An Assessment of Type-3 Clones as Detected by State-of-the-Art Tools

Rebecca Tiarks, Rainer Koschke, Raimar Falke
University of Bremen
Bremen, Germany
{*beccs,koschke,rfalke*}*@informatik.uni-bremen.de*

***Abstract*—Code reuse through copying and pasting leads to so-called software clones. These clones can be roughly categorized into identical fragments (type-1 clones), fragments with parameter substitution (type-2 clones), and similar fragments that differ through modified, deleted, or added statements (type-3 clones). Although there has been extensive research on detecting clones, detection of type-3 clones is still an open research issue due to the inherent vagueness in their definition.**

In this paper, we analyze type-3 clones detected by state-of-the-art tools and investigate type-3 clones in terms of their syntactic differences. Then, we derive their underlying semantic abstractions from their syntactic differences. Finally, we investigate whether there are any additional code characteristics that indicate that a tool-suggested clone candidate is a real type-3 clone from a human's perspective. Our findings can help developers of clone detectors to improve their tools.

***Keywords*-software clones; clone categorization; type-3 clones;**

I. Introduction

Over the past years there has been considerable research in the field of clone detection, that is, the search for identical or similar pieces of code. While finding code fragments identical or equal up to parameter substitution is quite reliable with today's automatic clone detectors [1], finding similar fragments with some modifications is still an open problem. This problem is due to the inherent vagueness of similarity: how many and what kind of changes can be tolerated in the decision whether two code fragments are still software clones?

So far there has been no empirical study that analyzes clones with modifications more closely. Such studies would be valuable because we would be able to improve existing detection techniques if we knew the nature of such clones better.

Contributions. In this paper, we present a study of such clones. We address the following research questions:

Q1 What kind of clones with modifications (type-3) do state-of-the-art clone detectors find, that is, what are their code characteristics?

Q2 How can we classify detected type-3 clones in terms of syntactic differences? And how often do they occur?

Q3 What common abstractions can be assigned to the clones?

Q4 Are there any code characteristics that indicate that a tool-suggested clone is a real type-3 clone from a human's perspective?

Overview. The remainder of this paper is organized as follows. Section II introduces required terminology and Section III describes related work. In Section IV, we present our study of analyzing clones with modifications and the answers to the above research questions. Section V, finally, concludes.

II. Terminology

Ira Baxter's frequently cited definition of clones stated at the *First International Workshop on Detection of Software Clones* in 2002 in Montreal stresses the inherent vagueness of software clones:

> "Clones are segments of code that are similar according to some definition of similarity."

The two similar code segments form a *clone pair*, short: *clone*. Their similarity can be based on program text or on semantics. Current research focuses on similar program text because detecting semantic similarity is undecidable in general. Program text similarity may be defined in terms of text, tokens, syntax, or data and control dependencies.

Program-text clones can be further distinguished by their degree of similarity based on the transformations required to transform the two clones into an identical representation. We can roughly distinguish three types of transformations: substitution, addition, and deletion. The following taxonomy summarizes the types of clones [2] (most research literature has distinguished only three types of clones; here we present a more refined classification and map it to the one commonly used):

- **Exact clone (type 1)** is an exact copy of consecutive code fragments without modifications (except for whitespace and comments when insignificant from the perspective of the language definition). That is, the transformation is the identity.
- **Parameter-substituted clone (type 2)** is a copy where only parameters (identifiers or literals) have been substituted. The transformation required is a suitable parameter substitution.
- **Structure-substituted clone (type 3)** is a copy where program structures (complete subtrees in the syntax tree) have been substituted. Given a suitable structure substitution, the transformed copy is a type-1 clone [3].

For parameter-substituted clones, we can replace one leaf in the syntax tree by another leaf. For structure-substituted clones, larger subtrees can be substituted.
- **Modified clone (type 3)** is a copy where pieces within the clone have been added and/or deleted.

Note that these four type classes form an inclusion hierarchy. Every exact clone is a parameter-substituted clone; every parameter-substituted clone is a structure-substituted clone. Substitution in turn is a special case of arbitrary modifications, namely, additions and deletions.

According to the above taxonomy, any two completely different fragments could be considered type-3 clones: one just removes all statements from the first fragment and adds all statements from the second fragment. Obviously, we need to limit the degree of freedom in selecting transformations. The limits of transformations need to be searched in the act of copying itself. Humans copy code because they want to reuse it. Hence, the clone shares some valuable property with its original. If a programmer copies executable statements, she wants to reuse some execution. If she copies declarative statements, she wants to reuse some representation. She can reuse execution and representation because the code she copies fulfills a purpose similar to the one she needs. That is, clones are similar in their intention and implementation.

Ira Baxter's updated definition stated at the recent *International Workshop on Detection of Software Clones* in 2009 in Kaiserslautern goes along these lines:

> "Clones are segments of code that have a common abstraction."

To better understand the nature of similarity among clones and their common abstractions, we look at type-3 clones in this paper.

III. Related Work

Here we present related research. Our focus is on clone taxonomies and tools that detect type-3 clones. For a more general discussion of related research, we refer the reader to our comprehensive surveys on the field [4]–[7].

Clone Taxonomies: The above categorization of ours is a refinement of our earlier categorization [1]. The many types of similarities of programs that may be used to distinguish types of clones were summarized by Walenstein et al. [8].

An early approach to classify clones with the aim to evaluate software systems was presented by Mayrand et al. [9]. Based upon the similarity of clones they define an ordinal scale containing eight categories. The cloning level is determined by various function metrics that compare the name, layout, expression and control flow of the clones.

Balazinska et al. introduced a classification for function clones [10] by distinguishing the type of differences between structure-substituted clones. Differences that do not affect the behavior of a method and its output are referred to as superficial changes. The next category consists of changes where the signature of a method is affected, for instance, return values or modifiers (`public`, `static`, etc.). The most complex differences are those affecting types of parameters or local variables or types modified by typecasts. All differences that do not fit in one of the previous mentioned categories are grouped together as other differences. This classification helps to select a suitable strategy for clone removal.

Another classification by Kapser et al.'s distinguishes by scope (within or across a file), type of region at file level (functions, global declarations, macros), degree of overlap or containment, and type of code sequence (e.g., initialization code) [11]–[13]. They state that cloned blocks of cloned code are difficult to characterize [12]. They found that blocks are often copied within a control structure such as `if/else`. In many cases, clone blocks are functions pairs that have more than 60 % of code in common which means that large parts of one function is copied and afterwards extended. They reported that in many cases whole functions were copied and modified by appending statement to the end [12]. They noted that the large number of clone blocks indicates that more clone categories exist that are not yet investigated.

Clearly, the definition of redundancy, similarity, and cloning in software is still an open issue. Walenstein has summarized the debate on terminology among the clone experts at a Dagstuhl seminar [14]. There is little consensus in this matter. One problem is that the viewpoint of what constitutes a clone may depend upon a particular task and other constraints. For instance, in a refactoring exercise, we want to find only syntactic clones that may be replaced by the available means of the programming language. On the other hand, we are interested in every type of copy when we modified a piece of code and need to check whether the same change must be replicated somewhere else.

Clone Detection: Here we describe the tools we used in our study. A comprehensive survey of existing tools was published earlier [7]. All the tools are able to detect type-3 clones.

Lexical clone detectors transform the program into a sequence of tokens and identify clones as equal token subsequences. For efficient detection, suffix trees are typically used [15]. These subsequences are either type-1 or type-2 clones. They may be combined into type-3 clones in a post-processing step if their gap in terms of lines of code is below a certain threshold. Our lexical tool CLONES implements this approach [16]. Rather than just using lexical lines of code to specify a gap, one could use data and control dependencies between these fragments instead [17].

Syntactic clone detectors parse a program and detect clones as similar subtrees in the syntax tree. An efficient way to find equal trees is to serialize the syntax tree into a tree node sequence and then to use suffix trees to

Table I
SUBJECT SYSTEMS

System	Version	Language	KLOC
Ant	date 15.2.02	Java	35
JDTCore	r3.3	Java	148
Swing	J2SDK1.4.0	Java	204
Javadoc	release 331	Java	19
bison	1.32	C	19
postgresql	7.2	C	235
snns	4.2	C	115
wget	1.5.3	C	16

find equal subsequences. Similarly to the lexical detection, close subsequences may be combined to type-3 clones. Our syntactic tool CLAST implements this approach [16].

A computationally more expensive approach is to apply syntax tree matching to subtrees in the syntax tree. To avoid unnecessary comparisons, one can filter subtrees by a hash function for trees [18]. If the comparison is an inexact tree match, one can find type-3 clones. Another option is to combine two subsequent cloned subtrees if their gap in terms of leaf nodes is below a certain threshold. Leaf nodes are atomic expressions such as identifiers and literals. Our syntactic tool CCDIML implements this approach [16].

Metric-based clone detectors gather code metrics and compare these metrics instead of the program text. If the distance between metric vectors characterizing code fragments is within an allowable tolerance, one can identify type-3 clones. CLAN implements this approach [1], [9].

Dependency-based clone detectors represent the program as a system dependency graph (SDG) whose nodes represent statements and expressions and whose edges are the control and data dependencies among these nodes. Type-3 clones are identified as similar subgraphs in the SDG. The tool DUPLIX implements this approach [1], [19].

IV. INVESTIGATION OF TYPE-3 CLONES

This section describes our investigation of type-3 clones. Here, we describe our study and the answers to our research questions.

Subject Systems: To investigate the research questions from Section I, we looked at clones in various systems. The selection of systems was aimed at covering a wide range of type-3 clones. For this reason, we choose eight different open-source systems varying in size from 19 KLOC to 235 KLOC written in C or Java. Table I lists these systems and their characteristics. These systems belong to the Bellon Benchmark used in a comprehensive quantitative evaluation of several clone detectors [1].

Tools: As described in Section III, we used five tools for the clone detection process. In order to find various kinds of type-3 clones, we chose tools that are based on different detection methods. Using different tools and detection methods decreases the degree of bias. The results from DUPLIX and CLAN were taken from the original database of the Bellon Benchmark experiment [1]. As the other tools involved in the former experiment did not report type-3 clones or did not classify their clones at all, we included only these two tools. Further we examined only those clones that were found by the tools in the mandatory part of the experiment. In the mandatory part, the tools had to be configured with their default settings. From our own tools we evaluated CCDIML, CLONES and CLAST. The details on the underlying detection techniques are described in Section III. To ensure comparability, we analyzed the source code with our tools' default settings.

The whole set of type-3 clones (referred to as candidates in the following) contained 381,628 clone pairs. All tools reported candidates for all systems except DUPLIX, which is able to analyze only C systems.

Clone Oracling: One author of this paper, namely, Tiarks, looked at 751 random samples of the set of candidates. This number was determined by the amount of time available for the study. Overall, it took one person week to validate the candidates. The tool used for the evaluation was an enhanced version of Stefan Bellon's tool *cloneval*, which he developed in context of the aforementioned experiment [1]. This tool selects clone candidates randomly and fairly, that is, the same number of candidates is selected for each tool and system. The two code fragments of each candidate clone were presented in a split text view side by side and could be evaluated in an interactive way. In order to categorize the clone pairs, we adapted the tool to our needs and added a possibility to assign categories to a presented candidate. We oracled only clones with a minimal length of six lines. Candidates containing import statements only were also skipped as they do not represent valid type-3 clones.

Out of the 751 evaluated candidates, 189 (25 %) were accepted as true type-3 clones. 49 (11 %) of the accepted type-3 clones were function clones and 89 % were sequences of declarative or executable statements.

A. Characteristics of Type-3 Clones

This section addresses research question *Q1* about characteristics of type-3 clones found by state-of-the-art tools. The oracled pairs (both accepted and rejected) were analyzed through several metrics.

Three different similarity measures were computed. Two of these measures use the string edit distance, also known as Levenshtein distance. Levenshtein counts the number of additions, deletions, and modifications needed to transform one sequences of symbols into another sequence of symbols. This value is normalized by the number of symbols of the longer string to obtain a similarity between zero and one. The two measures differ in their interpretation of a symbol. For the first similarity measure (called "token similarity", short: *tokenSim*) each token represents one symbol, that is, only the token type is used. For the second measure (called "text similarity", short: *textSim*) the token texts

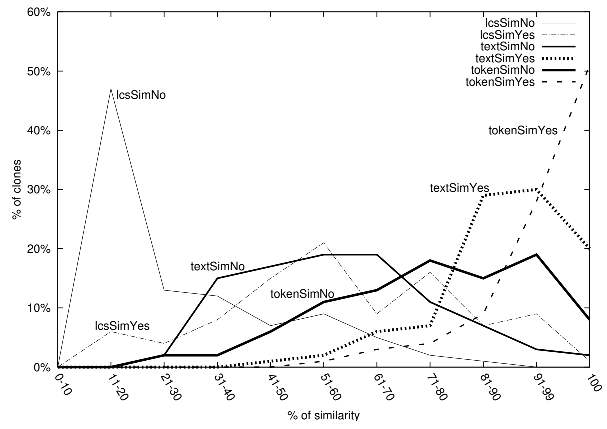


Figure 1. Similarity values for oracled candidates (dashed lines: accepted clones; solid lines: rejected ones).

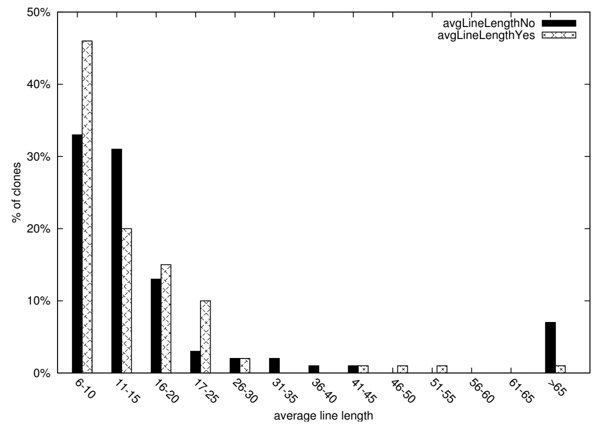


Figure 2. The line length for all oracled clones

of each fragment are concatenated (separated by exactly one whitespace between two token texts) and Levenshtein distance is computed where every character constitutes a symbol. Additionally a line-based similarity measure was implemented by counting the lines that are equal between the two fragments. To do that, the lines of both fragments have to be aligned first, which can be done by solving the *Longest Common Subsequence* problem. The degree of similarity is the relative number of equal lines, $common$, of each clone pair using the following equation (let l_1 and l_2 be the line length of the two fragments f_1 and f_2):

$$\mathit{lcsSim}(f_1, f_2) = \frac{common(l_1, l_2)}{avg(l_1, l_2)} * 100$$

The results from the similarity measure are depicted in Figure 1. The “is-clone” case is represented by the *Yes*-suffix in the legend and the “is not a clone” case has *No* appended to its name. The results clearly show that the percentage of accepted clones increases with a higher percentage of similarity. That means that clone pairs with a very low similarity also have a low probability to be accepted by the user as a valid type-3 clone.

The peak for *LCS*-similarity on the left hand side is due to the fact that the comparison operates on lines rather than tokens or characters. It computes lower similarity values, as two lines have to be exact matches of each other to produce a non-zero value. This metric is the weakest in distinguishing false and true positives of the three similarity metrics. Clones with a text similarity of over 90 % make up about 50 % of all valid clones whereas only 5 % of rejected clones have a similarity above that value. That means that this metric can possibly give hints to tools whether a clone is valid or not. The average text similarity over all accepted clones is 76 % and 47 % for the rejected ones. The average text similarity for all oracled clones is 55 %.

The line length of all oracled clones is depicted in Figure 2, measured as the average of the line lengths of both fragments of each clone pair. The patterned bars represent the accepted clones and the filled bars the rejected ones. The figure starts at line length of six as we oracled only clones with a minimal length of six lines. The average line length of all accepted candidates is 16 lines and rejected clones are 26 lines long on average. The average line length for all oracled clones is 22 lines. The results show that 46 % of accepted clones have a line length between six and ten lines. About 10 % of the rejected clones are longer than 65 lines which means that long clones have a rather low probability of being accepted, which was surprising.

B. Syntactic Difference Classification

This section addresses research question *Q2* about the syntactic differences of type-3 clones. These differences were obtained by manual inspection and judgment using only the accepted candidates. Table II summarizes the results.

At the first level, we distinguish differences in the signature of a function and at the statement level. The latter account for changes in cloned statement sequences as well as in bodies of cloned functions.

The percentages in Table II is the fraction of accepted clones that fall into the respective change category. For instance, we observed 39 changes in comments among the 189 accepted clones, which leads to $39/189 \approx 21\,\%$. Because signature changes can happen only in function clones, we divide the number of clones with signature changes by the number of function clones (49 in total).

Most changes in function signatures are changes of parameter types or result types. In some cases, modifiers or aspects of the parameter list other than just parameter types (i.e., addition of parameters, parameter renaming, and ordering) are changed.

Among the changes at the statement level, the routines being called, the conditions of *if* statements and the variables used are most frequent. The percentage for changes of local

Table II
SYNTACTIC DIFFERENCES OF TYPE-3 CLONES. THE MIDDLE COLUMN SHOWS THE ABSOLUTE NUMBER OF OCCURENCES AND THE RELATIVE AMOUNT IS DEPICTED IN THE THIRD COLUMN.

function signature		
modifiers (public, private..)	2	4 %
parameters		
parameter types	10	20 %
others	2	4 %
return type	7	14 %
statement level		
routines called	68	35 %
if conditional		
condition	64	34 %
then branch	57	30 %
else branch	8	4 %
loop	6	3 %
global variables	60	32 %
local variables	61	31 %
statement added	45	24 %
statement modified	8	4 %
comments	39	21 %
layout and order	17	9 %

variables contain changes of both their names and types.

Changes in the *then* branch are more frequent than changes in the *else* part, but this is relativized by the fact that the *else* branch of an `if` statement is optional. More surprising is that loops are hardly changed at all, neither their condition nor their body. Part of this phenomenon could be attributed to the fact that the clones under investigation rarely had any loops. Likewise, it is interesting that addition of statements is more frequent than modification.

Changes of comments are more frequent than changes of layout and statement order. Not all of the changes of comments were substantial, however. We found many cases in which just typos were corrected. But there were also cases in which one clone fragment had a comment for some of its statements and the other did not. That is, these comments were later either removed or added, but only in one fragment. Some changes were updates in the comments to reflect renames in the objects they document.

C. Common Abstractions

Section IV-B has described clones in terms of their syntactic differences. This section addresses research question *Q3* as to whether clones can be grouped according to common abstractions. The answer to this question addresses research question *Q1*, too, but from a semantic point of view. To answer *Q3*, we looked at all accepted type-3 clones and formed such a categorization bottom-up.

Table III describes the resulting categorization. The figure in the table is the number of occurrences, where a fragment could be assigned to multiple categories. We use the terms *concept* and *operation* in this categorization. In general, a concept is something conceived in the mind, that is, a unit of thinking. Here we use this term more narrowly to refer to concepts implemented as data structures. That is, *concept* is a unifying term for objects, classes, records and the like. Concepts are implemented through declarative code for types and variables. We use the term *operation* for algorithms implemented by executable code.

Table III
ABSTRACTIONS IN TYPE-3 CLONES.

Category	**Characteristic**
Concepts	declarations
→ Structure-Equivalent	same attributes (2)
Operations	algorithms
→ Generic	for different concepts
⟶ Type-Generic	type substitution (24)
⟶ Structure-Generic	structure substitution (2)
⟶ Parameter-Generic	parameter substitution (73)
⟶ Operation-Generic	function substitution (49)
⟶ Expression-Generic	expression substitution (9)
→ Operation Variants	variants for same concepts
⟶ Extensions	subsumption (30)
⟶ True Variants	different algorithms (20)
→ Patterns	use of same interface
⟶ Container Iteration	iteration scheme (3)
⟶ Protocol-Induced	same protocol (8)
⟶ Input-Induced	similar input syntax (5)
⟶ Output-Induced	similar output syntax (3)

At the first level, we can distinguish between reuse of declarative and executable code. Reuse of declarative code is done for structurally similar concepts, that is, two concepts have similar attributes and consequently a programmer copies attributes between data structures. In many cases, this act of copying will likely also lead to copying executable code to reuse algorithms.

In most cases, programmers seem to reuse algorithms (or today's detectors just find these), that is, control and data flow. Here we can distinguish three different subcategories:

- *generic operations*: the same algorithm is used for different concepts; here expressions and types have been substituted, but control and data flow are the same; as Table II already showed such substitutions are quite frequent both in function signatures and at the statement level (an example is shown in Figure 3)
- *operation variants*: similar algorithms handle the same concepts; here the control and data flow are slightly different, but equivalent types and variables are involved; such operation variants often have changes in the parameter list and added and modified statements (cf. Figure 4)
- *patterns*: the algorithm has different statements at specific locations, but control and data flow follow the same pattern; patterns occur when algorithms use the same interface in a very regular prescribed way; here, the called routines, involved types and variables as well as control and data flow are mostly the same, only some statements have been added or modified (cf. Figure 5)

For the latter category, one could argue that the fragments are not really clones of each other because they are not directly coupled but only indirectly via the interface they use.

```
if (!(binaryInfo instanceof IBinaryField )) return false;
  IBinaryField field = ( IBinaryField )binaryInfo;
  // field name
  if (!this.matchesName(this. name , field.getName ()))
     return false;
  // declaring type
  IBinaryType declaringType = (IBinaryType)enclosingBinaryInfo;
  if (declaringType != null) {
    char[] declaringTypeName = (char[])declaringType.getName().clone();
    CharOperation.replace(declaringTypeName, '/', '.');
    if (!this.matchesType(this.declaringSimpleName,
                          this.declaringQualification,
                          declaringTypeName)) {
     return false; } }
  // field type
  String fieldTypeSignature
     = new String( field.getTypeName ()).replace('/', '.');
```

(a) Clone Fragment

```
 if (!(binaryInfo instanceof IBinaryMethod )) return false;
IBinaryMethod method = ( IBinaryMethod )binaryInfo;
 // selector
 if (!this.matchesName(this. selector ,
method.getSelector ()))
       return false;
 // declaring type
 IBinaryType declaringType = (IBinaryType)enclosingBinaryInfo;
  if (declaringType != null) {
    char[] declaringTypeName = (char[])declaringType.getName().clone();
    CharOperation.replace(declaringTypeName, '/', '.');
    if (!this.matchesType(this.declaringSimpleName,
                          this.declaringQualification,
                          declaringTypeName)) {
         return false; } }
  String methodDescriptor
      = new String( method.getMethodDescriptor ()).replace('/', '.');
```

(b) Clone Fragment

Figure 3. Example Generic Operation (layout was modified to fit on this page)

```
topo_ptr = topo_ptr_array + (no_of_topo_units + 3);
wta_value = 0;
wta_pos = 0;


out_pat_value = 0;
out_pat_pos = 0;
j = 0;
/* Winner Takes All - error computation */
if (NoOfOutputUnits > 1) {
   /* calculate output units only  */
   while ((unit_ptr = *--topo_ptr) != NULL) {
        j++;


        if (wta_value < unit_ptr->Out.output) {
          wta_value = unit_ptr->Out.output;
          wta_pos = j; }
        if (out_pat_value < *(--out_pat)) {
          out_pat_value = *(out_pat);
          out_pat_pos = j; } }
   if (wta_pos != out_pat_pos) {
        return ((float) 1.0); }



} else {
   unit_ptr = *--topo_ptr;
   if ((float) unit_ptr->Out.output > 0.5) {
        if (*(--out_pat) < 0.5)
          return ((float) 1.0);



   } else {
        if (*(--out_pat) > 0.5)
          return ((float) 1.0); } }



return ((float) 0.0); }
```

(a) Clone Fragment

```
topo_ptr = topo_ptr_array + (no_of_topo_units + 3);
wta_value = 0;
wta_pos = 0;
 sum_value = 0;
out_pat_value = 0;
out_pat_pos = 0;
j = 0;
/* weighted Winner Takes All - error computation */
if (NoOfOutputUnits > 1) {
   /* calculate output units only  */
   while ((unit_ptr = *--topo_ptr) != NULL) {
        j++;
        sum_value += unit_ptr-> Out.output;
        if (wta_value < unit_ptr->Out.output) {
          wta_value = unit_ptr->Out.output;
          wta_pos = j; }
        if (out_pat_value < *(--out_pat)) {
          out_pat_value = *(out_pat);
          out_pat_pos = j; } }
   if (wta_pos != out_pat_pos)
        return ((float) 1.0);
   else
        return ((float) fabs(wta_value - (sum_value - wta_value)/ (–j)));
} else {
   unit_ptr = *--topo_ptr;
   if ((float) unit_ptr->Out.output > 0.5) {
        if (*(--out_pat) < 0.5)
          return ((float) 1.0);
        else
          return ((float) fabs(unit_ptr->Out.output - *(out_pat)) /2);
   } else {
        if (*(--out_pat) > 0.5)
          return ((float) 1.0);
        else
          return ((float) fabs(*(out_pat) - unit_ptr->Out.output) /2); } }
return ((float) 0.0); }
```

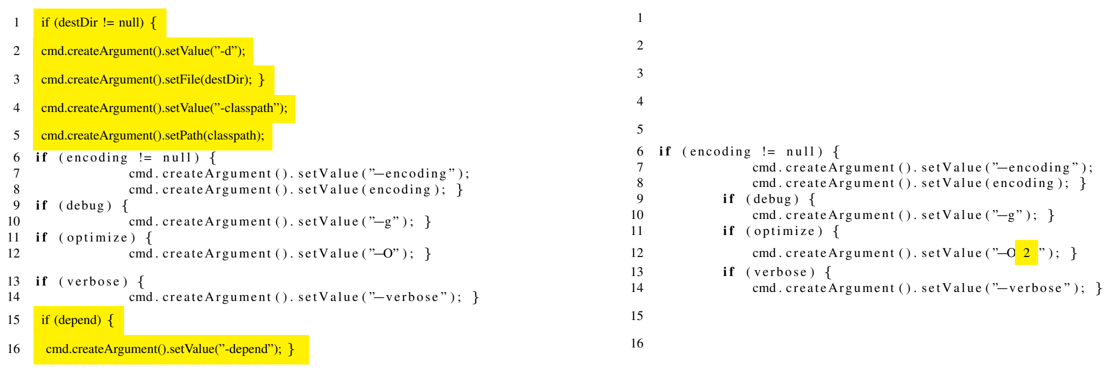
(b) Clone Fragment

Figure 4. Example Operation Variant (layout was modified to fit on this page)

Yet, these patterns are still interesting because they pinpoint to potential interface improvements. These patterns could be offered by the interface directly.

Among the generic operations, we can further distinguish according to the substitutions for their instances. The instances can differ in the types involved, for instance, one uses integer and the other float variables. Structure-generic operations are those generic operations whose handled concepts have the same structural properties, that is, the same attributes (record components or class attributes). Parameter-generic operations differ in their parameters, that is, formal parameters, local or global variables, or constants. Operation-generic operations call different functions or methods. In case of expression-generic operations, more complex expressions are replaced, for instance, `z` by `foo(x) + y`. In many cases, combinations of these different substitutions occur making it difficult to come up with just one simple unifying solution.

In case of operation variants, we can distinguish between extensions, in which case one fragment is completely contained in the other one, and truly different variants who share only part of their control and data flow.


```
if (destDir != null) {
cmd.createArgument().setValue(”-d”);
cmd.createArgument().setFile(destDir); }
cmd.createArgument().setValue(”-classpath”);
cmd.createArgument().setPath(classpath);
if (encoding != null) {
            cmd.createArgument().setValue(”—encoding”);
            cmd.createArgument().setValue(encoding); }
if (debug) {
            cmd.createArgument().setValue(”—g”); }
if (optimize) {
            cmd.createArgument().setValue(”—O”); }

if (verbose) {
            cmd.createArgument().setValue(”—verbose”); }
if (depend) {
 cmd.createArgument().setValue(”-depend”); }
```

(a) Clone Fragment

```
if (encoding != null) {
            cmd.createArgument().setValue(”—encoding”);
            cmd.createArgument().setValue(encoding); }
        if (debug) {
            cmd.createArgument().setValue(”—g”); }
        if (optimize) {

            cmd.createArgument().setValue(”—O 2 ”); }
        if (verbose) {
            cmd.createArgument().setValue(”—verbose”); }
```

(b) Clone Fragment

Figure 5. Example Pattern

Patterns indicate an identical or similar use of interfaces. Container iterations are patterns that iterate over all elements of a container data structure. They differ in the handling of the element obtained in an iteration. Protocol-induced patterns result from a fixed interface protocol that specifies the order in which interface elements are to be used. For instance, GUI code is often very similar because the way widgets are to be created and combined is very similar. Input-induced patterns use an interface for parsing structured input. Because the input has re-occurring similar syntactic structures, their parsing is similar, too. Output-induced patterns are analogous to input-induced patterns where the interface's purpose is to output structured data.

There were also five cases in which the fragments differed only in minor syntactic details, such as fully qualified names, visibility clauses (e.g., `public` or `private`), and order of independent statements. With some normalization, tools should detect them correctly as type-2 clones.

D. Code characteristics

This section addresses research question *Q4* by investigating whether code characteristics – measured through various source code metrics – can be used to distinguish true positives from false positives. The answer to this questions partly addresses research question *Q1*, too.

1) Metrics used: For each oracled clone pair, various metric values are gathered. These include two different text similarity measures, eight different measures for each token type and 25 measures based on the fragment lengths such as number of characters or tokens per line. In addition, the number of occurrences of each fragment in all source code files is counted and the internal repetition (see below) is measured.

Similarity Measures: We used *token similarity* and *text similarity* as described in Section IV-B.

Token Type Metrics: For each token type, the number of occurrences in both fragments is counted, yielding n_1 and n_2. Then the following eight derived metrics are calculated (let l_1 and l_2 be the length of the fragments measured in tokens and $r_1 = n_1/l_1$ and $r_2 = n_2/l_2$): $\frac{n_1+n_2}{2}$, $\min(n_1, n_2)$, $\max(n_1, n_2)$, $|n_1 - n_2|$, $\frac{r_1+r_2}{2}$, $\min(r_1, r_2)$, $\max(r_1, r_2)$ and $|r_1 - r_2|$.

Size and Size Difference Metrics: The following metrics are calculated for each fragment: size in characters, size in tokens, size in lines, tokens per line and characters per token. Let v_1 and v_2 denote the values for the two fragments. Then the following metrics are produced: $\frac{v_1+v_2}{2}$, $\min(v_1, v_2)$, $\max(v_1, v_2)$, $|v_1 - v_2|$ and $2\frac{|v_1-v_2|}{v_1+v_2}$.

Internal Repetition: A common group of false positive clone candidates are clones that consist in large parts of repeating sequences, for instance, array initializers. The repeated token sequence is called *r-element*. Multiple r-elements without a gap between them form an *r-block*. For each fragment, all r-blocks for this fragment are calculated. The length of the largest (in terms of tokens contained) r-block is divided by the length of the fragment (also measured in tokens). These calculations are carried out for both fragments yielding r_1 and r_2. From these the following metrics are derived and used in the following sections: $\frac{r_1+r_2}{2}$, $\min(r_1, r_2)$ and $\max(r_1, r_2)$.

2) Filtering metrics: Due to the variety of token types, over 700 metric values for each clone pair are calculated. In this list, we have to find the subset of metrics that can be used to distinguish positive from negative candidate clones. To search for significant characteristics, we plotted the distributions for the "is-clone" and "is not a clone" in the same diagram for each metric. Ideally the two curves do not overlap and a threshold for the metric under consideration can be selected to categorize the clone pairs. If the curves are very similar or even identical, the metric cannot be used to categorize the input.

To reduce the number of curves to be looked at, some extra metric values for each such frequency diagram were calculated: the area in between the two curves (should be as large as possible), the number of crossings (should be minimal) and the difference of the average of both curves (should be as large as possible). The 80 diagrams with the smallest common area were examined by hand. The other two metric values provided extra insight for the judgment. Table IV contains the calculated metric data for a subset of the 80 diagrams examined.

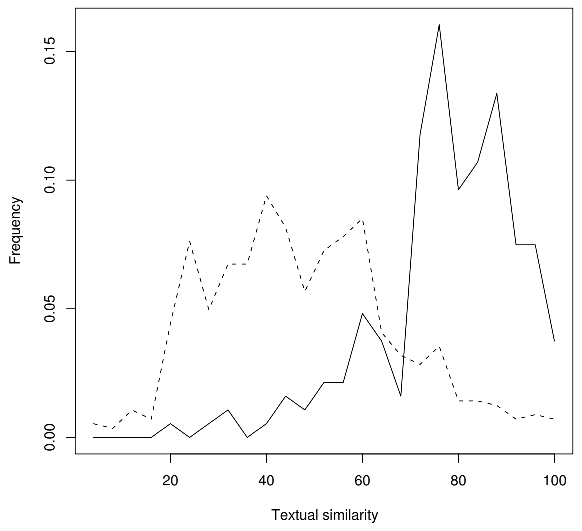


Figure 6. The two frequency distributions of the textual similarity (solid line: accepted clones; dashed line: rejected clones).

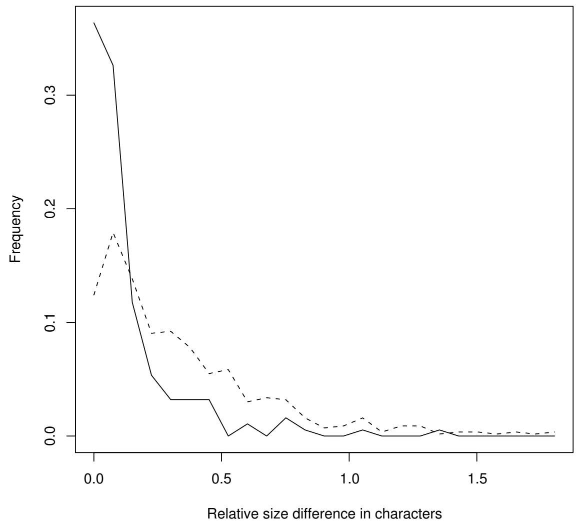


Figure 7. The two frequency distributions of the relative difference $2\frac{|v_1-v_2|}{v_1+v_2}$ of the fragment sizes measured in characters (solid line: accepted clones; dashed line: rejected clones).

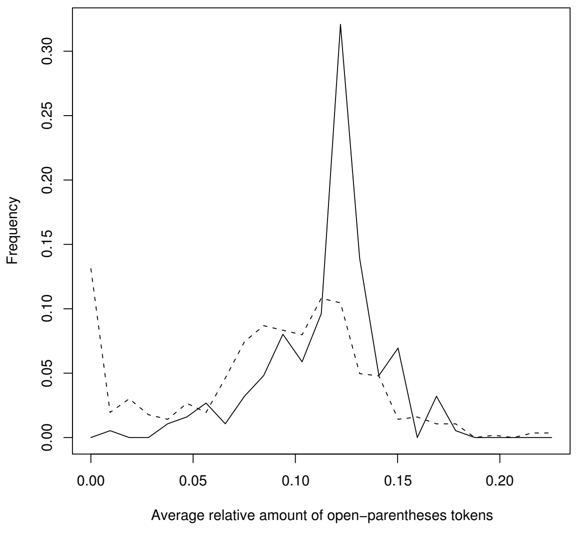


Figure 8. The two frequency distributions of the average rate of open-parentheses “(” tokens (solid line: accepted clones; dashed line: rejected clones).

3) Results: From the these 80 curves, the textual similarity (cf. Figure 6) was identified as the best distinguishing characteristic. The token similarity is less but still suitable for the task. Besides these two, the remaining metrics are much less suited. The relative difference in the fragment sizes $2\frac{|v_1-v_2|}{v_1+v_2}$ (cf. Figure 7) is connected with the similarity measures since a larger difference results in a smaller similarity, but overall the difference measure cannot be used.

In Figure 8, the distribution of the left-parentheses is shown. While there is a clear region where the “is-clone” case is more probable, in general the curves overlap in large areas and so the separability is low.

Still some conclusions can be drawn from these curves: if there is no equal (=) token (which indicates the absence of assignments), the chances that it is really a clone are considerably lower. This is also true if no `if` token, no parenthesis tokens, no braces nor any integer literal is in the fragments. Also when the fragment sizes are similar (measured in characters, tokens or lines), there are higher chances that the candidate is a real clone.

4) Combined metrics: So far the code metrics have been considered independently. A categorization system, however, would not be interested in how many different metrics can be used to decide the problem but how much improvement in the categorization a combination of the different metrics yields. From the last section, it is clear that this improvement has to be measured against the text similarity (the best metric).

Decision trees can be used to find metrics that improve the decision capability of the text similarity. The metrics used and their influence can be read easily from the resulting

Table IV
EXTRA METRIC VALUES FOR A NUMBER OF FREQUENCY DIAGRAMS. GIVEN IS THE AREA BETWEEN THE "IS-CLONE" AND "IS NOT CLONE" CURVES, THE NUMBER OF CROSSINGS OF THESE CURVES, AND THE ABSOLUTE VALUE OF THE DIFFERENCE OF THE AVERAGES OF THE TWO CURVES. ONLY A SUBSET IS SHOWN.

Metric	Area	Crossings	Difference of averages
textSim	1.35	1	30.2
tokenSim	0.99	3	21.2
min. rel. occurrence of a control flow token	0.87	7	10.8
min. rel. occurrence of ")"	0.84	6	15.0
min. rel. occurrence or "{"	0.84	15	10.6
min. rel. occurrence of "("	0.83	6	15.0
avg. rel. occurrence of a control flow token	0.82	9	8.8
...			
rel. difference between number of chars	0.78	3	11.1
min. rel. occurrence or "}"	0.78	8	8.0

tree. This makes them better suited than other machine learning approaches. We explored two different decision tree algorithms: the *rpart* package from the R-project and *C4.5* [20] to be able to draw conclusions that are less dependent upon the particularities of just one algorithm.

The complete list of oracled clone pairs is used for training and then also for the evaluation of the resulting tree. Consequently, our measurement of recall and precision is an overestimation. If we wanted to use the decision trees for a prediction model, we would rather use cross-validation. Yet, here we use decision trees just to identify the distinguishing characteristics, so it is fair to apply them on the whole set for training and evaluation.

In the default configuration, the *rpart* algorithm produces a tree with fifteen nodes with a precision of 0.875 and a recall of 0.711. These nodes use the metrics depicted in Table VIa (the higher its depth, the less significant is a characteristic). To compare both trees, the C4.5 tree was limited in growth to fifteen nodes. The resulting tree has a precision of 0.871 and a recall of 0.652 and uses the metrics in Table VIb.

The decision trees show that a handful metrics are sufficient to categorize the candidate clone pairs to a high degree. The two similarities of the fragments are the best choice as expected. The number of "`(`" indicates `if`, `for` and `while` statements but also function calls. The evaluation of the characteristic's distribution as described in Section IV-D3 suggests that the `if` statements are the most important contributor. This is also supported by the result of the C4.5 trees, since both equality and inequality are heavily used in `if`-statements. Another point is that the number of occurrences and also the amount of repetition can be used to further increase the quality of the categorization.

E. Threats to Validity

If we want to generalize our results, we need to be aware of possible threats of validity of our study. We looked only at those clones certain clone detectors were able to find. There may be other clones not found by these detectors and possibly not even by any other existing detector, and we analyzed only 751 out of their 381,628 clone candidates. Furthermore we looked only at clones of eight open-source systems written in two different programming language. Other systems and languages may have different clones. Open-source system may differ from closed-source systems. Although we have investigated a large number of metrics, we have certainly not covered all possible metrics. Last but not least, we are relying on our personal judgment in analyzing these clones. Other researchers may have different opinions what a type-3 clone is.

V. CONCLUSIONS

Our study shows that there is a need to improve existing detectors for type-3 clones. Only about 25 % of their candidates were accepted as clones by a human oracle. The accepted clones show a high similarity based on Levenshtein distance (LD) in particular when LD is applied to the character representation of code fragments. Furthermore longer fragments are less likely real clones.

Our analysis of differences between accepted clones shows an uneven distribution of types of changes. In most cases, simple parameters (function names, local and global variables and their types) are substituted, that is, type-3 clones are in large parts combinations of type-2 clones. A third of the type-3 clones have statements added or modified, where added statements clearly dominated. That means that many type-3 clones could be classified as type-2 clones with minor syntactic differences. Tools with a little more abstraction and normalization should be able to correctly detect and classify them.

Our analysis of the underlying abstractions of accepted clones yields a semantic classification of clones. Although we conducted our analysis manually, we can envision tools that automatically assign clones to these semantic classes based on their syntactic differences. This semantic classification can help us studying the reasons, effects, and evolution of clones in greater detail and can also help in directing appropriate refactorings.

Our study also shows that there is potential to filter clones based on Levenshtein text similarity and additional criteria

Table V
SIGNIFICANT CHARACTERISTICS ACCORDING TO DECISION TREES

depth	characteristic
0	textual similarity
1	max. relative occurrence of the "(" token
2	max. number of occurrences of each fragment and (in a different subtree) avg. number of tokens in each fragment
3	avg. relative occurrence of the "." token
4	min. relative occurrence of the "−>" token
5	min. percent of internal repetition

(a) *rpart* decision tree

depth	characteristic
0	textual similarity
1	no. of "!=" tokens equal in both fragments?
2	no. of "{" tokens equal in both fragments?
3	relative difference of the "==" token
4	textual similarity
5	between the two fragments: the larger number of characters per token
6	no. of "." tokens equal in both fragments?

(b) *C4.5* decision tree

such as token metrics. Automatic learning algorithms based on decision trees can distinguish real clones from false positives with high recall and precision. Clone detectors could use such data mining techniques to adapt to certain project characteristics and to improve their precision and recall.

REFERENCES

[1] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Computer Society Transactions on Software Engineering*, pp. 577–591, Sep. 2007.

[2] R. Koschke, "Frontiers in software clone management," in *Proc. of the International Conference on Software Maintenance*, 2008.

[3] W. S. Evans, C. W. Fraser, and F. Ma, "Clone detection via structural abstraction," in *WCRE*, 2007, pp. 150–159.

[4] R. Koschke, "Survey of research on software clones," in *Duplication, Redundancy, and Similarity in Software*, ser. Dagstuhl Seminar Proceedings, R. Koschke, E. Merlo, and A. Walenstein, Eds., 2007.

[5] ——, *Identifying and Removing Software Clones*. Springer Verlag, 2008, ch. 2, pp. 15–39.

[6] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," School of Computing, Queen's University at Kingston, Ontario, Canada, Technical Report No. 2007-541, 2007.

[7] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Journal of Science of Computer Programming*, 2009, accepted for publication.

[8] A. Walenstein, M. El-Ramly, J. R. Cordy, W. S., K. Mahdavi, M. Pizka, G. Ramalingam, and J. W. von Gudenberg, "Similarity in programs," in *Duplication, Redundancy, and Similarity in Software*, 2007.

[9] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *ICSM*. IEEE Computer Society, 1996, pp. 244–.

[10] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, "Advanced clone-analysis to support object-oriented system refactoring," in *WCRE*. IEEE Computer Society Press, Oct. 2000, pp. 98–107.

[11] C. J. Kapser and M. W. Godfrey, "Supporting the analysis of clones in software systems: Research articles," *Journal of Software Maintenance and Evolution*, vol. 18, no. 2, pp. 61–82, 2006.

[12] ——, "Toward a taxonomy of clones in source code: A case study," in *Evolution of Large Scale Industrial Software Architectures*, 2003, pp. 67–78.

[13] ——, "A taxonomy of clones in source code: The re-engineers most wanted list," in *In Proc. of IWDSC'03*, 2003.

[14] A. Walenstein, "Code clones: Reconsidering terminology," in *Duplication, Redundancy, and Similarity in Software*, ser. Dagstuhl Seminar Proceedings, No. 06301, 2007.

[15] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Proc. of WCRE*, L. Wills, P. Newcomb, and E. Chikofsky, Eds., July 1995, pp. 86–95.

[16] R. Falke, R. Koschke, and P. Frenzel, "Empirical evaluation of clone detection using syntax suffix trees," *Empirical Software Engineering*, vol. 13, no. 6, pp. 601–643, 2008.

[17] Y. Jia, D. Binkley, M. Harman, J. Krinke, and M. Matsushita, "Kclone: A proposed approach to fast precise code clone detection," in *Proc. of CSMR'09*, 2009, pp. 12–16.

[18] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *ICSM*, T. M. Koshgoftaar and K. Bennett, Eds., 1998, pp. 368–378.

[19] J. Krinke, "Identifying similar code with program dependence graphs," in *WCRE*, 2001, pp. 301–309.

[20] J. R. Quinlan, *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., January 1993.