Code Similarities Beyond Copy & Paste

Elmar Juergens, Florian Deissenboeck and Benjamin Hummel
Institut für Informatik, Technische Universität München, Germany
{juergens,deissenb,hummelb}@in.tum.de

Abstract

Redundant source code hinders software maintenance, since updates have to be performed in multiple places. This holds independent of whether redundancy was created by copy&paste or by independent development of behaviorally similar code. Existing clone detection tools successfully discover syntactically similar redundant code. They thus work well for redundancy that has been created by copy&paste. But: how syntactically similar is behaviorally similar code of independent origin? This paper presents the results of a controlled experiment that demonstrates that behaviorally similar code of independent origin is highly unlikely to be syntactically similar. In fact, it is so syntactically different, that existing clone detection approaches cannot identify more than 1% of such redundancy. This is unfortunate, as manual inspections of open source software indicate that behaviorally similar code of independent origin does exist in practice and does present problems to maintenance.

1. Similarity $\neq$ Similarity

Research in software maintenance has shown that many programs contain a significant amount of duplicated (cloned) code. Such cloned code is considered harmful for two reasons: (1) multiple, possibly unnecessary, duplicates of code increase maintenance costs [16, 24] and, (2) inconsistent changes to cloned code can create faults and, hence, lead to incorrect program behavior [13]. Obviously, the negative impact of clones on software maintenance is not due to copy&paste but caused by the semantic coupling of the clones. Hence, behaviorally similar code, independent of its origin, suffers from the same problems clones are known for. In fact, the re-creation of existing functionality can be seen as even more critical, since it represents a missed reuse opportunity.

The research community has developed a number of successful approaches to detect and manage code duplication. However, the capabilities of existing approaches are not fully clarified yet. While most previous work agrees that the existing approaches are, indeed, limited to detecting copy&pasted (and potentially modified) code, it is sometimes conjectured that they can also find code that is behaviorally similar but has been developed independently. One reason for this uncertainty is that we do not really know how structurally different independently developed code with similar behavior actually is. As a result, it is currently not well understood to which extent real world programs contain redundancy that cannot be attributed to copy&paste although intuition tells us that large projects are expected to contain multiple implementations of the same functionality.

To develop a better understanding of redundancy beyond copy&paste, this paper presents the results of an experiment that investigated how well existing clone detection approaches detect similarity in 109 independently developed variations of the same functionality. Strikingly, existing clone detection approaches did not achieve a *recall* of more than 1% in this experiment although they were run with a very unrestrictive configuration that would yield far too many false positives in practice. Furthermore, we used manual reviews of an open source system to identify if behaviorally similar code that does not result from copy&paste occurs in real world software. This investigation provides a strong indication that this type of redundancy occurs and, indeed, appears to be problematic for software maintenance.

Research Problem While clone detection is a proven approach to detect copy&pasted code, it is unclear in how far clone detection can be used to detect code that is behaviorally similar but not the result of copy&paste. Consequently, we currently do not know the *recall* of clone detection approaches with respect to similar code in general, *i. e.*, not limited to copy&paste.

Contribution We extend the existing empirical knowledge with an experiment that demonstrates that behaviorally similar code of independent origin is unlikely to be representationally similar. With this, we show that existing clone detection approaches are ill-suited to detect code that is behaviorally similar but has been developed independently. We illustrate the relevancy of this shortcoming with a case study in which we used manual reviews to identify behaviorally similar code in an open source system.

2. Notions of Similarity

In this section we differentiate between representational and behavioral similarity of code. To the best of our knowledge, neither clone detection, nor other research areas concerned with program equivalence (including program schemas [8], refactoring [21] and model checking [4]) provide suitable definitions that can serve as a crisp separation criterion between the two. Hence, we retreat to a more informal but also more intuitive description here.

2.1. Program-Representation-based Similarity

Numerous clone detection approaches have been suggested [16, 24]. All of them statically search a suitable program representation for similar parts. Amongst other things, they differ in the program representation they work on and the search algorithms they employ. Consequently, each approach has a *different notion of similarity* between the code fragments it can detect as clones. We classify them by the type of behavior-invariant variation they can compensate when recognizing equivalent code fragments and by the differences they tolerate between similar code fragments.

Text-based approaches detect clones that are equal on the character level. *Token*-based approaches can perform token-based filtering and normalization. They are thus robust against reformatting, documentation changes or renaming of variables, classes or methods. *AST*-based approaches can perform grammar-level normalization and are thus furthermore robust against differences in optional keywords or parentheses. *PDG*-based approaches are somewhat independent of statement order and are thus robust against reordering of commutative statements. In a nutshell, existing approaches exhibit varying degrees of robustness against changes to duplicated code that do not change its behavior.

Some approaches also tolerate differences between code fragments that change behavior. Most approaches employ some normalization that removes or replaces special tokens and can make code that exhibits different behavior look equivalent to the detection algorithm. Moreover, several approaches compute characteristic vectors for code fragments and use a distance threshold between vectors to identify clones. Depending on the approach, characteristic vectors are computed from metrics [15,19] or AST fragments [3,9]. Furthermore, ConQAT [13] detects code fragments that differ up to an absolute or relative edit distance as clones.

In a nutshell, notions of *representational similarity* as employed by state of the art clone detection approaches differ in the types of behavior-invariant changes they can compensate and the amount of further deviation they allow between code fragments. The amount of deviation that can be tolerated in practice is however severely limited by the amount of false positives it produces.

```
int x, y, z;

z = x*y;        | z = 0;
                | while (x > 0) {
                |   z += y;
                |   x -= 1;
                | }
                | while (x < 0) {
                |   z -= y;
                |   x += 1;
                | }
```

Figure 1. Code that is behaviorally equal but not representationally similar.

2.2. Behavioral Similarity

In contrast to a purely syntactical definition of similarity, we can also look at code in terms of I/O behavior. For a piece of code (*i. e.*, a sequence of statements) we call all variables written by this code its *output variables* and all variables which are read and do have an impact on the outputs its *input variables*. Each of the variables has a type which is uniquely determined from the context of the code. We can then interpret this code as a function from valuations of input variables to valuations of output variables, which is trivially state-less (and thus side-effect free), as we captured all global variables in the input and output variables.

We call two pieces of code *behaviorally equal*, iff they have the same sets of input and output variables (modulo renaming) and are equal with respect to their function interpretation. So, for each input valuation they have to produce exactly the same outputs. An example of code that is behaviorally equal but not representationally similar is shown in Figure 1.

For practical purposes often not only strictly equal pieces of code are relevant, but also similar ones. We call such similar code a *simion*. Simions are behaviorally simiIar code fragments where *behavioral similarity* is defined w.r.t. input/output behavior.

As we are interested in their (semi-) automatic detection, the kind of differences tolerated between simions depends both on the task to perform (*e. g.*, removal of redundant code) and the capabilities of the detection algorithm used. One definition would be to allow different outputs for a bounded number of inputs. This would capture code with isolated differences (*e. g.*, errors), for example in boundary cases. Another one could tolerate systematic differences, such as different return values in case of errors, or the infamous "off by one" errors.

2.3. Simion versus Clone

There is no single, precise and agreed-upon definition of the term "clone" in the clone detection community

[16, 24–26]. Instead, many different definitions of the term "clone" have been proposed [16, 24]. However, most of them denote a common origin of the cloned code fragments [25], as is also the case in biology. In this paper, however, we want to investigate code similarities independent of their mode of creation. Using a term that in most of its definitions inside and outside of the clone detection community implies duplication from a single ancestor as a mode of creation is hence counter-intuitive. We thus deliberately introduce the term "simion" to avoid confusion.

For the sake of clarity, we relate the term to those definitions of "clone" that are most closely related.

Accidental clones denote code fragments that have not been created by copy&paste [1]. Their similarity results typically from constraints or interaction-protocols imposed by the same libraries or APIs they use. However, while they are similar w.r.t. those constraints or protocols, they need not be similar on the behavioral level[1].

Semantic clones denote code fragments whose program dependence graph fragments are isomorphic [7]. Since the program dependence graphs are abstractions of the program semantics, and thus do not capture them precisely, they can, but need not have similar behavior.

Type-4 clones as defined by [24] are comparable to simions. However, we prefer a term that does not include the word "clone" as this implies that one similar instance is derived from another which is not the case if they have been developed independently.

3. Study description

In order to gain a better understanding of the nature of simions, we use a study design with 3 research questions that guide the investigation.

3.1. Research questions

The underlying question we analyze is how syntactically different behaviorally similar code of independent origin really is. In order to investigate this, we answer the following 3 more detailed research questions.

RQ 1 *How successfully can existing clone detection tools detect simions that do not result from copy&paste?*

Multiple clone detectors exist that search for similar program representation to detect similar code. The first question we need to answer is how well these tools are able to detect simions that have not been created by copy&paste. If existing detectors perform well, no novel detection tools need to be developed.

[1] In other words, even though the code of two UI dialogs looks similar in parts since the same widget toolkit is used, the dialogs differ fundamentally in their visual appearance and behavior.

RQ 2 *Is program-representation-similarity-based clone detection in principle suited to detect simions that do not result from copy&paste?*

Having established that simions are often too syntactically different to be detected by existing clone detectors, we need to understand whether the limitations reside in the tools or in the principles. If the problems reside in the tools but the approaches themselves are suitable, no fundamentally new approaches need to be developed.

RQ 3 *Do simions that do not result from copy&paste occur in practice?*

The third question we address is whether simions occur in real world systems. From a software engineering perspective, the answer to this question strongly influences the relevance of suitable detection approaches.

3.2. Searching simions with existing tools (RQ1)

Study objects We created a specification for a simple email address validator function that was implemented by computer science students. The function takes a string containing concatenated email addresses as input. It extracts individual addresses, validates them and returns collections of valid and invalid email addresses. About 400 undergraduate computer science students were asked to implement the specification in Java. They were allowed to work in teams of two or three. Each team only handed in a single solution. Implementation was done under supervision by tutors to avoid copy&paste between different teams. Participation was voluntary and anonymous to reduce pressure to copy for participants that did not succeed on their own. Behavioral similarity was controlled by a test suite. Students had access to this test suite while implementing the specification. To simplify evaluation, students had to enter the implementation into a single file.

We received 156 implementations of the specification. Of those, 109 compiled and passed our test suite. They were taken as study objects. Since all objects pass our test suite, they are known to exhibit equal output behavior for the test inputs. Output behavior for inputs not included in the test suite can vary. Figure 2 displays the size distribution of the study objects (import statements are not counted). The shortest implementation comprises 8, the longest 55 statements. In Figure 3 the study objects are also categorized by nesting depth, *i. e.*, the maximal depth of curly braces in the Java code, and McCabe's cyclomatic complexity [20]. These metrics, which both measure certain aspects of the control flow of a program, already separate the study objects quite strongly, with the two largest clusters having size 19 and 12. When looking for implementations that are structurally the same, it can be expected that these give the same values for both metrics and thus the search could be limited to clusters with the same metric values (denoted by the bubbles in the diagram).

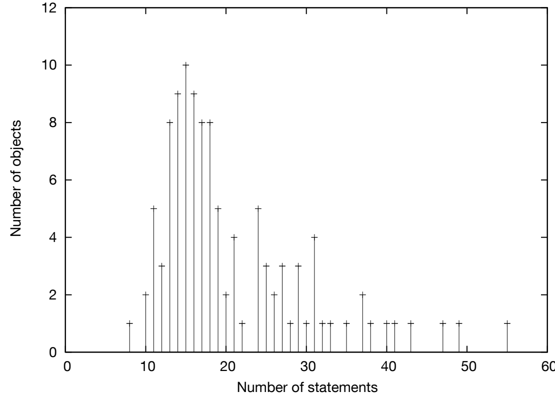

Figure 2. Size distribution of the study objects.

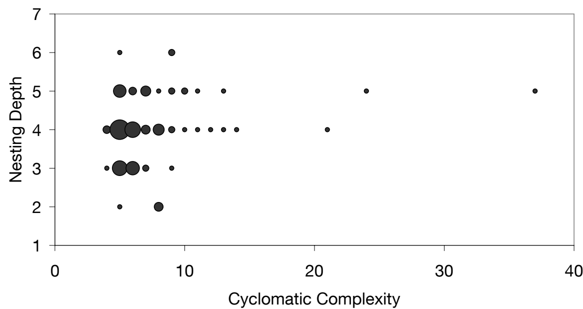

Figure 3. Study objects plotted by nesting depth and cyclomatic complexity. The area of each bubble is proportional to the number of study objects.

Study design To answer RQ1, we need to determine the recall of existing clone detectors when applied to the study objects. We denote two objects that share a clone relationship as a *clone pair*. Since we know all study objects to be behaviorally similar, we expect an ideal detector to identify each pair of study objects as clones. For our study, the recall is thus the ratio of detected clone pairs w.r.t. the number of all pairs. We compute the *full clone recall* and the *partial clone recall*. For the full clone recall, two objects must be complete clones of each other to form a clone pair. For the partial clone recall, it is sufficient if two objects share any clone (that does not need to cover them entirely) to form a clone pair. We included the partial clone recall, since even partial matches of simions could be useful in practice.

Procedure We chose ConQAT [12][2] and DECKARD [9] as state-of-the-art token-based and AST-based clone detectors. To separate clones between study objects from clones inside study objects, all clone groups that did not cover at least two different study objects were filtered from the results. The parameters used when running the detectors influence the detection results. Especially the minimal length parameter strongly impacts precision and recall. To ensure that we do not hereby miss relevant clones, we chose a very small minimal length threshold of 5 statements for ConQAT. To put this into perspective: when using ConQAT in practice [5, 13], we use thresholds between 10 and 15 statements for minimal clone length. Obviously such a small threshold can result in high false positive rates and thus low precision of the results. However, this affects the interpretation of the results w.r.t. the research question in a single direction only—if we fail to detect a significant number of clones even in presence of false positives, we cannot expect to detect more clones with more conservative parameter settings.

[2]The "CloneDetective" from the title of [12] is now part of ConQAT.

Results We executed ConQAT in three different configurations to detect clones of type[3] 1, types 1&2 and types 1-3. For type 3 clone detection, an edit distance of 33% of the length of the clone was accepted[4]. Partial clone recall was computed as the ratio of the number of pairs of study objects that share any clone, w.r.t. the number of all pairs. The full clone recall was computed as the ratio of the number of pairs of study objects that share clones that cover at least 90% of their statements w.r.t. to the number of all pairs. The number of all pairs is the number of edges in the complete undirected graph of size 109, namely 5778. DECKARD was executed with minimal clone length of 23 tokens (corresponding to 5 statements for an average token number of 4.5 per statement for the study objects), a stride of 0 and a similarity of 1 for detection of type 1&2 clones and 0.95 for detection of type 3 clones. Again, these values are a lot less restrictive than the values suggested in [17]. Since the version of DECKARD used for the study cannot process Java 1.5, it could not be executed on all 109 study objects. Instead, it was executed on 50 study objects that could be made Java 1.4 compatible by removal of type parameters[5]. For the 50 study objects, the number of all pairs is 1225. The results are depicted in table 1.

As can be expected, the recall values for clones of type 1-3 are higher than for type 1 or type 1&2 clones. Furthermore, the AST-based approach yields slightly higher values. This is not surprising since it performs additional normalization. However, even though we used very tolerant pa-

[3]Clone types classify the differences between clones: type 1 is limited to differences in layout and comments, type 2 further allows identifier renames and type 3 in addition allows statement changes, additions or deletions. Refer to [16] for details.

[4]As for minimal clone length, this value is more tolerant than what we typically employ in industrial settings [13].

[5]The remaining 59 study objects used additional post Java 1.4 features and were excluded from the study

Table 1. Results from clone detection

Detector	*Detected Clone Types*	*Partial Clone Recall*	*Full Clone Recall*
ConQAT	1	0.4%	0.0%
ConQAT	1&2	2.3%	0.0%
ConQAT	1-3	3.2%	0.1%
DECKARD	1&2	5.1%	0.1%
DECKARD	1-3	9.7%	0.8%

rameter values for clone detection, which probably result in a false positive rate that is too high for application in practice, both partial and full clone recall values are very low. The best value for full clone recall is below 1%, the best value for partial clone recall below 10%.

In other words: for two arbitrary study objects, the probability that any clones are detected between them is below 10%. The probability that they are detected to be full clones of each other is even below 1%. Given the very tolerant parameter values used during detection, we cannot expect these tools to be well suited for the detection of simions (not created by copy&paste) in real world software.

3.3. Limits of representation-based detection (RQ2)

Having established that existing tools are not well suited to detect simions, we investigate whether the causes reside in tool implementations or if their underlying detection approaches are fundamentally unsuited.

Study objects & design The study for RQ2 is performed on the 109 implementations of the email address validator function from RQ1 and comprises two parts. First, we collect differences between study objects. We categorize them based on their compensability. As stated above, to the best of our knowledge, there is no established formal boundary on the capabilities of program-representation-similarity-based (PRSB) detection approaches. Consequently, instead of using a formal boundary, we base the categorization on the capabilities of existing approaches. For that, we consider approaches not only from clone detection, but also from the related research area of algorithm recognition.

Second, having established and categorized these factors, we can look beyond the limitations of existing tools and can determine how well an ideal PRSB clone detection tool can detect simions. To that end, the differences between pairs of study objects are rated based on their category. This is performed by manual inspection. The ratio of pairs that only contain differences that can be compensated w.r.t. all pairs is computed. It is an upper bound for the recall PRSB approaches can in principle achieve on the study objects.

Procedure To keep inspection effort manageable, manual inspection was carried out on a random sample of study objects. The sample was generated in such a way, that each study object occurred at least once and contained 55 pairs. The study objects of each pair were compared manually and the differences between them recorded. As a starting point for the difference categorization, we used the categories of program variation proposed by Metzger and Wen [22] and Wills [28]. If the differences in a category can be compensated by any existing clone detection approach or by existing work from algorithm recognition, we classified it as within reach of PRSB approaches. Else, we classified the category as out of reach of PRSB approaches.

Categories of program variation The following list shows the categorization of differences encountered during manual inspection of pairs of study objects that were considered principally within reach of PRSB approaches. Examples with line number references of the form A-xx and B-yy refer to study objects A and B in Fig. 4.

Syntactic variation occurs if different concrete syntax constructs are used to express equivalent abstract syntax, such as the different statements used to create an empty string array in lines A-4 and B-4. In addition, it occurs if the same algorithm is realized in different code fragments by a different selection of control or binding constructs to achieve the same purpose. Examples are the implementation of the empty string checks as one (line B-3) or two *if* statements (lines A-3 and A-5) or the optional *else* branch in line B-6. Means to compensate syntactic variation include conversion into intermediate representation and control flow normalization [22].

Organization variation occurs if the same algorithm is realized using different partitionings or hierarchies of statements or variables that are used in the computation. In line B-14 for example, a matcher is created and used directly, whereas both the matcher and the match result are stored in local variables in lines A-17-19. Means to (partial) compensation include variable- or procedure-inlining and loop- and conditional distribution [22].

Generalization comprises differences in the level of generalization of source code. Types *List<String>* in line A-8 and *ArrayList<String>* in line B-8 are examples of this category. Means of compensation include replacements of declarations with the most abstract types, or, in a less accurate fashion, normalization of identifiers.

Delocalization occurs since the order of statements that are independent of each other can vary arbitrarily between code fragments. In a clone of study object A for example, the list initialization in line A-8 could be moved behind line A-14 without changing the behavior. Delocalization can *i. e.*, be compensated by search for subgraph isomorphism as done by PDG-based approaches [16, 24].

Unnecessary code comprises statements that do not affect the (relevant) IO-behavior of a code fragment. The debug statement in line A-14 for example can be removed without changing the output behavior tested for by the test

```java
1  public String[] validateEmailAddresses(String
       addresses, char separator, Set<String>
       invalidAddresses) {

3    if (addresses == null)
4      return new String[0];
5    if (addresses.equals(""))
6      return new String[0];

8    List<String> valid = new ArrayList<String>();

10   String sep = String.valueOf(separator);
11   if (separator == '\\')
12     sep = "\\\\";
13   String[] result1 = addresses.split(sep);
14   System.out.println(Arrays.toString(result1));

16   for (String adr : result1) {
17     Matcher m = emailPattern.matcher(adr);
18     boolean ergebnis = m.matches();
19     if (ergebnis)
20       valid.add(adr);
21     else
22       invalidAddresses.add(adr);
23   }

25   return valid.toArray(new String[0]);
26 }
```

```java
public String[] validateEmailAddresses(String                  1
    addresses, char separator, Set<String>
    invalidAddresses) {

  if(addresses == null || addresses.equals("")) {              3
    return new String[]{}; }                                   4

  else {                                                       6
    addresses.replace(" ", "");                                7
    ArrayList<String> validAddresses = new                     8
        ArrayList<String>();

    StringTokenizer tokenizer = new                            10
        StringTokenizer(addresses, String.valueOf(
        separator));

    while(tokenizer.hasMoreTokens()) {                         12
      String i = tokenizer.nextToken();                        13
      if(this.emailPattern.matcher(i).matches()){              14
        validAddresses.add(i);                                 15
      } else {                                                 16
        invalidAddresses.add(i);                               17
      }                                                        18
    }                                                          19

    return validAddresses.toArray(new String[]{});             21
  }                                                            22
}                                                              23
```

Figure 4. Study objects A and B

cases[6]. Means of compensation include backward slicing from output variables to identify unnecessary statements.

The following category contains types of program variation in the study objects that cannot be compensated by existing clone detection or algorithm recognition approaches.

Different data structure or algorithm: Code fragments use different data structures or algorithms to solve the same problem. One example for the use of different data structures encountered in the study objects is the concatenation of valid email addresses into a string that is subsequently split, instead of the use of a list. The use of different algorithms is illustrated by the various techniques we found to split the input string into individual addresses: in line A-13, a library method on the Java class *String* is called that uses regular expressions to split a string into parts. In line B-10, a *StringTokenizer* is used for splitting that does not use regular expressions. To illustrate the amount of variation that can be found even in a small program, Figures 5–9 depict different ways to implement the splitting. All examples were found in the study objects.

Inspection Result Of the 55 pairs of study objects inspected manually, only 4 did not contain program variation of category *different algorithm or data structure*. In other words, only about 7% of the manually inspected pairs contain only program variation that can (in principle) be compensated. Since this ratio is an upper bound on the recall PRSB approaches can in principle achieve, we consider PRSB approaches poorly suited for detection of simions that do not result from copy&paste.

3.4. Simions in real world software (RQ3)

RQ1 and RQ2 demonstrated that existing clone detection approaches are ill-suited to detect simions of independent origin. However, if this type of redundancy appears in real world software, *i. e.*, outside of our experiment, is still unclear. Lacking a detection tool, we investigated this question through manual techniques.

Study Objects, Design & Procedure To identify simions in a real-world system, we pair-reviewed the source code of the well-known open-source reference manager JabRef[7]. Obviously, the identification of simions is a hard problem as it requires full comprehension of the source code. As we did not know the source code of JabRef before, we limited our review to about 6,000 LOC[8] that contain utility functions that are mainly independent of JabRef's domain. Examples are functions that deal with string tokenization or with file system access. In this review, we did not only analyze if reviewed parts themselves contain simions but also took into

[6]Depending on the use case, debug messages can or can not be considered as part of the output of a function.

[7]`http://jabref.sourceforge.net/`

[8]Lines of code

```
String[] adresses2 = addresses.split(Pattern.
    quote(String.valueOf(separator)));
```

Figure 5. Splitting with java.lang.String.split()

```
ArrayList<String> validEmails = new ArrayList<
    String>();
StringTokenizer st = new StringTokenizer(
    addresses, Character.toString(separator));
while (st.hasMoreTokens()) {
  String tmp = st.nextToken();
  validEmails.add(tmp);
}
```

Figure 6. Splitting with java.util.StringTokenizer

```
List<String> result = new ArrayList<String>();
int z = 0;
for (int i=0; i<addresses.length(); i++) {
  if (i==addresses.length()-1) {
    result.add(addresses.substring(z, i+1));
  }
  if (addresses.charAt(i)==separator) {
    result.add(addresses.substring(z, i));
    z=i+1;
  }
}
```

Figure 7. Splitting with custom algorithm 1

```
List<String> curAddrs = new ArrayList<String>();
String buffer = "";
for(int i=0; i<addresses.length(); i++) {
  if(addresses.charAt(i) != separator) {
    buffer += addresses.charAt(i);
  } else {
    curAddrs.add(buffer);
    buffer = "";
  }
}
curAddrs.add(buffer);
```

Figure 8. Splitting with custom algorithm 2

```
List<String> emailListe= new ArrayList<String>();
int trenneralt = 0;
while (addresses.indexOf(separator, trenneralt) !=
     -1) {
  int trennerneu = addresses.indexOf(separator,
      trenneralt);
  emailListe.add(addresses.substring(trenneralt,
      trennerneu));
  trenneralt = trennerneu + 1;
}
```

Figure 9. Splitting with custom algorithm 3

account code that is behaviorally similar to library code as provided, for example, by the Apache Commons Library[9]. Such findings identify missed reuse opportunities.

Results Using manual reviews, we found multiple simions within JabRef's utility functions. An example is the function *nCase()* in the *Util* class that converts the first character of a string to upper case. The same functionality is also provided by class *CaseChanger* that allows to apply different strategies for changing the case of letters to strings.

Even more interesting, we found a large number of utility functions that are already provided by well-known libraries like the Apache Commons. For example, the above method is also provided by method *capitalize()* in the Apache Commons class *StringUtils*. The class *Util*, in particular, exhibits a high number of simions. It has 2,700 LOC and 86 utility methods of which 52 are not related to JabRef's domain but deal with strings, files or other data structures that are common in most programs. Of these 52 methods 32 exhibit, at least partly, a behavioral similarity to other methods within JabRef or to functionality provided by the Apache Commons library. Eleven methods are, in fact, behaviorally equivalent to code provided by Apache. Examples are methods that wrap strings at line boundaries or a method to obtain the extension of a filename.

Many of these methods in JabRef exhibit suboptimal implementations or even defects. For example, some of the string-related functions use a platform-specific line separator instead of the platform-independent one provided by Java. In another case, the escaping of a string to be used safely within HTML is done by escaping each character instead of using the more elegant functionality provided by Apache's *StringEscapeUtils* class. A drastic example is the JabRef class *ErrorConsole.TeeStream* that provides multiplexing functionality for streams and could be mostly replaced by Apache's class *TeeOutputStream*. The implementation provided by JabRef has a defect as it fails to close one of the multiplexed streams. Another example is class *BrowserLauncher* that executes a file system process without making sure that the standard-out and standard-error streams of the process are drained. In practice, this leads to a deadlock if the amount of characters written to these streams exceeds the capacity of the operating system buffers. Again, the problem could have been avoided by using Apache's class *DefaultExecutor*.

While the manual review of JabRef is not representative, it indicates that real-world programs, indeed, exhibit simions. While some of the simions are also representationally similar, the majority could not be identified with clone detection tools. This applies in particular for the simions that JabRef shares with the Apache Commons as the code has been developed by different organizations. A central in-

[9] http://commons.apache.org/

sight of our manual inspection was, that simions often represent missed reuse opportunities that do not only increase development efforts but also introduce defects.

4. Threats to Validity

This section discusses how we mitigated threats to validity.

Construct Validity For RQ1, we did not measure the impact of the parameters used for detection on precision. This has two reasons. (1) precision measured on the study objects, which are known to be behaviorally similar, is unlikely to be transferable to real world software, where we cannot expect the same amount of similarity. Precision measures would thus have to be repeated on further systems, still with questionable transferability beyond the systems under study. (2) Measuring precision through manual assessments is already difficult in general [27]. During the course of the study, we found it to be infeasible for very small clones (*e. g.*, of size below 4 statements) due to low inter-rater reliability. Instead, we chose very tolerant parameter values that, while likely to result in low precision, are unlikely to reduce recall. However, this strategy has a single sided effect on the results of the study in that it merely increases the probability to detect clones. It thus does not affect the validity of the results that existing tools are poorly suited to detect simions.

Internal Validity For RQ2, we classified categories of program variation according to whether they are in principle within reach of PRSB approaches. Misclassification can impact the results. We handled this threat by choosing a conservative classification strategy. Categories that can only partly be handled (*e. g.*, due to the use of heuristics that cannot guarantee completeness or high computation complexity that could be prohibitively expensive in practice) were rated as within reach of PRSB approaches. In addition, differences between the study objects that stemmed from differences in their behavior that were not detected by our test suite were ignored. This conservative strategy thus increases the probability to consider PRSB approaches as suited for the simion detection problem. It does, however, not impact the validity of the result that PRSB approaches are poorly suited for the simion detection problem.

Several factors can lead to less program variation among the study objects than could typically be encountered in real world software: (1) all students had access to the same test suite, (2) the signature of the validator function, including its types, was specified, (3) teams could ask tutors for help. However, all these factors only increase our chances of finding clones and thus do not invalidate the results.

External Validity We chose two state-of-the-art clone detectors for the study. Some detector we did not try might perform better. However, given the diversity and amount of program variation we discovered among the study objects, we do not expect any existing clone detector to perform substantially better, as would be required to invalidate our conclusions. The results for RQ2 illustrate that this is also valid for PDG-based detectors[10]. We do not claim transferability of the actual numbers (*e. g.*, recall measures) we measured on the study objects beyond the study. However, since the study objects were relatively simple compared to real world software, we do expect it to exhibit less program variation. On the contrary, we would expect program variation to be even larger for real world software, due to differences in conventions and practices between different teams and domains. Regarding the existence of simions in real- world programs that are not the result of copy&paste (RQ3), our approach can only provide an indication. In particular, it is too early to reason about the defect proneness of the missed reuse opportunities represented by simions.

5. Discussion

In the previous sections we explored the limits of current clone detection tools and also of their underlying approaches. In our experiment clone detection tools achieve a recall of less than 1% when analyzing behaviorally similar but independently developed code (RQ1). While it could have been expected that existing clone detection approaches have rather limited capabilities for finding simions, the dramatically low recall is nevertheless surprising. Moreover, the result of RQ2 shows that only a certain class of simions, those that are representationally similar modulo normalization, can be found with current clone detection approaches. Hence, we are inclined to disagree with [24] that states that "[. . .] attempts can be made to detect semantic clones [simions] by applying extensive and intelligent normalizations to the code.".

Furthermore, RQ1 demonstrated that independent programmers do not tend to create representationally similar code when facing the same implementation problem. Thus, we would expect to find simions "in the wild" which are not representationally similar and thus not detectable by current tools. RQ3 provides first indications for this fact. These results are also backed up by the study in [10], which mined a huge number of simions from the Linux kernel sources from which at least half of them where not representationally similar. Results that point in the same direction are also presented by Kawrykow and Robillard that report on significant amounts of reimplemented API methods they found in Java systems [14].

The simions inspected for RQ3 also confirmed our expectations that reuse of existing (library) functions often not only reduces implementation efforts but also the number of bugs. To provide some further indication, we used Google Code Search[11] to identify other Java programs that do not

[10] Also, we are not aware of an available PDG-based detector for Java.

[11] `http://www.google.com/codesearch`

reuse Apache's *DefaultExecutor* and exhibit the same deadlock problem as JabRef that we discovered in RQ3. Strikingly, of the first 10 hits for the search *lang:java process.waitfor*, 6 implementations contain exactly the same problem as JabRef although only 2 of them appear to be the result of copy&paste.

These facts indicate the detection of simions to be a practically relevant problem which is not yet solved by existing tools. The existing approaches discussed in the next section all focus on specific kinds of simions and usually have false positive rates which are way too large to be practically applicable. A working simion detector could not only help in reducing code size by eliminating redundant code, but also find bugs by including libraries of working code or bug patterns in the detection. Finally, such a tool would allow a more quantitative study of the amount and nature of simions in real-world software projects – something that could not be performed in this paper and was only partially done by [10]. So we consider the construction of algorithms and tools for simion detection a worthwhile and still open problem.

6. Related Work

First results of this work were published in an early version of this paper [11][12]. Other related publications are summarized in this section.

Clone Detection Comprehensive surveys of existing program-representation-similarity-based clone detection approaches is given in the surveys from Koschke [16] and Roy and Cordy [24]. For the experiment presented in this paper, the tools ConQAT [12] and DECKARD [9] have been used, which represent the most up-to-date implementations of token-based and AST-based clone detection algorithms, respectively.

Simion Detection Multiple authors dealt with the problem of finding behaviorally similar code, although often only for a specific kind of similarity.

An early paper on the subject by Marcus and Maletic [18] deals with the detection of so called *high-level concept clones*. The approach is based on reducing code chunks (usually methods or files) to token sets, and performing latent semantic indexing (LSI) and clustering on these sets to find parts of code which use the same vocabulary. The paper reports on finding multiple list implementations in a case study, but does not quantify the number of clones found or the precision of the approach. Limitations are identified especially in the case of missing or misleading comments, as these are included in the clone search.

The work of Kawrykow and Robillard [14] aims at finding methods in a Java program which reimplement functions available in libraries (APIs) used by the program. Therefore, methods are reduced to the set of classes, methods, and fields used, which are extracted from the byte-code, and then matched pairwise to find similar methods. Additional heuristics are employed to reduce the false positive rate. Application to ten open source projects identified 405 "imitations" of API methods with an average precision of 31% (worst precision 4%).

Nguyen et al. [23] apply a graph mining algorithm to a normalized control/data-flow graph in order to find "usage patterns" of objects. The focus of their work is not the detection of cloning, but rather of similar but inconsistent patterns, which hint at bugs. The precision for this process is about 20%[13].

The paper [10] by Jiang et al. introduces an approach that can be summarized by *dynamic equivalence checking*. The basic idea is, that if two functions are different they will return different results on the same random input with high probability. Their tool, called EQMINER, detects functionally equivalent functions in C code dynamically by executing them on random inputs. Using this tool, they find 32,996 clusters of similar code in a subset of about 2.8 million lines of the Linux kernel. Using their clone detector DECKARD they report that about 58% of the behaviorally similar code discovered is syntactically different. Since no systematic inspection of a sample of the clusters is reported, no precision numbers are available.

While the papers listed here propose new detection approaches, this work focuses on the limitations of existing approaches, providing rationale for the development of new algorithms for finding simions.

Studies on similarity of independently developed code In [6], Eckhardt et al. report on an experiment that evaluates n-version programming as a method of introducing fault-tolerance into software. For that purpose, 27 implementations of the same specification were created independently by different student groups. On them, *behavioral* similarity, namely the independence of programming errors, was evaluated. In contrast, this work investigates *representational* similarity of independently developed software.

In [1], Al-Ekram et al. search for cloning between different open-source systems using a token-based clone detector. They report that, to their surprise, they found little behaviorally similar code across different systems, although the systems offered related functionality. The clones they did find were typically in areas where the use of common APIs dictated a certain representation, thereby limiting program variation. They thus give further indication that (except in presence of canonizing factors such as common APIs), existing clone detectors are ill-suited to detect simions that were not created by copy&paste.

[12] Since at that time not all study objects had been handed in by the students, the numbers differ between the papers.

[13] When including "code that could be improved for readability and understandability" as flaws, the paper reports near 40% precision.

Algorithm recognition The goal of algorithm recognition [2, 22, 28] is to automatically recognize different forms of a known algorithm in source code. Just as clone detection, algorithm recognition has to cope with program variation. The categorizations of program variation from [28] and [22] provided valuable input for this paper. The most fundamental difference w.r.t. similar code detection is that for algorithm recognition as proposed by [22,28], the algorithms to be recognized need to be known in advance.

7. Conclusion

This paper sheds light on the differences between syntactical/representational and semantic/behavioral similarity of code and the detectability of these similarities. With a controlled empirical experiment we underpin the common intuition of the existence of behaviorally similar redundant code which can not be found automatically by existing approaches currently in use by the clone detection community. Due to the impact of redundancy on maintenance efforts and program correctness, we consider the reliable and scalable automatic detection of simions a relevant open research topic. Our future work will investigate in how far dynamic approaches can be applied to the detection of simions in practice.

Acknowledgments The authors want to thank the students and tutors for their efforts involved in the creation of the study objects and the anonymous reviewers and participants of IWSC 2009 for helpful comments on a precursory position paper on this topic [11]. In addition, the authors would like to thank Ira Baxter, Moritz Beller, Manfred Broy, Raimar Falke, Nils Goede, Jan Harder, Rainer Koschke, Jens Krinke, Tobias Nipkow and Rebecca Tiarks for helpful advice. Furthermore, the authors are grateful to Lingxiao Jiang for providing DECKARD and giving helpful advice on its use. This work has partially been supported by the German Federal Ministry of Education and Research (BMBF) in the project Quamoco (01 IS 08023B) and by a Google Research Award.

References

[1] R. Al-Ekram, C. Kapser, R. Holt, and M. Godfrey. Cloning by accident: an empirical study of source code cloning across software systems. In *Proc of ESEM'05*, 2005.

[2] C. Alias and D. Barthou. Algorithm recognition based on demand-driven data-flow analysis. In *WCRE'03*, 2003.

[3] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM'98*, 1998.

[4] E. Clarke, O. Grumberg, and D. Peled. *Model checking*. Springer, 1999.

[5] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. M. y Parareda, and M. Pizka. Tool support for continuous quality control. *IEEE Software*, 25(5):60–67, 2008.

[6] D. E. Eckhardt, A. K. Caglayan, J. C. Knight, L. D. Lee, D. F. McAllister, M. A. Vouk, and J. J. P. Kelly. An experimental evaluation of software redundancy as a strategy for improving reliability. *IEEE TSE*, 17, 1991.

[7] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *ICSE '08*. ACM, 2008.

[8] I. I. Ianov. On the equivalence and transformation of program schemes. *Commun. ACM*, 1958.

[9] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *ICSE'07*, 2007.

[10] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. In *ISSTA'09*, 2009.

[11] E. Juergens, F. Deissenboeck, and B. Hummel. Clone detection beyond copy & paste. In *IWSC'09*, 2009.

[12] E. Juergens, F. Deissenboeck, and B. Hummel. CloneDetective – a workbench for clone detection research. In *ICSE'09*, 2009.

[13] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *ICSE '09*, 2009.

[14] D. Kawrykow and M. Robillard. Improving API usage through detection of redundant code. In *ASE'09*, 2009.

[15] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. *Pattern matching for clone and concept detection*, pages 77–108. 1996.

[16] R. Koschke. Survey of research on software clones. In *Duplication, Redundancy, and Similarity in Software*. Dagstuhl Seminar Proceedings, 2007.

[17] E. C. Lingxiao Jiang, Zhendong Su. Context-based detection of clone-related bugs. In *ESEC/FSE'07*, 2007.

[18] A. Marcus and J. I. Maletic. Identification of high-level concept clones in source code. In *ASE '01*, 2001.

[19] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *ICSM '96*, 1996.

[20] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[21] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE TSE*, 2004.

[22] R. Metzger and Z. Wen. *Automatic algorithm recognition and replacement*. MIT Press, 2000.

[23] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *FSE'09*, 2009.

[24] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical Report 541, Queen's University at Kingston, 2007.

[25] A. Walenstein. Code clones: Reconsidering terminology. In *Duplication, Redundancy, and Similarity in Software*, Dagstuhl Seminar Proceedings, 2007.

[26] A. Walenstein, M. El-Ramly, J. R. Cordy, W. Evans, K. Mahdavi, M. Pizka, G. Ramalingam, J. W. von Gudenberg, and T. Kamiya. Similarity in programs. In *Duplication, Redundancy, and Similarity in Software*, Dagstuhl Seminar Proceedings, 2007.

[27] A. Walenstein, N. Jyoti, J. Li, Y. Yang, and A. Lakhotia. Problems creating task-relevant clone detection reference data. In *WCRE '03*, 2003.

[28] L. Wills. Flexible control for program recognition. In *WCRE'93*, 1993.