory but in different files. Even the largest C system, the Linux Kernel, has 55.5% (58.9% LOC) of its exact clone pairs in the same directory (but different files). The largest Java system, the Java 2 SDK *Swing*, has even more, 87.5% (90.0% LOC). When we look for near-miss clones by increasing the UPI threshold, these percentages seem to decrease, indicating that clone pairs either form within the same file or between different files of different directories. C systems, however, show a higher percentage of different directory exact clone pairs than Java systems. When we increase the UPI thresholds, these percentages tend to decrease, and at the same time the percentages for the same file clone pairs tend to increase, indicating that near-miss

Table 5. Percentage localization of clone pairs for C Systems

		Same File, Same Directory				Same Directory, Different File				Different Directory			
System	UPIT	0.0	0.10	0.20	0.30	0.0	0.10	0.20	0.30	0.0	0.10	0.20	0.30
Cook	Method	0.0	16.7	40.2	29.3	42.9	61.1	42.1	57.8	57.1	22.2	17.7	12.8
	LOC	0.0	32.3	51.3	41.1	27.3	59.8	35.9	49.1	72.7	7.8	12.7	9.8
Httpd	Method	0.0	2.7	15.2	33.5	6.6	7.1	9.3	7.3	93.4	90.2	75.4	59.2
	LOC	0.0	7.3	18.6	34.6	6.0	6.9	10.7	7.9	94.0	85.8	70.7	57.4
Postgresql	Method	0.0	62.5	89.7	87.6	14.3	4.2	2.5	6.9	85.7	33.3	7.6	5.5
	LOC	0.0	81.3	89.5	86.1	5.3	0.4	4.7	8.7	94.7	18.3	5.7	5.2
Snns	Method	0.0	15.9	45.2	54.3	62.4	55.4	37.9	33.3	37.6	28.6	16.9	12.3
	LOC	0.0	33.8	50.4	59.8	48.7	38.3	32.7	27.6	51.3	27.9	16.8	12.5
Weltab	Method	0.0	0.0	2.7	3.1	100.0	100.0	97.3	96.9	0.0	0.0	0.0	0.0
	LOC	0.0	0.0	0.4	0.5	100.0	100.0	99.5	99.5	0.0	0.0	0.0	0.0
Wget	Method	0.0	50.0	75.0	90.9	0.0	50.0	25.0	9.1	0.0	0.0	0.0	0.0
	LOC	0.0	56.5	79.6	92.7	0.0	43.5	20.4	7.3	0.0	0.0	0.0	0.0
Linux	Method	0.1	2.7	12.2	22.5	55.5	53.3	41.2	33.5	44.4	44.1	46.6	44.0
	LOC	0.3	8.6	20.2	28.0	58.9	52.5	40.0	34.7	40.8	39.0	39.8	37.2
Avg.	Method	0.0	21.5	40.0	45.9	40.2	47.3	36.5	35.0	45.5	31.2	23.5	19.1
	LOC	0.0	31.4	44.3	49.0	35.2	43.1	34.8	33.5	50.5	25.5	20.8	17.4

Table 6. Percentage localization of clone pairs for Java Systems

		Same File, Same Directory				Same Directory, Different File				Different Directory			
System	UPIT	0.0	0.10	0.20	0.30	0.0	0.10	0.20	0.30	0.0	0.10	0.20	0.30
Ant	Method	4.4	4.7	5.1	6.6	73.0	72.6	72.5	69.3	22.6	22.7	22.5	24.2
	LOC	4.2	5.2	6.1	11.6	75.2	73.3	72.3	62.6	20.6	21.5	21.6	25.8
EIRC	Method	0.0	0.0	0.8	5.4	65.0	65.0	65.3	63.1	35.0	35.0	33.9	31.5
	LOC	0.0	0.0	2.1	9.1	68.3	68.3	68.3	60.8	31.7	31.7	29.6	30.1
javadoc	Method	19.7	20.8	19.6	22.7	72.0	71.1	69.2	62.8	8.3	8.1	11.3	15.5
	LOC	14.3	22.5	25.8	25.8	76.3	69.4	63.4	60.2	9.4	8.0	10.8	14.1
Jdtcore	Method	0.9	3.1	11.8	41.9	75.8	72.9	68.9	46.4	23.3	24.0	19.3	11.7
	LOC	0.6	13.0	34.3	59.4	73.9	56.5	42.0	28.2	22.5	30.5	21.6	12.4
JHotDraw	Method	5.8	5.8	6.6	13.7	51.6	52.2	47.5	39.1	42.6	42.0	45.9	47.2
	LOC	4.5	4.2	7.5	18.7	50.5	53.3	46.0	36.6	45.0	42.5	46.6	44.6
Spule	Method	96.7	90.6	91.2	77.0	3.3	4.0	8.8	23.0	0.0	0.0	0.0	0.0
	LOC	97.2	86.1	87.3	64.6	2.9	13.9	12.7	35.4	0.0	0.0	0.0	0.0
Swing	Method	3.5	3.7	5.3	9.6	87.5	87.2	83.8	77.8	9.0	9.1	10.9	12.6
	LOC	2.4	3.2	8.3	18.5	90.0	88.5	81.0	68.7	7.6	8.3	10.7	12.9
Avg.	Method	18.7	18.4	20.1	25.3	61.2	60.7	59.4	54.5	20.1	20.1	20.5	20.4
	LOC	17.6	19.2	24.5	29.7	62.4	60.5	55.1	50.4	19.5	20.4	20.1	20.0

clone pairs tend to occur more within the same file than in different files of the same or different directories.

Since we consider all functions with 3 LOC or more in pretty-printed format, one might argue that the findings are biased on the size of the functions. However, it does not seem so when we look the values for LOC. In almost all cases, the metrics values are very close, showing that a significant proportion of files in each system have a significant proportion of similar code in the files themselves.

5. Related Work

Empirical study of clones in open source systems is not a new topic. When a new clone detection technique is published, it normally comes with an empirical study (at least in part). However, these studies focus on validating the proposed methods [25] rather than on the subject systems.

Several tool comparison studies have used open source systems for comparing different tools [25]. Of them, the Bellon et al. experiment [6, 5] is the most extensive to date, using four C and four Java systems to compare several state-of-the-art tools. Although we have used the subject systems from Bellon's experiment as a part of our study, our study differs in using a new hybrid clone detection tool, in the size of the subject systems analyzed (e.g., the entire Linux Kernel) and in providing the cloning status of the subject systems themselves in several different dimensions.

Kapser and Godfrey have conducted extensive empirical studies with Apache *httpd*, the Linux file system and several other open source systems. They provide a detailed categorization of code clones in the form of a taxonomy [15], propose a new analysis framework [14] and give an in-depth study on the harmfulness / usefulness of cloning [13]. Our study differs in that we focus on the comprehensive cloning status of a wide variety of different systems in different languages, whereas they focus on the maintenance implications of cloning.

Empirical studies of cloning in the Linux Kernel have also been carried out by several other researchers. Of them, Casazza et al. [7] and Antoniol et al. [4] provide interesting findings, but they focus on clone evolution, whereas we focus on the occurrence of copy/paste clones in several dimensions. Kim et al. [17] also studied the evolution of code clones in several systems and concluded that programmers often intentionally practice code cloning. Jiang and Hassan [10] also used the Linux Kernel as an example for their framework for understanding cloning in large systems.

Al-Ekram et al. [3] have also conducted a promising empirical study on cloning, focussing on C/C++ systems from two different domains. They examined different clone types (e.g., accidental clones) by analyzing clones across systems in the same domain, whereas we have studied a wide variety of systems and concentrated on copy/paste function clones of individual system. Krinke [18] has conducted an empirical study with five C/C++/Java systems, focussing on consistent and inconsistent changes to exact code clones in different versions of the subject systems. The most closely related work to ours is the work of Rajapakse and Jarzabek [20] which was also one of the motivations of our study. However, they studied cloning in a different domain, web applications, and have looked at only exact clones.

6. Conclusion

In this paper we have provided an empirical study of function clones in several C and Java open source software systems of varying size, including Apache *httpd* and the entire Linux Kernel, using the new hybrid clone detection method NICAD. The study has demonstrated that NICAD is capable of accurately finding both exact and near-miss function clones even in large systems and different languages, and that there seem to be a large number of copy/paste function clones in those systems. We have provided cloning statistics for these systems in several different dimensions and made the detailed results available in an online repository. These results can potentially be used as a benchmark for evaluating other clone detection tools.

Threats to Validity: One of the major threats to the results of this study is the lack of a sound definition of code clones. While one can precisely define exact clones, there is no agreed upon definition of near-miss clones. In this study we have used a dissimilarity threshold on the standard pretty-printed code as a measure of near-miss clones. While this gives good results, we cannot be sure that these are definitively the right set for software maintenance activities such as refactoring.

Future Work: We plan to repeat this study on systems in several other languages, for example C++ and C#, and using smaller clone granularities such as *begin-end* blocks. We also plan to explore the effect of more advanced NICAD features, such as flexible pretty-printing, code normalization and filtering, on the results.

Acknowledgements: The authors would like to thank the four anonymous reviewers for their valuable comments, suggestions and corrections in improving the paper. This work is supported in part by the Natural Sciences and Engineering Research Council of Canada.

References

[1] The Abyss: `http://abyss.sourceforge.net/` (Dec 2007)

[2] The Apache-httpd: `http://httpd.apache.org/` (April 2008)

[3] R. Al-Ekram, C. Kapser and M. Godfrey. Cloning by Accident: An Empirical Study of Source Code Cloning Across Software Systems. In *ISESE*, pp. 376-385, 2005.

[4] G. Antoniol, U. Villano, E. Merlo and M.D. Penta. Analyzing Cloning Evolution in the Linux Kernel. *Information and Software Technology*, 44 (13):755-765, 2002.

[5] S. Bellon and R. Koschke. Detection of Software Clone: Tool Comparison Experiment: `http://www.bauhaus-stuttgart.de/clones/` (December 2007).

[6] S. Bellon, R. Koschke, G. Antoniol, J. Krinke and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE TSE*, 33(9):577-591, 2007.

[7] G. Casazza, G. Antoniol, U. Villano, E. Merlo and M. Penta. Identifying Clones in the Linux Kernel. In *SCAM*, pp. 90-97, 2001.

[8] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[9] The Gzip-1.2.4 `http://www.gzip.org/` (Feb 2008).

[10] Z. Jiang and A. Hassan. A Framework for Studying Clones in Large Software Systems. In *SCAM*, pp. 203-212, 2007.

[11] The JHotDraw: `http://www.jhotdraw.org/` (June 2006)

[12] J. Johnson. Visualizing Textual Redundancy in Legacy Source. In *CASCON*, pp. 171-183, 1994.

[13] C. Kapser and M. Godfrey. "Cloning Considered Harmful" Considered Harmful. In *WCRE*, pp. 19-28, 2006.

[14] C. Kapser and M. Godfrey. Supporting the Analysis of Clones in Software Systems: A Case Study. *JSME: Research and Practice*, 18(2):61-82, 2006.

[15] C. Kapser and M. Godfrey. Toward a Taxonomy of Clones in Source Code: A Case Study. In *ELISA*, pp. 67-78, 2003.

[16] The Linux-2.6.24.2: `http://www.linux.org/` (March 2008)

[17] M. Kim and G. Murphy. An Empirical Study of Code Clone Genealogies. In *FSE*, pp. 187-196, 2005.

[18] J. Krinke. A Study of Consistent and Inconsistent Changes to Code Clones. In *WCRE*, pp. 170-178, 2007.

[19] Z. Li, S. Lu, S. Myagmar and Y. Zhou. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE TSE*, 32(3):176-192, 2006.

[20] D. C. Rajapakse and S. Jarzabek. An Investigation of Cloning in Web Applications. In *WWW*, pp. 924-925, 2005.

[21] M. Rieger, S. Ducasse and M. Lanza. Insights into System–Wide Code Duplication. In *WCRE*, pp. 100-109, 2004.

[22] The Roy/Cordy WCRE'08 Clone Results: `http://www.cs.queensu.ca/home/stl/download/NICADOutput/`.

[23] C.K. Roy and J.R. Cordy. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *ICPC*, pp. 172-181, 2008.

[24] C.K. Roy and J.R. Cordy. Scenario-Based Comparison of Clone Detection Techniques. In *ICPC*, pp. 153-162, 2008.

[25] C.K. Roy and J.R. Cordy. *A Survey on Software Clone Detection Research*. Queen's School of Computing TR 2007-541, 115 pp., 2007.

[26] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. Gemini: Maintenance Support Environment Based on Code Clone Analysis. In *METRICS*, pp. 67-76, 2002.