Are Scripting Languages Really Different?

Chanchal K. Roy
Department of Computer Science
University of Saskatchewan
Saskatoon, SK, Canada S7N 5C9
croy@cs.usask.ca

James R. Cordy
School of Computing
Queen's University
Kingston, ON, Canada K7L 3N6
cordy@cs.queensu.ca

ABSTRACT

Scripting languages such as Python, Perl, Ruby and PHP are increasingly important in new software systems as web technology becomes a dominant force. These languages are often spoken of as having different properties, in particular with respect to cloning, and the question arises whether the observations made based on traditional languages also apply to them. In this paper we present a first experiment in measuring the cloning properties of open source software systems written in the Python scripting language using the NiCad clone detector. We compare our results for Python with previous observations of C, C#, and Java, and discover that perhaps scripting languages are not so different after all.

Categories and Subject Descriptors

D.2.7 [**Distribution, Maintenance and Enhancement**]: Restructuring, reverse engineering, and reengineering

General Terms

Languages, Measurement, Experimentation

Keywords

code clones, empirical study, scripting languages, Python

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 code fragments in them. Such duplicated fragments are called *code clones* or simply *clones*. Although cloning is at times beneficial [3, 8] and often programmers intentionally use it [10], it can be detrimental to software maintenance [7]. In response, many techniques and tools for detecting code clones have been proposed [20, 21].

While there have been many empirical studies on cloning to validate and compare tools [4, 5, 20, 21], and to study maintenance impact [3, 8, 10, 11], taxonomies [9] and evolution [2] of clones, until recently little work has looked directly at the cloning properties of the systems and languages themselves. In our work we focus directly on this problem. In our first study we compared the cloning characteristics of open source software systems written in C, Java and C# [16]. In this paper we extend that work to look at a completely different kind of language - the scripting language Python. While there has been one in-depth study of exact clones in web applications [13], this is to our knowledge the first study of the cloning properties of a scripting language itself.

Scripting languages, those whose primary purpose is the rapid development of code to support the generation or coordination of other processes and technologies, are increasingly widely used due to the rapid growth of web applications and web services. The design of scripting languages, which includes much higher level data structures and programming concepts than traditional programming languages, suggests that there may be fewer clones in systems written using them, since repeated operations such as string and list manipulations, which are a rich source of clones in traditional languages, are often built-in to scripting languages. On the other hand, the rapid results-oriented development methods normally used with scripting languages might actually lead to more clones, since scripting programmers tend to work more quickly.

In order to help answer these questions, in this paper we provide an in-depth study of function clones in eight open source applications written in the Python scripting language. The applications range from relatively small (*sct*, at 9 KLOC) to some of the largest systems written in Python (*Eric*, at 99 KLOC and *Zope*, at 272 KLOC), and represent a variety of systems typical of the kinds of applications scripting languages are used for, including an IDE, a software build tool, a content management system, a web application server, a wiki server and a network simulation tool.

We provide an in-depth empirical study of function clones in these Python systems and compare the results to the more than twenty open source C, Java and C# systems, including the entire Linux Kernel, Apache httpd, J2SDK-Swing and db4o, that we have previously studied [16]. We use the NiCad clone detector [19], which has been shown to be highly accurate both with respect to precision [19] and recall [22] in the detection of copy/pasted near-miss clones. Using the web-based clone class user interface provided by NiCad, we manually verify all detected clones in this study. The adaptation of NiCad to scripting languages, and Python in particular, forms a part of the contribution of this paper. To our knowledge there is only one other language-sensitive clone detection method [6] designed to detect clones in Python systems.

We compare the cloning characteristics of our new results for Python with those for C, Java and C#, as well as comparing all of the languages and systems with respect to clone localization across subsystems. In particular, we focus on three research questions:

1. What is the cloning status of open source Python systems? Are there many clones? Are there more near-miss than exact clones?
2. Are there significant differences in cloning between Python and the other languages used in open source systems?

IWSC2010 May 8, 2010, Cape Town.

3. Are there significant differences in cloning in large Python systems compared to medium and small-sized 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, with up to 30% (three lines in ten) of editing changes.

The rest of the paper is organized as follows. Following the experimental setup of our study in Section 2, we present and analyze our findings in Section 3. Section 4 considers other empirical studies and their relation to ours. Finally, Section 5 concludes the paper.

2. EXPERIMENTAL SETUP

In this experiment we applied NiCad to find function clones in a number of open source Python systems, and compared with the results of our previous study of function clones in open source C, Java and C# systems [16]. As in our earlier study, we have used a set of metrics to analyze and compare the results. This section introduces the systems we have studied and the metrics used, including a brief overview of our clone definition and methodology for manual verification of detected clones.

2.1 Subject Systems

In this study we have analyzed eight Python systems ranging from 9K to 272K lines of code (LOC), and compared them with ten C, seven Java and six C# systems varying in size from 4K LOC to 6265K LOC including the entire Linux Kernel. The open source Python systems [12] include *Eric*, a full featured Python and Ruby IDE based on the Qt toolkit (99K LOC, 7440 functions); *EricPlugins*, a suite of plugins for the Eric IDE (30K LOC, 2983 functions); *Moin*, a Python-based Wiki system (74K LOC, 5820 functions); *NetworkX*, a simulation toolkit for complex networks (15K LOC, 890 functions); *Plone*, a popular web-based content management system (CMS) (74K LOC, 6877 functions); *scons*, a software build system similar to Linux "make" (52K LOC, 3967 functions); *Zope*, a complete web application server system (272K LOC, 26985 functions), and *sct*, a collection of Django applications in Python (9K LOC, 865 functions).

The C, Java and C# systems we compare to include all of those from Bellon's experiment [4], as well as Apache *httpd*, *JHotDraw*, and the entire Linux 2.6 Kernel, as well as six C# systems of different sizes and kinds, including *db4o*, a popular and widely used production commercial and open source object database. Since the Linux kernel is almost two orders of magnitude larger than any of the other systems, we have treated it as an outlier, and we provide statistical results both including and not including Linux in Section 3.1, and in later subsections we have dropped Linux from averages to avoid any bias due to its exceptionally large size.

2.2 Clone Definition

In this study we have considered all non-empty functions of at least 3 lines 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 size-sensitive UPI (Unique Percentage of Items) thresholds [19] to find exact and near-miss (copy/paste/modify) function clones. For example, if the UPI threshold (UPIT) is 0.0, we detect only exact clones; if the UPIT is 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). In this study we used the representative set of UPI thresholds 0.0, 0.10, 0.20 and 0.30, corresponding to editing changes of from 0 to 30%, or 0 to 3 lines in every ten.

2.3 Validation of Clones

All clones detected in this study were validated by hand using NiCad's interactive HTML web page output to give an overall view of the original source of the clone classes, followed by a pairwise comparison of the original source of the functions in each clone class using Linux *diff*, manually examining those with a greater difference than the UPI trheshold. Manual validation using this method is both easy and efficient [16].

2.4 Metrics and Visualizations

This subsection describes the different metrics and visualizations that we have used in this study. These are the same metrics that we have used in our previous study [16] which were either adapted or reused from previous studies of cloning [13, 1, 9, 24, 14].

Total Cloned Methods (TCM): In this study we focus on function clones, and thus our first metric is related to the number of functions/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 with respect to 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 define similar metrics with respect to number of lines. We define *TCSppLOC* as the total number of cloned lines in a system for a given UPI threshold and *TCSppLOCp* as the percentage of total cloned lines in the system for a given UPI threshold. Since we apply standard pretty-printing before clone detection, which eliminates formatting and layout differences, resolves *#ifdefs* (in C systems) and ignores comments, we can get a more accurate percentage of cloned lines, and thus we use pretty-printed rather than raw lines. In practice there is little difference.

File Associated with Clones (FAWC): While the above metrics give the overall cloning statistics for a subject system, they can't tell us whether the clones are from some specific files, or scattered among many files all over the system. *FAWC* provides 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 a different file. We define *FAWCp* as the percentage of files associated with clones in a system for a given threshold. For example, "*FAWCp* of a system 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 maintenance point of view, a lower value of *FAWCp* is better, since clones localized to certain specific files 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 with another method in the system (either in the same file or a different file) 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 with respect to 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 [13] and the *RSA* metric of Ueda et al. [24], although they ignore clone pairs in the same file.

With *CRFM* we can identify the highly cloned files of a system and can potentially help predict maintenance difficulty. For example, consider two systems x and y of similar size and with 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 concentrated into a certain set of files. From a clone treatment perspective, system y is more interesting than x because clones in y might be more easily handled than those of x.

Qualifying File Count for Methods (QFCM): As in Rajapakse and Jarzabek [13] 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 [9] 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, and usefulness / harmfulness for each of these categories [9]. 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 in category i. $PCLMp(i)$, the percentage of clone pairs in 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*), and over a range of UPI thresholds.

Profiles of Remote Cloning Locality w.r.t Methods (PRCLM): In order to gain further insight into cloning locality, we define three additional metrics for remote clone pairs (those in different directories). Two fragments of a clone pair that are neither within the same file nor under the same directory (i.e., do not have the same parent), might have the same grandparent directory (category 1), or they might be in the same subsystem (category 2), or in the worst case, might be in a different subsystem (category 3). Similarly to the above definition, we define metrics for the three categories, $PRCLM(1)$ for category 1, $PRCLM(2)$ for category 2 and $PRCLM(3)$ for category 3 where $PRCLM(i)$ gives the total number of clone pairs in category i. $PRCLMp(i)$, the percentage of clone pairs in category i is defined as follows:

$$PRCLMp(i) = \frac{PRCLM(i) * 100}{Total\ no.\ of\ diff\ dir\ clone\ pairs\ in\ the\ system}$$

As usual similar metrics are defined with respect to lines of code (*PRCLLOC and PRCLLOCp*) and over a range of UPI thresholds.

3. COMPARISON OF RESULTS

In this section we provide a detailed comparison of the cloning properties of open source systems written in Python with the cloning properties of several other open source systems written in traditional languages such as C, C# and Java. We provide the comparison results starting from overall cloning level in Python, C, Java and C# systems and then for each individual system in a variety of measures based on the metrics described in Section 2.4. While we provide here only the overall findings and statistical measures, the detailed results and the raw data for each of the systems using different UPI thresholds can be found in an online repository [17] as XML databases and an HTML website. Due to limited space, we have not shown all of the detailed results for the C, C# and Java systems, and have omitted results for UPI thresholds 0.1 and 0.2 in some tables. Detailed statistics for all systems and all thresholds for the other languages can be found elsewhere [16, 15].

3.1 Overall Cloning Level

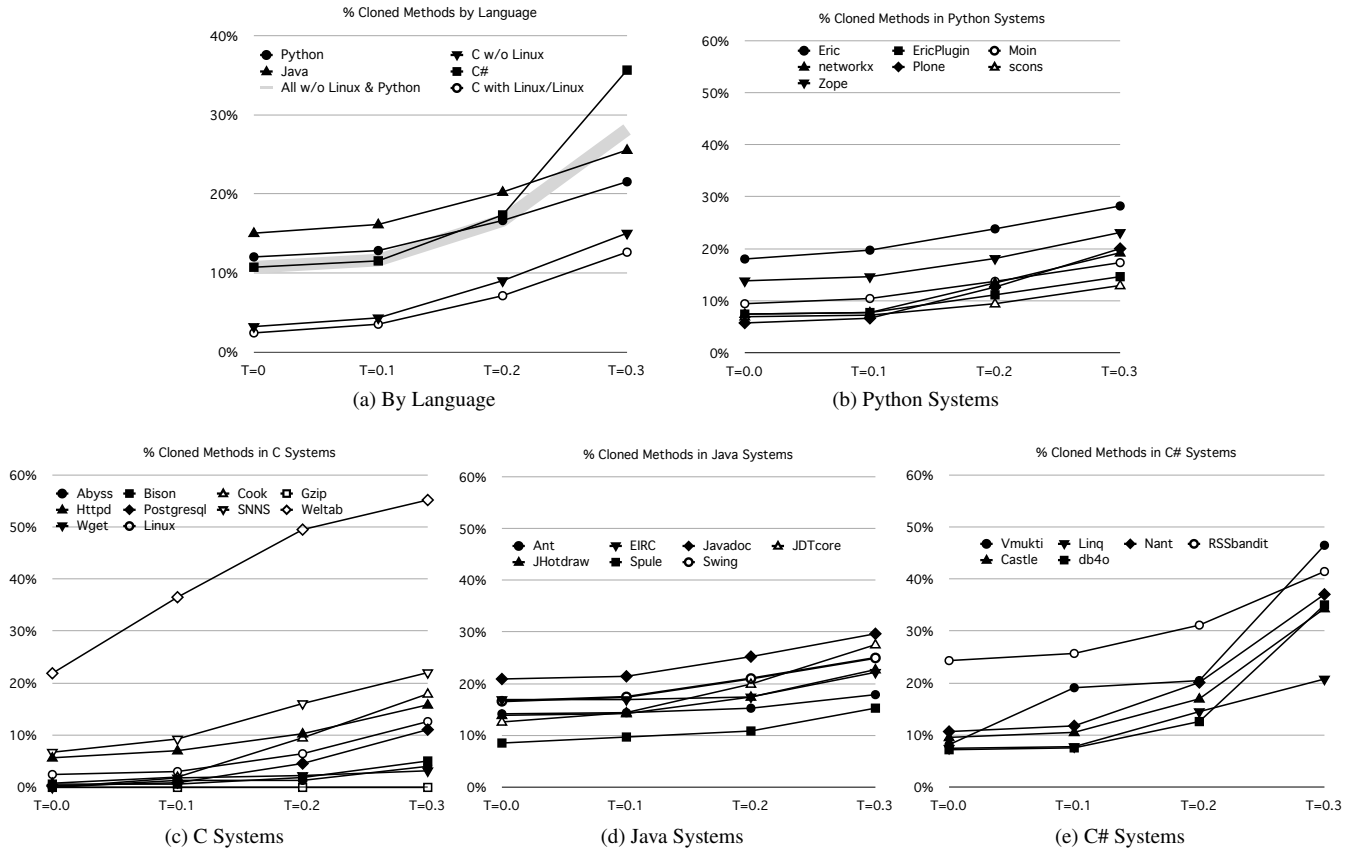
We begin by looking at the overall cloning level, both in terms of number of methods and in terms of number of pretty-printed LOC (i.e., the values of the *TCM*-related metrics of subsection 2.4). Figure 1a summarizes our results for the Python, C, Java and C# systems by the proportion of functions (or methods in the case of Java) that are cloned (i.e., *TCMp* over languages). The corresponding values for the *TCSppLOCp* metric (i.e., the proportion of clones by number of pretty-printed LOC for each language) can be found in the *%Total* rows of Table 1.

Our detailed comparison of the C, C# and Java systems has been presented elsewhere [16]. Here we are interested in comparing the results for Python systems with systems in other languages. The first thing we notice (Figure 1a, Table 1) is that the percentage of function clones in open source Python systems is close to the overall average for the open source systems written in C, C# and Java. For exact clones (with UPI threshold 0.0), the cloning percentage for Python systems is even higher than the average. On average, 12.1% (7.4% LOC) of functions in these Python systems are exact clones - those with no changes at all (except changes in formatting, whitespace and comments), whereas about 10.8% (4.9% LOC) functions are exact clones for the rest of the systems.

The second thing we can notice in Figure 1a (and in the *% Total* rows of Table 1 for LOC) is that the effect of increasing the UPI threshold is almost identical across the languages, even for Python. We can interpret this as meaning that the numbers of small changes made to cloned functions in each of these languages is roughly the same in these systems, and that scripting languages are really no different. This is in some ways surprising - there is no particular reason why the pattern of changes to copied code should be similar across languages. Up to UPI threshold 0.2, Python systems have a higher or similar cloning rate to the rest of systems, at about 16.7% with UPI threshold 0.2. C# systems are very similar to Python up to that point. However, when we use UPI threshold 0.3 to detect relaxed near-miss clones, overall cloning in C# is much higher than in Python, and the highest among all the languages. But it is interesting to see that the cloning rate in Python systems is still much higher than C systems, including the Linux kernel.

Figures 1b, 1c, 1d and 1e (also columns 3 to 6 of Table 1) refine Figure 1a to show a detailed view of the same information for the individual open source Python, C, Java and C# systems respectively. As expected, the overall trends for each language are much like the summaries in Figure 1a (or the *% Total* rows of Table 1), with lower levels of cloning in C than Python, Java and C#.

Figures 1b and 1d (also Table 1 for LOC) are even more interesting, because while the C and C# systems vary, the Python and Java systems are remarkably consistent in their cloning characteristics. All Python and Java systems begin with a relatively high level of exact method clones (between 7 and 22 percent), and allowing for changes increases the proportion only modestly. This property seems to be independent of system size, and appears to be a characteristic of the language, leading us to believe that Python and Java may have similar cloning characteristics. The only exception seems to be the Java system *JDTcore*, with about twice as many near-miss clones at the 0.30 dissimilarity level than exact clones.



(a) By Language (b) Python Systems

(c) C Systems (d) Java Systems (e) C# Systems

Figure 1: Proportion of cloned functions / methods in the systems

In Table 1 (columns 7 to 14) we also provide the number of clone pairs and clone classes for most of the systems at varying UPI thresholds. It is interesting that the average number of clone pairs per clone class (the *%Total* row for each language) is more or less consistent for Python, C, C# and Java systems for different UPI thresholds. The only exception is C# systems at UPI threshold 0.3. which have a surprisingly high number (39.7 clone pairs per clone class), indicating that copy/paste cloning of functions with a significant amount of editing is frequent in the C# systems. What's interesting to us is that Python systems show a consistent ratio compared to the other systems, and that the average number of clone pairs per clone class in Python systems is even higher than in C systems. We also see no significant differences between individual systems in Python and other languages.

3.2 Clone Associated Files

The *FAWC* and *FAWCp* metrics of Section 2 address the issue of what proportion of the files in a system is associated with clones, that is, contain at least one cloned method. A system with more clones but associated with only a few files may be 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 and compare the Python systems with systems in C, Java and C#. Figure 2a shows the average values of *FAWCp* by language with varying UPI thresholds.

We see that on average 32.1% of files in the Python systems, 15% of the files in the C systems, 46% of the files in the Java systems, and 29% of files in the C# systems are associated with exact clones (i.e., with UPI threshold 0.0). These values simply indicate the fact that there are no fewer clones in Python systems than in C, C# and Java systems, rather it appears that more files in Python systems are associated with exact clones than C and C# systems. The higher percentage for Java systems can be explained by Java's many small similar accessor methods (see [16] for details).

Figures 2b, 2c, 2d and 2e refine Figure 2a to show a detailed view of the same information for the individual open source Python, C, Java and C# systems respectively. We see that while most of the C systems have lower and most of the Java systems have higher percentages at UPI threshold 0.0, Python and C# systems appear to stay in the middle and show very similar percentages (20% to 45%), except for the small C# system *Linq*.

When we increase the UPI threshold to detect near-miss clones, the C and C# systems show a faster growing ratio than the Python and Java systems, indicating the fact that there might be more near-miss clones in the C and C# systems than in Python and Java. What we see from these metrics is that Python systems are not outliers with respect to cloning, and possibly we should give similar priority to dealing with clones in Python and other scripting languages.

3.3 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, and observe what happens in Python systems compared to systems of other languages. Table 2 provides the data for the Python, C, Java and C# systems. The second and third 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 *Avg.* row of the table shows the average values of the

Table 1: Proportion of cloned LOC, clone pairs and clone classes

Lang	System	% Cloned LOC T=0.0	T=0.1	T=0.2	T=0.3	No. of Clone Pairs T=0.0	T=0.1	T=0.2	T=0.3	No. of Clone Classes T=0.0	T=0.1	T=0.2	T=0.3
C	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
	Linux	1.0	2.6	8.3	10.8	5953	7362	13813	25767	1505	2263	4613	7918
% Total C with Linux		1.1	2.8	8.4	11.0	**#CP per Clone Class**				3.7	2.9	2.8	3.2
% Total C w/o Linux		2.0	4.7	8.6	13.2	**#CP per Clone Class**				1.8	2.0	2.2	2.7
Java	Ant	5.1	5.4	6.3	9.7	363	365	374	426	92	94	101	119
	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
	Swing	9.4	11.0	15	19.4	8115	8203	9978	11209	516	558	687	843
% Total Java		7.2	9.4	14.4	20.0	**#CP per Clone Class**				8.8	8.3	8.2	8.6
C#	Linq	3.4	3.8	8.3	12.4	427	428	523	565	4	5	16	31
	Nant	3.7	5.8	12.8	21.6	2325	2341	3519	8554	45	57	110	192
	RSS	9.8	11.7	15.3	20.6	1657	1698	2240	6405	440	469	533	605
	Castle	5.4	7.6	15.5	28.4	2110	2311	5124	21351	347	380	585	981
	db4o	4.8	5.4	10.6	26.6	1109	1149	2289	82571	391	411	652	1187
% Total C#		6.0	7.6	13.3	24.9	**#CP per Clone Class**				6.2	6.0	7.2	39.7
% Total Overall w/o Python and Linux		4.9	7.1	11.9	19.1	**#CP per Clone Class**				7.1	6.7	7.0	23.9
Python	Eric	10.5	14.4	20.1	25.0	1121	1220	1577	2359	589	646	768	877
	EricPlugin	3.5	3.9	9.2	13.8	375	380	531	639	70	75	110	150
	Moin	6.6	8.7	12.5	16.4	470	510	692	1037	243	268	348	420
	networkx	3.6	3.7	8.8	13.3	39	40	82	121	32	33	54	77
	Plone	2.6	4.3	11.1	19.8	474	508	1868	2458	153	181	342	541
	scons	2.5	2.9	4.9	8.2	316	322	401	545	109	115	150	205
	Zope	9.4	11.1	15.3	20.3	7876	8028	10931	15324	1479	1581	1977	2470
% Total Python		7.4	9.3	13.9	19.0	**#CP per Clone Class**				4.0	3.8	4.3	4.7

Table 2: Proportion of files that have clones over a certain percentage

Lang	System Name	UPIT	v=0+% Both	v = 10+% Meth	v = 10+% LOC	v = 20+% Meth	v = 20+% LOC	v = 30+% Meth	v = 30+% LOC	v = 40+% Meth	v = 40+% LOC	v = 50+% Meth	v = 50+% LOC	v=100% Both
C	**Avg. w/o Linux**	0.0	15.3	14.1	9.9	12.5	6.0	10.5	4.8	9.2	4.5	8.8	4.4	0.3
		0.3	42.2	38.4	34.4	32.8	28.0	28.5	25.0	25.2	20.9	22.9	16.6	12.5
	Linux	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
		0.3	49.6	38.1	31.0	26.0	21.5	18.0	15.7	13.3	12.1	10.7	9.5	3.1
Java	**Avg.**	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
		0.3	59.4	54.3	44.9	44.1	33.9	34.2	26.1	26.9	20.1	22.2	16.7	5.1
C#	**Avg.**	0.0	28.6	25.1	20.1	21.6	16.7	18.2	14.0	14.3	12.3	13.5	10.1	5.7
		0.3	65.4	62.8	55.3	56.1	46.3	48.4	40.2	41.8	34.6	38.5	28.5	14.7
Python	EricPlugin	0	31	18.8	15	14.1	8.5	8.9	3.3	6.6	0.5	3.3	0	0
		0.3	48.8	37.1	33.3	30	25.8	22.5	19.2	17.4	16	16	13.1	3.3
	moin	0	24.1	19	11.9	14.6	8.6	12.2	6.6	9.3	5.3	8.2	4.4	2.6
		0.3	37.2	31.4	24.8	24.8	18.2	20.1	14.4	15.9	12.8	14.6	11.1	4.2
	networkx	0	25.0	21.0	12.0	14.0	12.0	11.0	10.0	10.0	9.0	10.0	8.0	8.0
		0.3	41.0	39.0	35.0	30.0	28.0	24.0	21.0	18.0	16.0	14.0	12.0	8.0
	Plone	0	29.7	20	10.4	12.6	6.2	7.7	4.2	4.8	3.5	4.2	3.3	2.2
		0.3	57.3	50.7	43.6	38.3	27.1	24.7	20.5	15.2	14.5	11.2	9.5	3.1
	scons	0	25.7	15.8	11.6	12.2	9.9	10.9	9.2	10.2	7.9	9.2	7.6	5.9
		0.3	38.3	31	25.4	25.1	19.5	19.5	16.2	16.5	13.5	16.5	12.9	9.9
	Zope	0	43.8	33.7	20.6	24.5	14.6	18.3	11.9	14.6	10.5	13	9.6	5.8
		0.3	60.9	52.3	40.9	40.6	30.7	30.6	24.2	25	20.5	21.9	18.5	9.3
	Avg.	0.0	32.1	24.0	16.5	18.1	12.3	13.8	9.6	11.4	7.8	9.9	6.7	4.6
		0.3	49.0	42.3	35.9	33.8	27.3	26.2	21.8	20.6	18.0	18.3	15.1	7.2

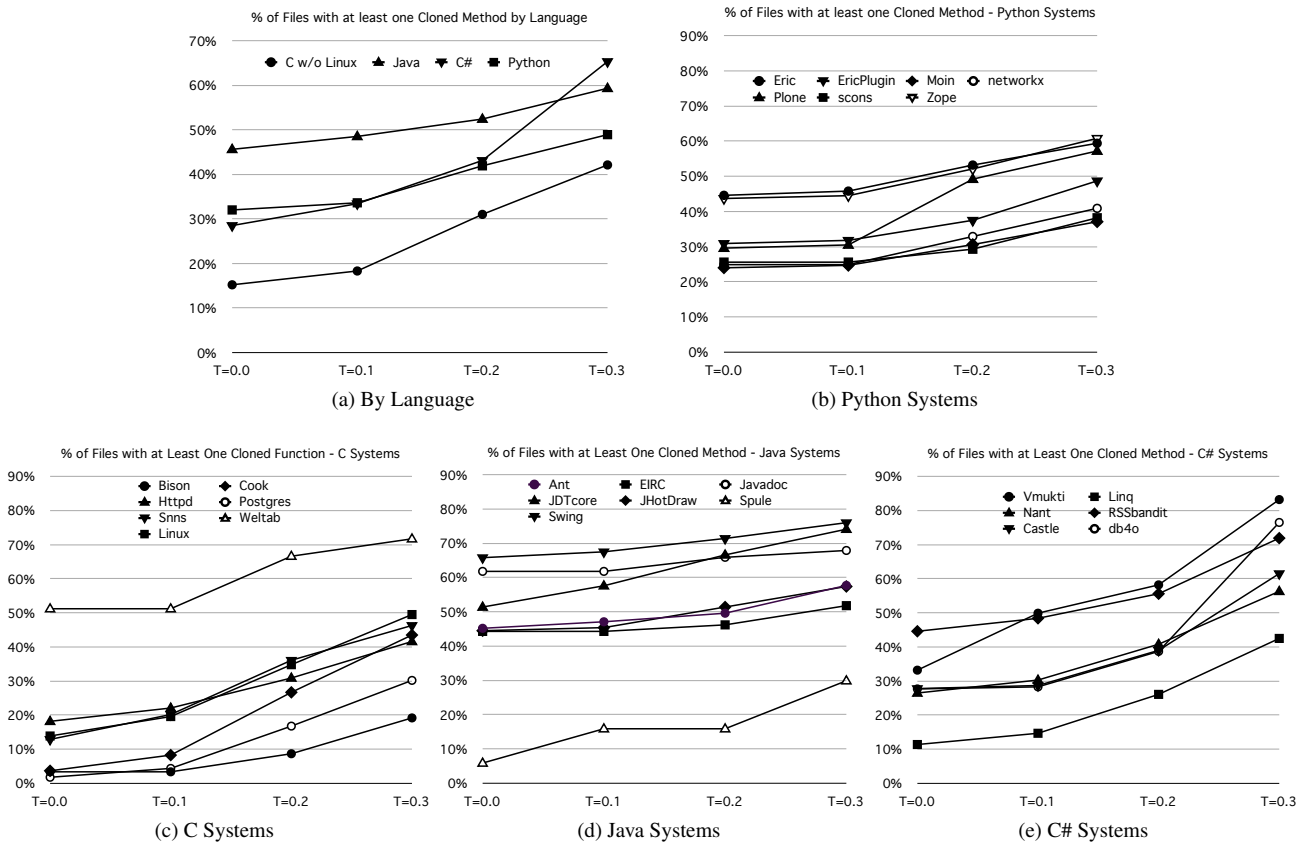
metrics for each of the languages of the systems. When *v=0+%* or *v=100%* (the fourth and last columns) both metrics are the same.

From the average rows of the table with *v=10+%* we can see that on average 14.1% (9.9% 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). C# and Python systems, on the other hand, are in the middle, at 25.1% (20.1% LOC) for C# and at 24.0% (16.5%) for Python with respect to the number of cloned methods/functions. When we increase the UPI threshold to 0.3, the C# systems become the winners, at 62.8% (55.3% LOC), with Java systems second at 54.3% (44.9% LOC) compared to 38.1% (31.0% LOC) for the C systems, and 42.3% (35.9% LOC) for the Python systems. For higher values of v, say 50+% with UPI threshold 0.3, C# systems are still the winners, at 38.5% (28.5% LOC). Both C and Java systems tend to have about the same percentage, at 22.9% (16.6% LOC) for C systems and 22.2% (16.7% LOC) for Java systems, while Python systems a bit lower, at 18.5% (16.1% LOC).

When we look at the individual Python systems, these also exhibit clone density percentages very similar to similarly sized systems in other languages. Thus for clone density Python systems again seem not particularly different for the systems we studied.

3.4 Profiles of Cloning Localization

In this section we examine the localization of Python clones compared to systems in other languages using the $PCLMp$ and $PRCLMp$ related metrics. The location of a clone pair is a factor in software maintenance [9, 24]. A code fragment can form a clone pair with another fragment within the same file, in a different file located in the same directory, or in a different file in a different



(a) By Language (b) Python Systems

(c) C Systems (d) Java Systems (e) C# Systems

Figure 2: Proportion of source files associated with cloned code

directory. Our overall comparison can be summarIzed by Figure 3, which compares the localization of clones in Python systems to systems in other languages at the UPIT=0.3 difference threshold.

In Table 3 we show the percentage clone pairs for each of the different categories for the Python, C, Java and C# 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., 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). Due to limited space, we include only the summary averages for C, Java and C# systems - details for these systems can be found in our previous study [16]

From the table 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, 19.3% (18.6% LOC) for C# systems, and 12.4% (8.9% LOC) for Python systems.

However, when we detect near-miss clones by allowing a higher UPI threshold, we see that the metrics values grow at a higher rate for the C systems than the Python, Java and C# systems. For example, when UPI threshold is 0.3, on average 49.8% (52.5% 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 and 24.7% (33.7% LOC) in the Python systems. C# systems show 33.0% (36.2%), a little bit higher percentages than 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 Python, Java and C# systems. We also see that there is a higher correlation between the Java and Python systems than C and C# systems.

Both Python and Java systems tend to have a higher percentage of exact clones (UPI threshold 0.0) in the same directory but different files than the C and C# systems, with C systems higher than C# systems. However, 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). Among the C# systems, *Castle*, the biggest C# system in this study, shows the highest, 56.5% (58.4% LOC), whereas the largest Python system, *Zope*, has only 15.8% (15.9% LOC). C# systems are the winners for exact clone pairs in different directories though, at 62.9% (63.2% LOC), compared to 45.5% (50.5% LOC) for C systems, 36.2% (39.9% LOC) for Python systems, and 20.1% (19.5% LOC) for Java systems.

When we increase the UPI threshold to look for near-miss clones, we see an interesting difference between language paradigms. While the percentages of different directory clone pairs decrease significantly for procedural C systems, from 45.6% (52.1% LOC) to 15.0% (14.2% LOC), they tend to remain constant or even increase for the object-oriented Java and C# systems. Python systems fall in the middle, remaining constant up to UPI threshold 0.2 but decreasing with UPI threshold 0.3. Figure 3a provides an overview of the localization of near-miss clone pairs at UPI threshold 0.3.

To provide further insight into clone locality, we also computed the $PRCLMp$ metrics for locality of remote clone pairs (i.e., those in different directories). Two fragments of a different directory clone pair might still share the same grandparent directory, or the same subsystem, or they may be from different subsystems. Table 4 shows the data for the Python, C, Java and C# systems that have different directory clone pairs. Due to limited space we include only the summary averages for C, Java and C# systems - details for these systems can be found in our previous study [16].

From the *Average* rows of the table, we see that in general exact clone pairs (UPI threshold 0.0) that are not in the same file or

Table 3: Percentage localization of clone pairs

Lang	System Name		Same File and Same Dir				Same Dir but Different Files				Different Dirs			
		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
C	Avg. w/o Linux	Meth	0.0	24.6	44.7	49.8	37.7	46.3	35.7	35.2	45.6	29.1	19.6	15.0
		LOC	0.0	35.2	48.3	52.5	31.2	41.5	34.0	33.4	52.1	23.3	17.7	14.2
	Linux	Meth	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
Java	Average	Meth	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
C#	Average	Meth	19.3	20.8	28.3	33.0	24.0	23.3	19.5	14.5	56.7	55.9	52.3	52.4
		LOC	18.6	22.8	31.8	36.2	24.4	22.6	18.5	15.3	57.0	54.7	49.9	48.5
Python	EricPlugin	Meth	20.5	20.5	22.6	26.6	67.5	67.1	53.5	47.7	12	12.4	23.9	25.7
		LOC	9.2	9.6	26.5	30.9	74.3	72.8	39.5	35.7	16.5	17.6	33.9	33.4
	Moin	Meth	14.5	13.7	23.1	28.5	53	51.2	45.5	41.7	32.6	35.1	31.4	29.8
		LOC	6.8	6.1	15.9	24.1	63.3	55.2	51.4	49.5	29.9	38.6	32.7	26.4
	networkx	Meth	0	0	28	29.8	76.9	77.5	58.5	60.3	23.1	22.5	13.4	9.9
		LOC	0	0	48.6	46.4	56.9	58.6	35.9	42.9	43.1	41.4	15.6	10.8
	Plone	Meth	38	40.4	20.6	30	33.8	32.9	40.3	37.5	28.3	26.8	39.1	32.5
		LOC	34.9	47.9	36.8	50.8	38.8	32.6	34.3	29.1	26.3	19.5	29	20.1
	scons	Meth	9.2	9.9	19.5	29.7	59.2	59	54.6	45.7	31.6	31.1	25.9	24.6
		LOC	8	12.9	37.2	48.6	57	55.5	42.1	34.3	35.1	31.7	20.8	17.1
	Zope	Meth	3.1	3.3	4.3	12.9	15.7	15.8	13.4	12.1	81.2	80.9	82.2	75
		LOC	2.8	4.7	9.3	19.7	15.9	15.9	13.2	12.1	81.3	79.4	77.5	68.2
	Avg.	Meth	12.4	12.7	17.4	24.7	51.4	51.0	45.8	41.4	36.2	36.3	36.8	33.9
		LOC	8.9	11.7	25.5	33.7	51.1	48.4	37.7	34.7	39.9	39.9	36.9	31.6

Table 4: Percentage localization of remote clone pairs

Lang	System Name		Same Grandparent				Same Subsystem				Different Subsystem			
		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
C	Avg. w/o Linux	Method	44.2	49.8	53.4	53.8	30.7	24.4	20.3	17.9	25.1	25.7	26.2	28.3
		LOC	39.6	52.3	50.9	57.7	35.3	21.2	22.4	14.6	25.0	26.5	26.7	27.6
	Linux	Meth	31.1	31.1	36.9	35.7	64.2	64.4	59.2	59.4	4.7	4.5	3.9	4.9
		LOC	30.7	33.2	41.0	40.9	65.7	62.8	55.0	54.5	3.5	4.0	4.0	4.6
Java	Average	Meth	38.1	38.1	37.1	31.6	41.7	41.9	40.8	42.6	20.2	20.0	22.1	25.8
		LOC	35.5	34.9	34.6	29.7	42.4	44.3	42.2	43.6	22.1	20.8	23.2	26.7
C#	Average	Meth	36.6	36.5	40.0	38.1	48.7	48.7	48.3	38.8	14.7	14.8	11.7	23.1
		LOC	38.5	38.4	40.8	37.3	47.2	47.2	48.0	39.2	14.3	14.4	11.3	23.5
Python	EricPlugin	Meth	53.3	53.2	66.9	55.5	15.6	14.9	5.5	5.5	31.1	31.9	27.6	39.0
		LOC	56.1	56.8	64.3	50.4	4.0	3.6	0.8	0.8	39.9	39.6	34.9	48.8
	Moin	Meth	22.2	19.6	18.0	20.4	76.5	79.3	81.1	79.0	1.3	1.1	0.9	0.6
		LOC	21.3	13.8	12.0	13.3	78.1	85.9	87.8	86.5	0.5	0.3	0.2	0.2
	networkx	Meth	55.6	55.6	45.5	41.7	44.4	44.4	36.4	33.3	0.0	0.0	18.2	25.0
		LOC	90.2	90.2	60.9	56.9	9.8	9.8	6.6	6.2	0.0	0.0	32.5	37.0
	Plone	Meth	14.2	14.7	3.6	4.6	85.8	85.3	96.4	95.4	0.0	0.0	0.0	0.0
		LOC	16.6	19.6	6.2	7.9	83.4	80.4	93.8	92.1	0.0	0.0	0.0	0.0
	scons	Meth	16.0	16.0	16.3	17.2	73.0	73.0	71.2	64.2	11.0	11.0	12.5	18.7
		LOC	23.9	23.9	23.9	27.2	66.1	66.1	63.1	58.0	10.1	10.1	13.1	14.8
	Zope	Meth	11.6	11.5	9.3	13	84.1	84	87.4	84.3	4.4	4.4	3.3	2.7
		LOC	9.3	9.4	8.2	11.6	72.9	73.5	80.2	79.6	17.8	17.2	11.6	8.8
	Avg.	Meth	34.7	34.3	32.9	32.2	58.5	58.8	58.2	55.5	6.8	6.9	8.9	12.3
		LOC	41.1	40.4	35.6	34.8	49.1	50.0	51.2	49.6	9.8	9.6	13.2	15.7

directory tend to have the same grandparents or subsystems. For example, 36.6% (38.5% LOC) of different directory exact clone pairs of C# systems have the same grandparents, while only 14.7% (14.3% LOC) are under different subsystems. Python, C and Java tend to have similar percentages to C#.

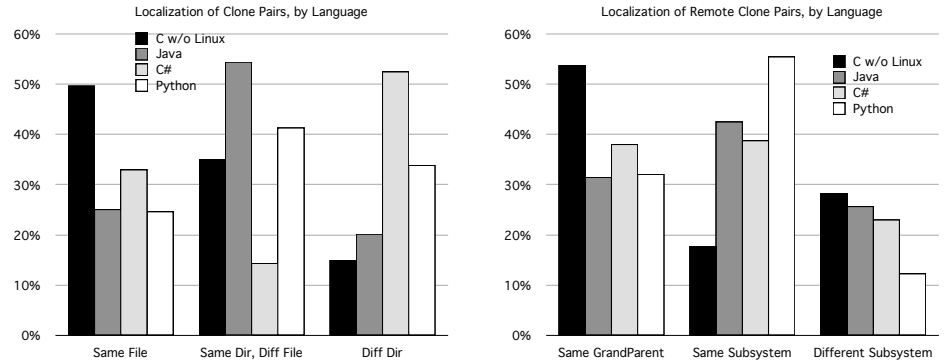
When we increase the UPI threshold to detect near-miss clones, we see that for Python, C and C# systems, the percent of same grandparent clone pairs tends to remain constant or even increase, while Java systems decrease (from 37% to 31%). Percentages of clone pairs for the other two metrics (same subsystem and different subsystems) also tend to remain constant for Python, C and C# systems, but increase for Java systems. Figure 3b provides an overview of the localization of remote near-miss clone pairs at UPI threshold 0.3. Once again these metrics show that in terms of cloning, Python systems are not very different from traditional systems.

3.5 Summary

In this experiment we have explored the question of whether the cloning properties of scripting languages, as represented by Python, differ from those observed for systems written in the more traditional languages C, Java and C#. In light of the higher level operations available in Python, we might have expected fewer clones, and conversely, in light of the rapid development style of Python programming, perhaps more. For the systems we have studied, Python turns out to be right in the middle, and similar in every cloning respect to other languages, and particularly to Java. In overall cloning level, clone associated files, cloning density and both near and far clone localization, we observed that the properties of Python systems are very similar to those implemented in other languages.

3.6 Threats to Validity

The main threat to the conclusions of this study is the lack of a sound definition of code clones. While one can precisely define exact clones, there is no widely accepted definition for near-miss clones. In this study we have used a dissimilarity threshold on standard pretty-printed code as a measure of near-miss clones, intended to model the copy/paste/edit cycle used by programmers. Another threat is the question of whether our open source systems are typical of their languages. We have handled this issue by choosing a



(a) % localization of clone pairs, by language

(b) % localization of remote clone pairs, by language

Figure 3: Localization of near-miss clone pairs by language, at UPIT = 0.3

range of kinds of applications in each language. Finally, we cannot be sure that Python is typical of scripting languages in general. This we can only address with further experiments on other languages.

4. 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, but these studies focus on validating the methods [21] rather than on the subject systems.

Several tool comparison studies have used open source systems for comparing different tools [21]. The Bellon et al. experiment [5, 4] used four C and four Java systems to compare several state-of-the-art tools. Kapser and Godfrey conducted extensive empirical studies [9, 8] with Apache *httpd*, the Linux file system and several other open source systems, providing a detailed categorization of code clones in the form of a taxonomy. Empirical studies of cloning in the Linux Kernel have also been carried out [2], focussing on clone evolution. Al-Ekram et al. [1] have also conducted an promising empirical study on cloning, focussing on C/C++ systems from two different domains. Uchida et al. [23] studied code clones in 125 open source C packages for software analysis.

The most closely related work to ours is the work of Rajapakse and Jarzabek [13], which was also one of the motivations of our study. While they studied exact cloning in web applications, we look at the scripting language itself, and at near-miss clones, in a variety of applications. This work is directly related to our previous studies [16, 18] on the cloning properties of open source C, Java and C# systems. In this study, we extend our work to Python systems and compare the results to the more traditional languages.

5. CONCLUSION

In this paper we have provided an empirical study of function clones in a variety of open source applications written in the Python scripting language, and compared it to several C, Java and C# open source software systems of varying size. Our results indicate that while there is good reason to believe that scripting languages such as Python might be different with respect to cloning, the evidence is that they are quite similar to traditional languages in this respect, exhibiting a large number of copy/paste/edit clones. In fact, at least in the case of Python, the cloning characteristics seem to be very similar to Java, on the face of it a much lower level language.

Acknowledgements

This work is supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) and by IBM Canada through a CAS Faculty Award.

6. REFERENCES

[1] 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.

[2] 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.

[3] L. Aversano, L. Cerulo, and Massimiliano Di Penta. How Clones are Maintained: An Empirical Study. In *CSMR*, pp. 81-90, 2007.

[4] S. Bellon. Detection of Software Clones: Tool Comparison Experiment: *http://www.bauhaus-stuttgart.de/clones/* (December 2007).

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

[6] P. Bulychev and M. Minea. An Evaluation of Duplicate Code Detection using Anti-unification. In *IWSC*, pp. 22-27, 2009.

[7] E. Juergens, F. Deissenboeck, B. Hummel and S. Wagner. Do Code Clones Matter? In *ICSE*, pp. 485-495, 2009.

[8] C. Kapser and M.W. Godfrey. "Cloning Considered Harmful" Considered Harmful: Patterns of Cloning in Software. *Empirical Software Engineering*, 13(6):645-692 (2008).

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

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

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

[12] Open Source Software in Python. *http://pythonsource.com/* (Jan 2010)

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

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

[15] C.K. Roy. Detection and Analysis of Near-miss Software Clones. Ph.D. Thesis, Queen's University, Kingston, Canada, 263 pp., 2009.

[16] C.K. Roy and J.R. Cordy. Near-miss Function Clones in Open Source Software: An Empirical Study. *Journal of Software Maintenance and Evolution*, 25 pp. (online October 2009, in press)

[17] C.K. Roy and J.R. Cordy. IWSC'10 Clone Results: *http://www.cs.queensu.ca/home/stl/download/NICADOutput/*.

[18] C.K. Roy and J.R. Cordy. An Empirical Study of Function Clones in Open Source Software Systems. In *WCRE*, pp. 81-90, 2008.

[19] 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.

[20] C.K. Roy, J.R. Cordy and R. Koschke. Comparison and Evaluation of Clone Detection Techniques and Tools: A Qualitative Approach. *Science of Computer Programming*, 74(7):470-495, 2009.

[21] 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.

[22] C.K. Roy and J.R. Cordy. A Mutation / Injection-based Automatic Framework for Evaluating Clone Detection Tools. In *Mutation'09*, pp. 157-166, 2009.

[23] S. Uchida, A. Monden, N. Ohsugi and T. Kamiya. Software Analysis by Code Clones in Open Source Software. *Journal of Computer Information Systems*, XLV(3):1-11, 2005.

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