Distinguishing Copies from Originals in Software Clones

Jens Krinke, Nicolas Gold, Yue Jia
King's College London
Centre for Research on Evolution, Search and Testing (CREST)
{jens.krinke,nicolas.gold,yue.jia}@kcl.ac.uk

David Binkley
Loyola University Maryland
Baltimore, MD, USA
binkley@cs.loyola.edu

ABSTRACT

Cloning is widespread in today's systems where automated assistance is required to locate cloned code. Although the evolution of clones has been studied for many years, no attempt has been made so far to automatically distinguish the original source code leading to cloned copies. This paper presents an approach to classify the clones of a clone pair based on the version information available in version control systems. This automatic classification attempts to distinguish the original from the copy. It allows for the fact that the clones may be modified and thus consist of lines coming from different versions. An evaluation, based on two case studies, shows that when comments are ignored and a small tolerance is accepted, for the majority of clone pairs the proposed approach can automatically distinguish between the original and the copy.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*Software configuration management*; D.2.13 [**Software Engineering**]: Reusable Software—*Reusable libraries*

General Terms

Algorithms

Keywords

Clone detection, mining software archives, software evolution

1. INTRODUCTION

The duplication of code is a common practice to make software development faster, to enable "experimental" development without impacting the original code, or to enable independent evolution [7]. Since these practices involve both duplication and modification, they are collectively called *code cloning* and the duplicated code is called a *code clone*. A *clone group* consists of code clones that are clones of each other (sometimes this is also called a *clone class*). During the software development life cycle, code cloning is an easy and inexpensive (in both effort and money) way to reuse existing code. However, such practices can complicate software maintenance so it has been suggested that too much cloned code is a risk, albeit the practice itself is not generally harmful [16]. Because of these problems, many approaches to detecting cloned code have been developed [2, 3, 8, 15, 18–20, 24, 26]. While methods to identify clones automatically and efficiently are to some extent understood, it is still disputable whether the presence of clones is a risk. To better understand why and how code is cloned, recent empirical studies of cloned code have focused mainly on examining the evolution of clones, such as whether cloned code is more stable or changed consistently [1, 10, 12, 17, 21, 22, 27].

A lot of research has been done on finding and identifying software clones, but without additional information it is impossible to distinguish the original from the copy. Most of the above mentioned previous empirical studies used version control systems to extract limited information about the discovered clones; for example, when a clone appears in some previous version. However, so far there has been no general approach proposed to distinguish originals from copies except for a study done by German et al. [11] who tracked when clones appeared in the version history to identify the clone of a pair that appeared first. This paper presents an approach that uses line-by-line version information available from version control systems to distinguish the original from the copied code clone in a clone pair.

Most version control systems have a 'blame' command which shows author and version information for each line in a file. This information, which includes the version when the line was added or last modified, can be used as a line age: if all lines in one clone have older versions than the lines in the other clone of a clone pair, then the clone with the older lines may be the original and the other may be the copy (assuming that the clone with the oldest lines existed first). However, usually, it is not that simple because the original and the copy may have been modified in turn after the copy was created.

This paper makes the following contributions:

- A language-independent approach to identify the clones in one version of a program and distinguish the original from its copy in every clone pair by mapping the version information, retrieved from a version control system, to each line of the clones.
- Two initial case studies evaluating the approach show that when comments are ignored and a small tolerance is accepted, the majority of clone pairs can be automatically separated into the original and the copied clone.

The following section presents background on clones and clone detection and the retrieval of version information. Section 3 then presents the approach to distinguishing copied clones from original clones. This is then evaluated in Section 4. Related work is discussed in Section 5 and the last section discusses future work and concludes.

2. BACKGROUND

This section presents the framework in which code clones, groups of code clones, and changes to code clones are defined. This is followed by a description of how version information is retrieved from version control systems and how it is mapped onto the source code lines.

2.1 Code Clones

Code clones are usually described as source code ranges (or fragments) that are identical or very similar. They are grouped into *clone groups* (sometime called *clone classes*) which are sets of identical or very similar code clones. A code clone $c = (s, l, f)$ is the source code range starting at line s with the following l lines of code in file f, thus the last line of the code clone is $s + l - 1$. A clone group $G = \{c_1, \ldots, c_n\}$ is a set of n code clones $c_1, \ldots, c_n$, where each of the code clones is a clone of the others. A group consisting of two clones is a *clone pair*. The clone pairs of a group are generated by pairing all clones of a group.

For the purpose of this study, the effects of *split* or *fragmented* code clones are ignored. Such clones would consist of multiple source code ranges in the same file. An example of such a code clone is a source code range that is copied and additional source code subsequently inserted into the copied code.

The code clones do not have to be disjoint: it is possible for two code clones $c_1 = (s_1, l_1, f)$ and $c_2 = (s_2, l_2, f)$ to share a common source range ($min(s_1 + l_1, s_2 + l_2) > max(s_1, s_2)$).

2.2 Version Information

Most current version control systems can track changes to a file line-by-line to show for each line the version when the line was last changed. *CVS* has an “annotate” command and *subversion* names the command “blame” because it shows the version and the author (‘to be blamed’). These commands give crude information about the origins of the code based on when it was last changed and who made that change.

Usually, the blame command retrieves the version information for the current version or for a specific version for one file or a list of files. In the following, the existence of a function $\mathrm{V}(f, n)$, which retrieves the version (age) of source code line n from source file f of the current version of the program is assumed. This function can be used to retrieve the version of each source line in a clone $c = (s, l, f)$ present in the current version of the program.

3. CLASSIFICATION OF CLONES

Based on the framework to describe clones and the version of a source code line, this section presents a simple approach that uses the version information to classify clone pairs and to distinguish copies from originals.

The version information can be used to classify a clone pair c_1, c_2 with $c_1 = (s_1, l_1, f_1)$ and $c_2 = (s_2, l_2, f_2)$ into specific patterns. First, it is assumed that both clones have the same length ($l_1 = l_2$). How to classify pairs with different lengths will be addressed later. The patterns are

Identical. A clone pair is *identical* if all corresponding lines in c_1 and c_2 have the same version:

$$\forall_{i=0\ldots l_1-1} \mathrm{V}(f_1, s_1 + i) = \mathrm{V}(f_2, s_2 + i)$$

Copied. A clone pair is *copied* if the versions of all lines in c_1 are either larger or smaller than the corresponding lines' versions in c_2:

$$\forall_{i=0\ldots l_1-1} \mathrm{V}(f_1, s_1 + i) < \mathrm{V}(f_2, s_2 + i)$$

or

$$\forall_{i=0\ldots l_1-1} \mathrm{V}(f_1, s_1 + i) > \mathrm{V}(f_2, s_2 + i)$$

If the first condition holds, c_1 is most likely the original and c_2 is most likely the copy. If the second condition holds, it is the other way round.

Unclassifiable. A clone pair is unclassifiable if it is neither identical nor copied.

This classification is coarse-grained and can be extended to take into account if all lines in a clone have the same version or if they come from different versions. In the later case, the clone has been modified at least once after its creation. Thus, the classification becomes

Identical with a single version (IS). To this class belong all identical clone pairs where all lines are from the same version:

$$\forall_{i=1\ldots l_1-1} \mathrm{V}(f_1, s_1) = \mathrm{V}(f_1, s_1 + i)$$

Identical with multiple version (IM). All identical clone pairs that are not identical with a single version belong to this class.

Copied, single version to single version (CS2S). To this class belong all copied clone pairs where all lines in a clone are from the same version:

$$\begin{aligned}\forall_{i=1\ldots l_1-1}\ & \mathrm{V}(f_1, s_1) = \mathrm{V}(f_1, s_1 + i) \wedge \\ & \mathrm{V}(f_2, s_2) = \mathrm{V}(f_2, s_2 + i)\end{aligned}$$

This class contains all copied clones where the original and the copy are not modified after their creation.

Copied, multiple version to single version (CM2S). To this class belong all copied clone pairs where all lines in the copy are from the same version (and that are not copied, single version to single version). Assume that c_1 is the original and c_2 the copy (if not, swap the pair).

$$\forall_{i=1\ldots l_1-1} \mathrm{V}(f_2, s_2) = \mathrm{V}(f_2, s_2 + i)$$

This class contains all copied clones where the copy is not modified after its creation (but the original is).

Copied, single version to multiple version (CS2M). To this class belong all copied clone pairs where all lines in the original are from the same version (and that are not copied, single version to single version). Assume that c_1 is the original and c_2 the copy (if not, swap the pair).

$$\forall_{i=1\ldots l_1-1} \mathrm{V}(f_1, s_1) = \mathrm{V}(f_1, s_1 + i)$$

This class contains all copied clones where the original is not modified after its creation (but the copy is).

Copied, multiple version to multiple version (CM2M). This class contains all copied pairs that do not belong to one of the previous three classes.

The above classification can only be applied to clones of equal length and does not allow for small differences of a few lines. In the following an extension is introduced that allows some tolerance. To do this, a limited number of source lines in the clones of a pair may be removed.

The clones of a clone pair c_1, c_2 with $c_1 = (s_1, l_1, f_1)$ and $c_2 = (s_2, l_2, f_2)$ are said to be *classifiable with a tolerance of* t if after removing t source lines the resulting pair can be classified according to the above classification. The removal of the same line in c_1 and c_2 will count as one removal.

This can be efficiently implemented by computing a modified Levenshtein distance: for each clone $c = (s, l, f)$, the list of versions $R = v_1, ..., v_l$ with $v_i = V(f, s + i - 1)$ is generated. Then three distances are computed for the version lists R_1 and R_2 of clone pair c_1, c_2:

1. $d_=$ is the normal Levenshtein distance between R_1 and R_2.
2. $d_\leq$ is the Levenshtein distance between R_1 and R_2 where two versions are considered to be equal if the version of R_1 is less than or equal to the version in R_2.
3. $d_\geq$ is the Levenshtein distance between R_1 and R_2 where two versions are considered to be equal if the version of R_1 is greater than or equal to the version in R_2.

The distances can now be used to do the classification with a tolerance t:

Identical. A clone pair is *identical with tolerance* t if the distance $d_=$ between the version lists of its clones is below the tolerance ($d_= \leq t$).

Copied. A clone pair is *copied with tolerance* t if the distance $d_\leq$ or the distance $d_\geq$ between the version lists of its clones is below the tolerance ($d_\leq \leq t \vee d_\geq \leq t$).

Unclassifiable. A clone pair is unclassifiable with tolerance t if it is neither identical nor copied with tolerance t.

The above classification is the same as the initial one when the tolerance is set to zero.

4. CASE STUDY

The classification presented above will be evaluated in this section to validate if it allows an automatic distinction between the original and the copy in a clone pair. The approach was implemented in a tool that uses Simian[1] to identify the clones in a system and then applies the classification based on the version information available from the system's subversion repository.

4.1 ArgoUML

In a first case study, the tool was applied to ArgoUML[2], which is a UML modeling tool that includes support for standard UML diagrams. This system is often used in case studies for clone detection (for example in the studies by Kim et al. [17], Aversano et al. [1], Krinke [21], and Thummalapenta et al. [27]). It is written in Java and its version archive is available via subversion. It has been checked out from its subversion repository at version 17737 (January 2010). Simian (version 2.2.24) was used to identify the clones in all Java files in ArgoUML. Simian has been used with the default settings except that the minimal size of a clone was set to 10 lines. In this initial study, lines that only consist of whitespace are ignored. Table 1 shows a summary of the results.

[1]`http://www.redhillconsulting.com.au/products/simian`
[2]`http://argouml.tigris.org`

Table 1: Classification Results

141	Identical	
61	"	with a single version
80	"	with multiple versions
251	Copied	
30	"	single version to single version
107	"	multiple version to single version
0	"	single version to multiple version
114	"	multiple version to multiple version
169	Unclassifiable	

Table 2: Results ignoring Whitespace and Comments

154	Identical	
61	"	with a single version
93	"	with multiple versions
306	Copied	
37	"	single version to single version
130	"	multiple version to single version
2	"	single version to multiple version
137	"	multiple version to multiple version
101	Unclassifiable	

Figure 1 shows an example of a clone pair that has been classified as copied. The clones of the clone pair consist of 11 identical source code lines. The version numbers of the first clone are all larger than or equal to the version numbers of the corresponding lines in the second clone and thus, the approach classified it as 'copied, multiple version to multiple version'. The second clone (the original) has most of the lines dating back to version 8533 (one even back to 8186). The corresponding lines in the first clone are dating back to version 15147, suggesting that the original has been cloned at that version. Moreover, the lines 1, 5, 7, and 11 are dating back to version 15154 in both clones. This suggests that there was a consistent change to both clones in that version.

A further manual inspection revealed that many clone pairs could not be classified because they differed only in comment lines. Therefore, in addition to whitespace, lines containing (only) comments were also ignored. Table 2 shows the results with a zero tolerance: The number of clone pairs that can be classified as copied has increased significantly. It can be seen that in the majority of cases (306 out of 561), these clone pairs can be automatically separated into original and copy. In 267 cases, the original was modified before it was copied (the pairs have been classified as 'copied, multiple version to single/multiple version(s)'. 154 clone pairs had the same version for all corresponding lines (classified as identical). 61 of them had the same version for all lines; thus they came in to existence at the same time and were not subsequently modified. The other 93 pairs were modified after their creation but in a coherent way so that both clones contain the same versions.

Without additional information, the clone pairs classified as identical cannot be distinguished if one is the copy of the other or the pair was created as a pair of clones at the same time intentionally.

The approach was not able to classify 101 pairs, because the versions in both clones did not match. In most cases, only a few lines do not match. Such lines have been changed by different authors in different versions. A manual inspection revealed that in some cases, the copy or the original had additional lines only containing closing brackets ("}") that were not present in the other clone. Thus, the original and the copy had different block structures. Whether this is a code smell is left for further investigation.

ModeContract.java:92,102

```
 1: 15154 int startOffset = layer.getNodeIndex(startY);
 2: 15147 int endOffset;
 3: 15147 if (startY > endY) {
 4: 15147     endOffset = startOffset;
 5: 15154     startOffset = layer.getNodeIndex(endY);
 6: 15147 } else {
 7: 15154     endOffset = layer.getNodeIndex(endY);
 8: 15147 }
 9: 15147 int diff = endOffset - startOffset;
10: 15147 if (diff > 0) {
11: 15154     layer.contractDiagram(startOffset, diff);
```

ModeChangeHeight.java:95,105

```
 1: 15154 int startOffset = layer.getNodeIndex(startY);
 2:  8186 int endOffset;
 3:  8533 if (startY > endY) {
 4:  8533     endOffset = startOffset;
 5: 15154     startOffset = layer.getNodeIndex(endY);
 6:  8533 } else {
 7: 15154     endOffset = layer.getNodeIndex(endY);
 8:  8533 }
 9:  8533 int diff = endOffset - startOffset;
10:  8533 if (diff > 0) {
11: 15154     layer.contractDiagram(startOffset, diff);
```

Figure 1: An Example of a Clone Pair classified as Copied

Table 3: Results with Tolerance (ignoring Whitespace and Comments)

$t:1$	$t:2$	$t:3$	*Classification*
173	187	198	Identical
61	61	61	" with a single version
112	126	137	" with multiple versions
320	323	326	Copied
37	37	37	" single version to single version
130	132	132	" multiple version to single version
1	2	2	" single version to multiple version
152	152	155	" multiple version to multiple version
68	51	37	Unclassifiable

For larger clones, it is often the case that the clones consist of multiple methods that are changed in different ways. If each method is considered as a separate clone, the clone pairs would often be classifiable.

Table 3 shows the results with tolerances $t = 1...3$. For tolerance $t = 1$, it can be seen that 33 additional clone pairs have now been classified (reducing the clone pairs that have not been classified by a third). However, 19 additional clone pairs have now been classified as identical and only 14 additional clone pairs could be classified as copied. Note that by allowing a tolerance, a clone pair that was classified as copied before may now have been classified as identical.

By further increasing the tolerance to $t = 2$, 17 additional clone pairs can be classified (as shown in the second column of Table 3). However, only three additional classifications as copied appear while the number of identical classifications increased by 14. An increase to $t = 3$ adds three more classifications as copied but 11 classification as identical. This suggests that an even further increased tolerance will not significantly change the ability to automatically classify clone pairs as copied. Figure 2 shows the trend of the three main categories for increasing tolerance. The chart makes is clear that an increased tolerance mainly affects the number of clone pairs classified as identical and to a lesser extent the number of clone pairs classified as copied.

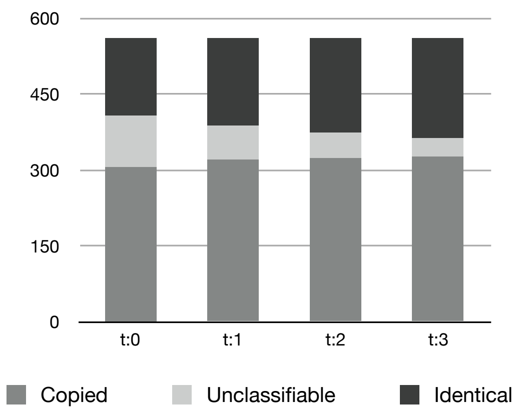

Figure 2: Classifications with increasing Tolerance

An example where a clone pair is almost identical is shown in Figure 3 where ignored lines have no line number. Only lines 12 and 13 have different versions in both clones. Here it seems that the author has created the two clones at the same time (both clones differ with respect to their comments) and modified them incoherently in the next version. By looking closely at the information available in the subversion repository, it becomes clear that the file “EvaluateExpression.java” has been created in version 15296 and the author has only fixed the programming style in version 15297.

EvaluateExpression.java:775,793

```
 1: 15297 if (node.getPathName() != null) {
 -: 15297     // TODO support other name kinds
 2: 15296     node.getPathName().apply(this);
 -: 15297
 3: 15296     feature = node.getPathName().toString().trim();
 4: 15296 }
 5: 15297 if (node.getTimeExpression() != null) {
 -: 15296     // XXX hypothesis: no time expression (inv)
 6: 15296     node.getTimeExpression().apply(this);
 7: 15296 }
 8: 15297 if (node.getQualifiers() != null) {
 -: 15296     // TODO understand qualifiers
 9: 15296     node.getQualifiers().apply(this);
10: 15296 }
11: 15297 if (node.getFeatureCallParameters() != null) {
12: 15297     val = null;
13: 15296     node.getFeatureCallParameters().apply(this);
 -: 15297
14: 15924     parameters = (List) val;
```

EvaluateExpression.java:440,456

```
 1: 15297 if (node.getPathName() != null) {
 -: 15297     // TODO support other name kinds
 2: 15296     node.getPathName().apply(this);
 3: 15296     feature = node.getPathName().toString().trim();
 4: 15296 }
 5: 15297 if (node.getTimeExpression() != null) {
 -: 15296     // hypotheses no time expression (only invariants)
 6: 15296     node.getTimeExpression().apply(this);
 7: 15296 }
 8: 15297 if (node.getQualifiers() != null) {
 -: 15296     // XXX: hypotheses no qualifiers (I don't know)
 9: 15296     node.getQualifiers().apply(this);
10: 15296 }
11: 15297 if (node.getFeatureCallParameters() != null) {
12: 15296     val = null;
13: 15297     node.getFeatureCallParameters().apply(this);
14: 15924     parameters = (List) val;
```

Figure 3: Example of a Clone Pair that could not be classified automatically

Table 4: Classification Results for Apache

$t:0$	$t:1$	$t:2$	*Classification*
37	51	54	Identical
21	21	21	" with a single version
16	30	33	" with multiple versions
96	113	119	Copied
16	16	16	" single version to single version
25	25	25	" multiple version to single version
0	0	1	" single version to multiple version
55	72	77	" multiple version to multiple version
80	49	40	Unclassifiable

Table 5: Results for Apache (ignoring Whitespace and Comments)

$t:0$	$t:1$	$t:2$	*Classification*
38	54	60	Identical
21	21	21	" with a single version
17	33	39	" with multiple versions
103	118	124	Copied
15	15	23	" single version to single version
27	27	27	" multiple version to single version
0	0	1	" single version to multiple version
61	76	73	" multiple version to multiple version
72	41	29	Unclassifiable

However, while this clone pair cannot be classified with zero tolerance, it is wrongly classified as copied with a tolerance of one. Only with a tolerance of two it is correctly classified as identical. This shows that an increased tolerance can increase the number of false positives.

This example also shows that there are some differences in the comments present in the clones. If comments were not ignored, the pair would not be classifiable even with higher tolerance.

For the majority of clone pairs in the above case study, the original and the copy can be distinguished. The number of clone pairs classified as copied increases when comments are ignored and a small tolerance is accepted.

4.2 Apache

In a second case study, the http server Apache[3] was analyzed. It is an open source system written in C. From its subversion archive version 900584 (January 2010) was retrieved and used for clone detection with Simian executed on all C files (excluding header files). Simian was again used with the default settings except that the minimal size of a clone had been set to 10 lines.

Table 4 shows the results for different tolerances when only whitespace lines are ignored. From 213 clone pairs, only 96 have been classified as copied with zero tolerance. 37 have been classified as identical and 80 were unclassifiable. The numbers improve with an increased tolerance. With $t = 2$, the unclassified pairs drop to 40, the pairs classified as copied increase to 119, however, the identical pairs increase to 54.

Table 5 shows the results for different tolerances when whitespace and comment lines are ignored. With zero tolerance, the number of pairs classified as copied is up to 103, while 38 pairs are classified as identical and 72 pairs are not classified. This further improves with a tolerance of $t = 2$: Only 29 pairs are now unclassifiable and 124 pairs have been classified as copied, while the pairs classified as identical increases to 60.

[3]`http://httpd.apache.org`

The case study for Apache confirms the findings of the first case study: when comments are ignored and a small tolerance is accepted, for the majority of clone pairs the original and the copied clones can be distinguished.

4.3 Threats to Validity

There are some potential threats to validity in the presented study. First of all, there is no clear definition of a clone. Moreover, a clone detected by a clone detector may not be a clone in reality (false positive) or a clone in a system may be missed by a clone detector (false negative). It is known that clone detectors have a low recall [4], so the false negatives cause a threat to validity which cannot be estimated. Another potential threat to validity is caused by the use of a version control system to retrieve version information. Because subversion (and CVS) detect changes with *diff*, simple changes in the amount of whitespace generate new versions. Thus, changing the layout of a file will likely set many lines to the current version. Also, because *diff* does not identify the movement of text or code, refactorings or restructurings causes deletions and additions.

The experiment is also influenced by the analyzed systems. So far only two systems have been analyzed. However, they are both of different application types, are written in different programming languages, are of sufficient size, and have a long history.

5. RELATED WORK

The presented approach was also used to study how code is copied and cloned between subprojects of the GNOME Desktop Suite [23]. The case study revealed a complex flow of reused code between the different subprojects. In particular, it showed that the majority of larger clones (with a minimal size of 28 lines or higher) exist between the subprojects and more than 60% of the clone pairs can be automatically separated into original and copied clone.

German et al. [11] used version information to identify the version where a clone has been introduced. For a clone pair between two systems it was then possible to identify the system where the cloned code appeared first. In contrast, they did not use the fine grained information available from blaming, but used clone detection between the versions to track a clone as a clone pair between the current and previous versions which is an expensive operation as it requires many invocations of the clone detector (as opposed to only a single invocation).

A number of authors have studied the evolution of clones. Kim et al. [17] investigated the evolution of code clones and defined several evolution patterns to classify all possible changes during the clone evolution. The results suggested that during the evolution of the code clones, there were fewer consistent changes to the clones than anticipated. Aversano et al. [1] did a similar empirical study with a slightly refined framework. Similar to Kim et al., they analyzed so called co-changes that are changes committed by the same author, with the same notes, and within 200 seconds (into a CVS repository). They used a Java-only clone detector that compares subtrees in the abstract syntax tree. The analyzed systems were DNSJava and ArgoUML. Aversano et al. state that the majority of clone classes are always maintained consistently.

A similar framework and experiment was presented by Krinke [21] to study the evolution of code clones with respect to consistent and inconsistent changes. The changes were reconstructed from data stored in a version control system (subversion or CVS). He found that the number of consistent and inconsistent changes were similar. In a second study, Krinke [22] investigated whether cloned or non-cloned code is more stable with respect to the number of changes applied to cloned and non-cloned code. Again, he reconstructed the changes from version control systems.

With a similar setup where changes are extracted from version control systems, Göde [12] presented a model for clone evolution where he tracked the evolution of individual clones throughout the history of a program. Thummalapenta [27] uses an automatic approach to classify the evolution of source code clone fragments and investigates to what extent clones are consistently changed or evolve independently. Clone fragments are also tracked individually and the evolution of clones are classified into patterns.

Previous studies have shown (and suffered from) the fact that systems often undergo cosmetic changes. This blurs the identification of whether a clone has been changed. Prause [25] uses techniques similar to clone detection to identify probable ancestors of source code and to locate origins of inserted code. Canfora [5, 6] introduces a technique to track the evolution of source code lines, identifying whether a CVS change is due to line modifications rather than due to additions and deletions. The technique compares the sets of lines added and deleted in a change set, combining the use of Information Retrieval (IR) techniques with the Levenshtein edit distance. Other approaches use the syntactic structure of the software to identify the changes. Fluri et al. [9] introduce *change distilling*, a tree differencing algorithm for fine-grained source code change extraction. Their algorithm extracts changes by finding a match between the nodes of the two compared abstract syntax trees. Weißgerber and Diehl [28] use a similar approach to automatically identify refactorings. In addition, they use clone detection techniques to rank the refactoring candidates. Godfrey and Zou [13] present techniques to apply origin analysis to detect instances of merging and splitting in source code together with a set of merge/split patterns.

6. CONCLUSIONS AND FUTURE WORK

This paper presents an approach that uses version information from version control systems to automatically classify the clones of a clone pair into the original and the copy. The classification allows for the fact that the clones may be modified and thus consist of lines coming from different versions. The classification may allow small tolerances (i.e., a small number of lines of the clones may be ignored for the purposes of classification).

The evaluation, based on two case studies, showed that when comments are ignored and a small tolerance is accepted, the origin or copied nature of the majority of clone pairs can be automatically distinguished. A larger evaluation with more case studies will follow in the future to determine whether the approach presented here is able to distinguish the original from copied clones on a larger scale. It is also planned to use better origin tracking approaches like those of Prause [25] or Canfora [5, 6] to enable a more precise identification of a source code line's version and a better distinction between original and copied clones.

The current approach is limited to gap-less clones but can handle near miss clones [26] by allowing some tolerance. It is planned to extend it so that it is able to handle gapped clone and to evaluate it with KClone [14].

In summary, this paper has presented a method for automatically classifying a clone pair into the original and its copy, using line-age information derived from a version control system. The method is shown to be successful when applied to two open-source case studies.

7. ACKNOWLEDGEMENTS

This work is funded in part by Hewlett Packard and the FOSSology Project.

8. REFERENCES

[1] L. Aversano, L. Cerulo, and M. D. Penta. How clones are maintained: An empirical study. In *11th European Conference on Software Maintenance and Reengineering (CSMR)*, 2007.

[2] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Second Working Conference on Reverse Engineering*, pages 86–95, 1995.

[3] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *International Conference on Software Maintenance (ICSM)*, pages 368–378, 1998.

[4] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. Softw. Eng.*, 33(9):577–591, Sept. 2007.

[5] G. Canfora, L. Cerulo, and M. Di Penta. Identifying changed source code lines from version repositories. In *In Fourth International Workshop on Mining Software Repositories, MSR*, volume 19, 2007.

[6] G. Canfora, L. Cerulo, and M. Di Penta. Tracking your changes: A language-independent approach. *IEEE Software*, 26(1):50–57, January 2009.

[7] J. Cordy. Comprehending reality – practical barriers to industrial adoption of software maintenance automation. In *11th IEEE International Workshop on Program Comprehension*, pages 196–205, 2003.

[8] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *International Conference on Software Maintenance (ICSM)*, pages 109–118, 1999.

[9] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. Change distilling:tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, November 2007.

[10] R. Geiger, B. Fluri, H. C. Gall, and M. Pinzger. Relation of code clones and change couplings. In *9th International Conference of Funtamental Approaches to Software Engineering (FASE)*, number 3922 in LNCS, pages 411–425. Springer, Mar. 2006.

[11] D. M. German, M. Di Penta, Y.-G. Gueheneuc, and G. Antoniol. Code siblings: Technical and legal implications of copying code between applications. In *6th IEEE International Working Conference on Mining Software Repositories*, pages 81–90. IEEE Computer Society, May 2009.

[12] N. Göde. Evolution of type-1 clones. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 77–86. IEEE Computer Society, 2009.

[13] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, February 2005.

[14] Y. Jia, D. Binkley, M. Harman, J. Krinke, and M. Matsushita. KClone: a proposed approach to fast precise code clone detection. In *Third International Workshop on Detection of Software Clones (IWSC)*, 2009.

[15] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, July 2002.

[16] C. Kapser and M. W. Godfrey. "Cloning considered harmful"

considered harmful. In *13th Working Conference on Reverse Engineering (WCRE)*, pages 19–28, 2006.

[17] M. Kim, V. Sazawal, and D. Notkin. An empirical study of code clone genealogies. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE)*, pages 187–196, 2005.

[18] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Eigth International Static Analysis Symposium (SAS)*, volume 2126 of *LNCS*, 2001.

[19] K. Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *Fourth Working Conference on Reverse Engineering*, pages 44–54, 1997.

[20] J. Krinke. Identifying similar code with program dependence graphs. In *Proc. Eigth Working Conference on Reverse Engineering*, pages 301–309, 2001.

[21] J. Krinke. A study of consistent and inconsistent changes to code clones. In *14th Working Conference on Reverse Engineering (WCRE)*, Oct. 2007.

[22] J. Krinke. Is cloned code more stable than non-cloned code? In *Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 57–66. IEEE Computer Society, September 2008.

[23] J. Krinke, N. Gold, Y. Jia, and D. Binkley. Cloning and copying between gnome projects. In *7th IEEE Working Conference on Mining Software Repositories*, may 2010.

[24] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *International Conference on Software Maintenance (ICSM)*, pages 244–254, 1996.

[25] C. R. Prause. Maintaining fine-grained code metadata regardless of moving, copying and merging. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 109–118. IEEE Computer Society, 2009.

[26] C. K. Roy and J. R. Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *The 16th IEEE International Conference on Program Comprehension*, pages 172–181. IEEE Computer Society, 2008.

[27] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta. An empirical study on the maintenance of source code clones. *Empirical Software Engineering*, March 2009.

[28] P. Weißgerber and S. Diehl. Identifying refactorings from source-code changes. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 231–240. IEEE Computer Society, September 2006.