Testing Semantic Clone Detection Candidates

Steven Raemaekers
Philips Healthcare
R&D FXD Best
University Of Amsterdam

Abstract

In this study, a comparative analysis of automated semantic clone detection techniques is performed. Semantic clones are pieces of code that fulfill the same requirement and could therefore be considered as functionally redundant. Reducing the number of semantic clones could lead to cost reduction and higher maintainability.
Semantic clones were detected manually in three open source projects, and results were verified by a number of independent observers. Secondly, automated techniques were performed on the same code samples. Test characteristics like precision and recall were calculated based on comparison of automated test outcomes to manual results.
Results show that in our set of test cases, matching tokens exactly is the only significant predictor for semantic clones. Including dictionary matches of words which were not directly matched did not improve performance of this test.

1. Introduction

Research shows that a considerable part of software consists of code clones: this number can be as big as 5 to 20%. These clones are introduced when a programmer copies, pastes and adjusts existing code [21, 22].
This leads to higher maintenance costs [7] since all changes to it have to be applied multiple times on multiple places in the code. It can also introduce subtle differences between code clones, which will make the code harder to maintain [19, 20]. Clones are also believed to cause architectural degradation [18]. Detecting code clones can therefore be interesting because reducing the number of code clones can eventually lead to reduced maintenance costs and prevent architectural degradation.

The same arguments can be used for duplicated functionality. When duplicate functionality could be detected and removed a program would contain the same functionality with less code. Such a situation could emerge when, inside a company, two software projects that have large parts in common are merged. This decision may be made by project management in order to save resources or to lower maintenance costs. Another situation could be a company merger in which one company acquires another. Either way, projects are selected to be merged, and there is a big chance that there exists some kind of functional resemblance between both projects. Merging duplicate functionality may also reduce maintenance cost and simplify architecture.

A couple of different ways to automatically detect code clones have been proposed in literature, which are all based on comparison of syntax:

String-based techniques perform a comparison based on characters. This technique stays most close to the original source code. Comparing two strings is usually done with a technique that calculates some form of edit distance [21, 23].

Token-based techniques split up source code in tokens (as done by a lexer). Whitespace and semicolons often serve as token separators. Different calculations on these tokens can be performed, as in [14].

Metric-based techniques create a “fingerprint” based on metrics of the code. Metrics can include numbers like the fan-in (number of places at which this function is called, in case of a function) and fan-out (number of other functions this function calls) [22].

Tree-based techniques parse the source code and create an Abstract Syntax Tree (AST). Some kind of distance function between two AST’s can be calculated, or metrics can be compared [9].

PDG-based techniques create a Program Dependence Graph. A PDG is not sensitive to reordering of statements, and makes data and control flow dependencies explicit. This is the most abstract technique that is used today [6].

All methods eventually obtain differences between pieces of code based on a certain measure, and select pairs in which these differences are the smallest. The selected cases are considered to be code clones.

```
mp_or(const MPNumber *x, const
     MPNumber *y, MPNumber *z)
{
  mp_bitwise(x, y, mp_bitwise_or,
             z, 0);
}
```

```
static KNumber ExecOr(const KNumber &
     left_op, const KNumber & right_op)
{
  return (left_op | right_op);
}
```

Figure 1. Example of a semantic clone. Above a sample from gcalctool, below from kcalc.

The problem with these techniques is that they are in essence syntactical measures. Text comparison is a technique that is purely based on syntax and which is highly sensitive to changes in source code. PDG-based techniques perform a comparison on the highest level of abstraction. It is capable of dealing with reordering of statements, but comparison of individual statements is eventually string-based.

We believe that previously mentioned techniques can be described as being able to detect “syntactic clones”, and are based on similarities in syntax. On a higher level would be techniques that detect clones based on similarity in functionality, which we will call “semantic clones”. We define a semantic clone as two pieces of code which fulfill the same requirements. Figure 1 shows an example of two pieces of code that we regard to be a semantic clone. The requirement which is implemented in these pieces of code is that a desktop calculator must be capable of calculating the binary (bitwise) OR-value of two numbers.

We are interested to find out to what degree automated techniques are capable of detecting semantic clones. Since source code is the only input to our automated tests, we use syntactic measures as a pointer to semantic resemblance. We are aware of the fact that semantic comparison is an unsolvable problem, but we nevertheless try to construct tests that are cheap to perform and give reasonable results. In this study we only perform a static syntactical analysis and we omit analysis of runtime behavior. We also choose to avoid a strict mathematical definition which includes semantic functions. We draw conclusions from statistical means. For a more detailed explanation see section **Discussion**.
As shown in research, each automated technique as mentioned above is proven to be capable of detecting *syntactic* clones. How good they are in detecting *semantic clones* as defined by us is not known. We make the assumption that humans are better capable of detecting functional redundancy than computers. We want to find out how close automated techniques get to human judgment.

In this study, semantic clones will be detected manually in three open source projects by a number of independent observers. Automatic tests will be performed on the same projects. By combining manual with automatic results, test characteristics of a couple of different tests will be calculated. Finally, ways to improve these tests will be discussed.

2. Related work

As stated in the introduction, current code clone research uses different methods for detecting code clones. In these methods a distinction between syntactic and semantic clones is almost never made, at least not in the way we make it. Most authors implicitly assume a definition of a clone which results from the detection technique used. If a token-based comparison technique is used, a code clone is considered to be two pieces of code having a score for a token-based test below a certain threshold [32], although this is often not explicitly stated as such. If a string-based technique is used, a code clone is considered to be two identical strings [33], or two substrings that have a metric score below a certain threshold, thus differing only a minimal amount in characters.

The most abstract definition of a code clone in use today are two pieces of code having isomorphic PDG’s [6]. This definition may come more close to semantics as we defined it than plain string comparison, but still it is not perfect. It does not take into account the fact that the same functional requirements can be fulfilled by two systems which have completely different designs, which would result in two functions with completely different PDG’s.

In literature, several levels of detection (“granularity”) have been tried, like detection at token [14], line [23], subtree [9], method [22], class [26] and file [27] level, although no comparison between different granularity levels in the same study is performed. We expect that choice of granularity influences test results tremendously. We include file and function level detection in our analysis to test this influence.

Current state-of-the-art tools are CCFinder [14] and CP-Miner [29], which both use a token-based technique for detecting code clones. This eventually comes down to counting the number of matching tokens between a pair of potential code clones. DECKARD [12] calculates a characteristic vector of abstract syntax trees. This method basically performs a count of node types in an abstract syntax tree. For a more complete description of this technique, see section **Techniques**.
Point is that only the most trivial copy-and-paste code can be detected with these tools, which can often be described as containing some insertions, substitutions and deletions. In this study, we try to raise the standard for the type of clones tests will return. Semantic clones could provide more added value than clones as returned by software previously mentioned. This would mean a step forward in machine understanding of source code.

We are realistic about the goal of our study and we do not expect to create a test which has full semantic awareness. We set ourselves the goal of creating some tests, calculating test characteristics and comparing them to human judgment, which we consider to be the gold standard of detection of functional redundancy. Accuracy of our analysis is ensured by calculating statistical significance for tests in comparison to chance (flipping a coin). New is a detection method which uses a dictionary to look up related words. Finally, since we perform a controlled experiment in which the total number of cases is known, we are able to include missing cases in our analysis as well (false negatives). In classic information retrieval experiments, this number is not always available [34].

3. Research method

We quantify previously mentioned comparison of automated test results with human judgment by using a binary classification setup. Resulting from this binary classification setup are test characteristics like sensitivity and specificity, which can all be calculated from the number of true/false positives/negatives in this setup. See figure 2 for a binary classification table.

In a binary classification test, test results (which can be either “yes” or “no” for a certain pair of code units) are compared with results from a gold standard or oracle, in this case manual detection. When an automatic test returns a score close enough to zero, two pieces of code are a clone according to this test. This is independent of the question whether or not a human would agree with this outcome. When an automatic test says that two pieces of code are a clone, and a human would agree, the number of true positives (TP) would be increased with one. In the same way the other numbers can be calculated.
In the table below results of all test input combinations would be shown. Test characteristics of a test cannot be calculated without setting a threshold value first, under which two inputs are a code clone, and above which they are not, according to this test. Test characteristics only depends on choice of a threshold value. A binary classification table (or sometimes called a *confusion matrix*) for a certain test and a certain threshold would look like figure 2. For more information on binary classification tests, see [30] and [31].

Type of test granularity		Manual detection	
		Yes	*no*
Test outcome	*Yes*	TP	FP
	No	FN	TN

Figure 2. Binary classification table/confusion matrix. TP = True Positives, FP = False Positives, FN = False Negatives, TN = True Negatives.

3.1 Manual detection

Manual detection was originally performed by a single observer. This raised questions about the external validity and reliability of our gold standard. This section will first discuss the gold standard as it was originally performed, and then improvements on the gold standard, which include asking multiple independent observers to judge code samples, and trying to including code samples in our analysis which we overlooked at first. All test scores as displayed in section **Results** are performed with the improved gold standard. For a more detailed discussion on this setup, see section **Improving the gold standard** in the discussion.

3.1.1 Single observer

To find semantic clones manually, we familiarized ourselves with the architecture of each project. This way the chance to find semantic clones would be increased since semantic clones could occur anywhere in a system, and understanding the architecture would make it more easy to find places where semantic clones were likely to occur.

We scanned source code for functionally equivalent files and functions. For files, especially filenames, function headers inside files, comments on top of a file and global function contents were examined. For functions, mostly function headers, function contents and comments on top of the function were examined. We decide to omit statement level because single statements depend too much on context to be compared individually.
We chose to omit class-level comparison because we focused on programs written in C, which do not support classes. Multi-word and multi-statement levels were also omitted due to the combinatorial explosion which would result in testing all possible combinations.
Questions There are a couple of questions we considered to be helpful in determining whether two pieces of code could be considered a semantic clone. These questions are shown in figure 3. The rationale behind these questions is the following.
First, two pieces of code were considered to be semantically identical if they are syntactically identical. It is not possible for two pieces of identical code to perform different functions. If they are not identical, the smaller the edit distance between two pieces of code is, the more chance there is they are a syntactic clone and thus a semantic clone. Since judgment is subjective in nature we dot not define which exact edit distance we considered to be small, but the smaller it is the more chance it can be considered a clone.
If pre- and post conditions were known or could be determined and turned out to be the same, two pieces of code were considered to be a semantic clone. From a mathematical viewpoint, two functions that only differ in preconditions are two completely different functions. From a pragmatic viewpoint, each case in which only preconditions are different is judged separately. For instance, assertions of variable values may take place inside a function itself, but we did not regard this to be contributing to main functionality. These cases were still considered to be semantic clones, despite the fact that they mathematically do not have the same preconditions.
Two pieces of code can perform basically the same functionality and execute some extra tasks along the way. These extra tasks do not have to be the same for both functions. As long as the main task of both functions is the same, we consider them to be semantic clones. When it is easy to remove these extra tasks from one piece of code and merge the two functions while positioning the extra functionality somewhere else, this is an indication that two pieces of code are semantic clone candidates. Examples of extra tasks that are considered to be insignificant side effects and do not contribute to main functionality are logging, statistics or assertions.
It is a convention in the C programming language to put two pieces of code which are functionally related in the same translation unit (C file), but this is never enforced by the compiler. It can therefore only serve as a hint to the location of possible semantic clones. If two pieces of code are in the same translation unit, the chance that they are functionally related increases. This of course depends on system design and the specific file in which two pieces of

code appear. Often, when a file contains functionally related pieces of code, this is indicated with a comment on top of the file explaining the goal of it.
Names of function headers and variables are an indicator of functionality. We pay special attention to the occurrence of the same words or synonyms. Not all words are of equal importance since the most common English words (e.g. “the”, “and”) appear in almost all source code, regardless of them being semantic clones or not.

Note that these questions can only help as a guide in determining whether two pieces of code are a semantic clone or not. As said before, this process is per definition very hard to quantify and subjective in nature and will therefore give different results depending on the observer. To address this problem, multiple independent observers will be consulted to judge the same code samples. In figures 16, 17 and 18 results of manual collection for all projects are shown.

- *Are the two code units syntactically identical?*
- *If not so, is it easy to convert the first code unit into the second one (small edit distance)?*
- *Are pre- and post conditions given with source code?*
    - *If not, can they be deducted from source code?*
- *Are pre- and post conditions exactly the same?*
    - *Are only post conditions the same?*
- *Is base functionality of the two code units comparable?*
- *Is it possible to merge the two routines relatively easy by creating a wrapper function?*
- *Are the two code units located in the same translation unit (a single C file)?*
- *Are the two code units executed in the same functional context?*
- *Do the function headers show any signs of implementation of comparable requirements?*
- *Are there any significant tokens in the function body that match?*

Figure 3. Example questions for the manual detection of semantic clones. These questions can help judging code units but only served as a guide and are by no means complete.

Problems with a single observer as gold standard are further discussed in section **Discussion**. To address these problems, we tried to obtain a more reliable gold standard by consulting multiple independent observers. Next, this process will be described.

3.1.2 Multiple independent observers

A number of independent human observers are asked to fill in a form in which requirements have to be connected to two different implementations of this requirement. A sample form is shown in figure 4.

If an observer believes that function A.3 and function B.2 are two semantic clones which both implement requirement 1, the observer puts an 3 and a 2 on his answer form, respectively. This survey will be answered by a number of independent observers. Function bodies are also supplied to observers.

Requirement 1: Two numbers can be multiplied
A: *B:*
Requirement 2: The square root of a number can be calculated
A: *B:*

```
A.1: int divide(int a, int b)
A.2: int add(int a, int b)
A.3: int multiply(int a, int b)
B.1: int subtract(int x, int y)
B.2: int mult(int x, int y)
B.3: int add_numbers(int x, int y)
```

Figure 4. Example requirements and two lists of candidate functions. For clarity, function bodies have been omitted, which were included in the actual survey.

Desktop calculators	
DC1.	The value of numeric expressions can be evaluated
DC2.	After every calculation the latest result is displayed on screen
DC3.	A logical AND-operation can be performed on two numbers
DC4.	A logical OR-operation can be performed on two numbers
DC5.	Two numbers can be added
DC6.	Two numbers can be subtracted
DC7.	Operations can be reversed (undo)
DC8.	Previously reversed operations can be re-executed (redo)
DC9.	The accuracy of calculations can be set
DC10.	Text from the clipboard can be pasted into the application
DC11.	Calculations in other numeric bases can be performed (e.g. binary, hexadecimal)
Shells	
SH1.	Entered commands can be executed
SH2.	Errors can be printed to screen
SH3.	A hash table for fast lookup of defined commands can be created
SH4.	Text of previously entered commands is stored so they can be re-entered (“arrow-up” in a command line)
Text editors	
TE1.	A help screen can be displayed
TE2.	A message can be displayed showing parameters and options to enter through the command line (“--help” parameter)
TE3.	Macros can be defined which can be executed with keyboard shortcuts

Figure 5. Requirements as shown in the questionnaire.

We limit the number of functions in each list to no more than 25 because observers are expected to complete the questionnaire in approximately 20 minutes, and experiments with our survey show that it takes considerable time to match even a small amount of requirements to functions.
Functions are inserted in both lists in random order. To make answering the questionnaire more difficult, random requirements without matching functions are added to the

list. Random functions without semantic equivalents, and random functions without corresponding requirements are also added. Figure 5 shows a list of all requirements in the questionnaire.

Answers from all observers are analyzed, as shown in section **Results**, **Multiple independent observers**. Eventually, all combinations of code have to be labeled either being, or not being a semantic clone. This is visible in figure 2, where manual detection only allows two categories. Therefore, a way to merge answers from all observers will be discussed.

3.1.3 Adding previously undetected clones

Even with multiple observers, validity problems can arise. For a further discussion on the reliability of our gold standard, see section **Discussion**. In an attempt to improve the gold standard and to find out what clones we potentially missed, we mail the original authors of each project and ask them if they are aware of any functional resemblance between the project they work on and its functional equivalent (e.g. kcalc and gcalctool). We ask them to ignore obvious duplicate functionality (adding, subtracting, etc) which is directly linked to buttons on the screen, because we have found these cases ourselves already.

As it turns out, in both kcalc and gcalctool the same library for multiple precision arithmetic calculations is used. These functions were not included in our original analysis. According to our definition, these functions are syntactically identical and therefore semantically identical, see section **Manual detection, single observer**. We decided not to include these test cases in our test set because it is obvious that every automated test will automatically return scores of 0 on syntactic clones (exact copy-and-paste).

4. Techniques

In the previous section we described the manual detection process as performed by humans. To compare manual results with automated test results, automated test results have to be acquired. All tests that will be executed are discussed in this section.
For each test we describe the algorithm used to calculate a score, given two pieces of code as input. We also describe why we expect this technique to detect semantic clones.
From now on, we will call a piece of code a code unit. See **Definition 5.1 (Code Unit)**. A code unit can be the source text of a complete file or of a function. Note that tests in this study possess most properties of a metric, but some tests miss certain properties as described in **Definition 5.2 (Test)**. An example of one of such properties is that a score of 0 means that two code units are exactly the same (identity of indiscernibles). Strictly speaking the metric vector test cannot be considered a true metric, since a score of 0 for this test does not necessarily mean that two code units are identical (both code units can have identical vectors but still be completely different in their structure). We neverthelesss chose to include this test in our analysis because it can still contain useful properties which possibly makes it good at detecting semantic clones, despite the fact it does not fully conform to the definition of a metric.
The higher the score the more different two code units are, according to a certain test. In the definition of a metric there is no upper limit on scores, but due to the fact that most tests are normalized for length of both code units, most scores range from 0 to 1. This does not include the metric vector test, which is not normalized. See **Definition 5.2 (Test)**.

4.1 Edit Distance

The edit distance between two strings is the minimal number of insertions, deletions or substitutions needed to transform a string into another. See **Definition 5.6 (Levenshtein edit distance)**. Because length of both strings should be taken into account, final score is normalized for length of both strings, while maintaining characteristics of a metric [25]. See **Definition 5.10 (Normalized Edit Distance)**. Figure 6 shows the (unwanted) result of an unnormalized edit distance test.

$D_E(ab, ac) = 1$
$d_E(abcdefgX, abcdefgY) = 1$

Figure 6. The edit distance test should take string length into account. The first string pair differs 50%, the second pair much less than that, while edit distance is 1 in both cases.

A score of 0 means that two strings are identical, a score of 1 means that two strings are completely different. Since we defined syntactic clones as two character arrays having an edit distance of 0, this is a perfect test to detect syntactic clones. The question is how good this test performs at predicting semantic clones.

The reason we believe this test is able to detect semantic clones is that two code units which fulfill the same functionality are expected to be written down approximately the same way. This would result in lower edit distances for semantic clones.

A problem with this test could be that there are different ways to fulfill the same requirement, while edit distance between two different methods do not necessarily have to be small. This test may be considered to stay “too close” to syntax, because it does not take any semantic meaning into account. But whether or not this effect is significant remains to be seen. The algorithm of this technique is shown in figure 7.

4.2 Exact Token Match

This test counts the number of tokens of the first code unit that appear in the second code unit. Each code unit is considered to be a collection of tokens. It counts the size of the union of both collections and takes into account size

```
int LD (char s[1..m], char t[1..n])
{
   int d[0..m, 0..n];

   for i from 0 to m
     d[i, 0] := i;
   for j from 0 to n
     d[0, j] := j;

   for j from 1 to n
   {
      for i from 1 to m
     {
       if s[i] = t[j] then
         d[i, j] := d[i - 1, j - 1];
       else
         d[i, j] := min
                  (
                     d[i - 1, j] + 1,
                     d[i, j - 1] + 1,
                     d[i - 1, j - 1] + 1
                  );
      }
   }

   return (2 * d[m, n]) / (m + n + d[m, n]);
}
```

Figure 7. Normalized Levenshtein distance calculation

of the biggest collections. The result is a normalized score, where 1 means that no tokens in the smallest code unit appear in the largest, and 0 means that all tokens in the smallest code unit appear in the largest. See **Definition 5.11 (Exact Token Match Distance)**. It does not count tokens that are smaller than a certain length. The reason for a minimal size of tokens is that tokens smaller than 3, for instance, do not carry any semantic meaning, speaking in terms of natural language. This is demonstrated in figure 8.
The idea behind this test is the hypothesis that people tend to choose similar names for variables and functions when functionality is identical.

```
int a = i * j * s;
```

```
int totalTableSize = numberOfRows *
   numberOfColumns * cellSize;
```

Figure 8. Above a statement with small token size, carrying almost no semantic meaning. Below a comparable statement with meaningful names, also having greater average token size.

We do not know whether or not this test will be a significant predictor for semantic clones. In a normal code unit there are a lot of keywords and types that occur so often that this test is expected to return high amounts of false positives. These “aspecific” tokens are expected to pollute results, but unknown is whether or not this effect is strong enough to influence final test performance. Without this effect, this test is expected to be a significant predictor.

4.3 Kolmogorov LZMA

According to the Kolmogorov complexity theory, there exists a smallest lossless form of storage for every piece of information. Further compressing this piece of information without losing information is not possible.

```
“abababababab”
“6xab”
“4c1j5b2p0cv4w1x8”
```

Figure 9. Above an uncompressed string and a more compressed string, carrying the same information. The bottom string will be more difficult to compress any further, since it contains random numbers and characters.

The shortest notation for a piece of information cannot be determined, but the difference in Kolmogorov complexity between two code units can be approached by the *normalized compression distance*. See **Definition 5.9 (Normalized Compression Distance)**.
The reason why this test is supposed to detect semantic clones is the idea that two code units that are a semantic clone will carry the same amount of “information”, irrespective of the way information is written down in source code. By compressing this information, specific notation of this method could be ignored. When two code units are a semantic clone, both of them would compress to a file of the same size.
Compression can be applied to source code or to compiled object code. The advantage of using compiled object code is that the compiler has already applied optimizations, in which case two functional identical code units would (hopefully) transform into comparable object code. A presumption is that in both code units, a compiler with the same settings is used.

Due to the highly experimental nature of this test, it is highly unsure whether our theory will hold. Whatever the outcome, we expect more from compression distance between compressed object code files than plain source code. When the same compiler and compiler settings are used, functionally equivalent code is expected to compile to syntactically identical object code. A good example that supports this hypothesis is the fact that both while- and for-loops eventually compile to an assembly construction with a “jmp”-statement a label.

More aggressive compiler optimization settings are expected to decrease compression distance between two pieces of object code even further, since we expect that object code of two semantically identical code units will eventually converge to comparable syntactic assembly instructions. We hypothesize that there is eventually only one way to implement a certain functionality, which cannot be optimized any further.

4.4 Metric Vector

This test calculates a vector of certain metrics which are demonstrated by Kontogiannis to be a good indicator of similarity in abstract syntax trees [3].

The included metrics are the following:

S-Complexity $S = calls(U_x)^2$

Where calls(U_x) is the number of individual function calls in U_x (fan-out).

D-Complexity $D = \frac{|globals(U_x)|}{calls(U_x) + 1}$

Where $globals(U_x)$ is the number of global variables used or updated within U_x. A global variable for U_x is a variable which is used or updated in U_x but not declared in U_x.

McCabe Complexity $M = dp(U_x) + 1$

Where $dp(U_x)$ is the number of control decision predicates in U_x.

Kafura Metric

$$K = \Big(\big(fp(U_x) + vars(U_x) + callsTo(U_x)\big) * \big(calls(U_x) + globals(U_x) + refparams(U_x)\big)\Big)^2$$

Where $fp(U_x)$ is the number of formal parameters in U_x, $vars(U_x)$ the number of variables used in U_x, $callsTo(U_x)$ the number of function calls to U_x and $refparams(U_x)$ the number of reference (pointer type) parameters that are updated in U_x.

Kontogiannis bases this technique on the assumption that if two code units are clones, they will share a number of structural and data flow characteristics.
We expect that these characteristics as measured by these metrics are also applicable to semantic clones because we expect that functions that perform the same functionality are likely to be implemented in the same way, which would show in the number of calls made, the number of globals used, etc.

4.5 Dictionary Lookup

This method can be regarded as an extension to the exact token match test. When a word cannot be directly matched, it is looked up in Wiktionary[1], which will return a list of words that appear inside a dictionary entry for this keyword. The offline dump file[2] of the English version of Wiktionary is used to look up keywords. Matches between words looked up in the dictionary and the other code unit are counted. Figure 10 gives an example of a dictionary lookup test.

[1]http://en.wiktionary.org
[2]http://download.wikipedia.org/enwiktionary/latest/enwiktionary-latest-pages-articles.xml.bz2

This test is an improvement of the exact token match test, and should therefore at least have the same test characteristics as the exact token match test, but expected is that the test is a substantial improvement.

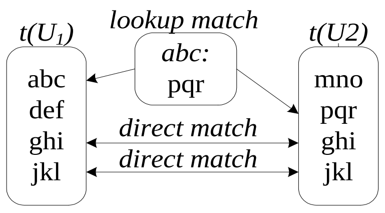


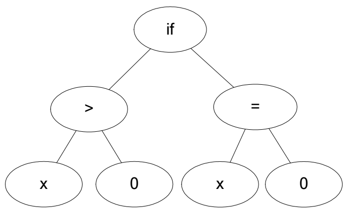
Figure 10. Example of a dictionary lookup test, with 2 direct matches and 1 looked-up match.

4.6 AST comparison

This tree-based technique has Abstract Syntax Trees (ASTs) as input and compares them by building a characteristic vector of this tree, as described by L. Jiang in [12]. Each number in this vector is the count of the occurrence of a specific pattern (e.g. while, if, expression, call) inside this tree. Code clones are detected by calculating the Euclidian distance between two vectors. See **Definition 5.7 (Euclidian distance)**. For an explanation of this technique, see figure 11.

In this example, only a small piece of code is shown. The characteristic vector contains all possible nodes that can occur inside an AST. In this example, only a small selection of node types is shown. Node types that do not occur in the tree (like a while-statement) are included with a count of 0.

```
if (x > 0)
    x = 0;
```



```
<if_stmt, while_stmt, bool_expr, asmgnt, gt> =
      <1, 0, 1, 1, 1>
```

Figure 11. Example of a characteristic vector of a piece of code. Normally, all node types (even with a count of 0) are included in the vector.

Code similar in functionality is expected to have structural characteristics in common. This technique can also be considered a metric-based technique which creates a fingerprint of subtrees. A strong point of this technique is that subtrees of code can be combined into a new vector easily by adding two vectors, and thus clones in a complete or partial tree can be easily detected. Another advantage is that calculation of these vectors is fast and straightforward. Even if no AST is available, pattern counting could be performed by regular expression matching. A disadvantage of this approach is that vectors of subtrees cannot be easily calculated.

5. Definitions

The following definitions are used in this study. For more details see the explanation for each test in section 4 (**Techniques**).

Definition 5.1 (Code unit) A *code unit* U_x is the input of a test T. In this study, it can be the text of a complete source file or function, represented as a string.

Definition 5.2 (Test) In this study, a *test* is a function $T : U_1 \times U_2 \rightarrow \mathbf{R}$. In addition, the function satisfies the following properties, which makes it a metric [5]:

- $T(U_1,U_2) = T(U_2,U_1)$
- $T(U_1,U_2) \geq 0$
- $T(U_1,U_2) = 0$ if and only if $U_1 = U_2$
- $T(U_1,U_3) \leq T(U_1, U_2) + T(U_2, U_3)$

Definition 5.3 (Code clone) Two code units U_1 and U_2 are a *code clone* according to a test T, if $T(U_1, U_2) \leq \alpha$, with α being a threshold value.

Definition 5.4 (Syntactic clone) Two code units U_1 and U_2 are a *syntactic clone* if $d_E(U_1, U_2) = 0$ (they have identical character arrays). See **Definition 5.6 (Levenshtein edit distance)**.

Definition 5.5 (Semantic clone) Two code units are a *semantic clone* if they fulfill the same requirement.

Definition 5.6 (Levenshtein edit distance) The *Levenshtein edit distance* [15] of two code units U_1 and U_2, denoted as $d_E(U_1, U_2)$ is the minimal sequence of edit operations (substitute, insert or delete) that transforms U_1 into U_2. Equal weight of each operation is assumed.

d_E(*abcde*, *bcdef*) = 2. *abcde* becomes *bcde* via deletion of *a*. *bcde* then becomes *bcdef* through insertion of *f*.

Figure 12. Example of the Levenshtein edit distance calculation.

Definition 5.7 (Euclidean distance) The *Euclidian distance* of two vectors υ_1 and υ_2 is

$$d(\upsilon_1, \upsilon_2) = \sqrt{\sum_{i=1}^{n} (\upsilon_{1i} - \upsilon_{2i})^2}$$

Where *n* is the length of both vectors. Equal vector lengths are assumed.

Definition 5.8 (Kolmogorov Complexity) The *Algorithmic Kolmogorov Complexity* of a code unit U_x is defined as the length of the shortest program that computes or outputs U_x, where the program is run on some fixed reference universal computer. This complexity can be approached by the length of the shortest description of U_x [11].

Definition 5.9 (Normalized Compression Distance) The *Normalized Compression Distance* $NCD(U_1, U_2)$ is defined as follows [16, 17]:

$$NCD(U_1,U_2) = \frac{C(U_1U_2) - \min\{C(U_1), C(U_2)\}}{\max\{C(U_1), C(U_2)\}}$$

Where $C(U_1U_2)$ is the size of compressed file containing the concatenated text of U_1 and U_2.

Definition 5.10 (Normalized Edit Distance) The *Normalized Edit Distance* [25] between two code units U_1 and U_2 is defined as follows:

$$NED(U_1,U_2) = \frac{2d_E(U_1,U_2)}{\propto (|U_1| + |U_2|) + d_E(U_1,U_2)}$$

Where $|U_1|$ is the number of characters in U_1. Since equal weight of substitution, insertion and deletion operations is assumed, $\propto = 1$.

Definition 5.11 (Exact Token Match Distance) The *exact token match distance* between two code units U_1 and U_2 is defined as follows:

$$ETMD(U_1,U_2) = \frac{\max\{|t(U_1)|,|t(U_2)|\} - |\cap(t(U_1),t(U_2))|}{\max\{|t(U_1)|,|t(U_2)|\}}$$

where $t(U_x)$ is the collection of tokens in U_x. A token is every substring of U_x that is separated by a non-ASCII character, underscore character or a capital. The selection can be filtered by the length of each token, shown in the following example.

$U_x \leftarrow$ "calculateAverage(int first_value, int second_value, int* r)"
$t_3(U_x)$ = ("calculate", "average", "int", "first", "value", "second")

Figure 13. Example of tokens in a code unit, selecting only unique tokens that have a length greater than or equal to 3.

Definition 5.12 (Dictionary look-up distance) The *dictionary lookup distance* of two code units U_1 and U_2 is the same as the *exact token match distance*, but when a token cannot be directly matched, it is looked up in a dictionary, and is matched against the returned collection of words in the dictionary entry.

Definition 5.13 (Abstract Syntax Tree) An *abstract syntax tree* is an abstract representation of source code, as generated by a parser. It is a more compact representation of code in which details unnecessary for the compilation process are omitted. See Figure 11 for an example.

6. Open source programs

The following open source programs were used in our analysis:

kcalc 4.4.1[3] **and gcalctool 5.29.92**[4] are two C/C++ desktop calculators from the KDE desktop and from GNU (Linux). Because of small size and limited domain, a lot of duplicate functionality is expected.
bash 4.1[5] **and tcsh 6.17**[6] are two shells that are included in most Linux distributions. Both shells should perform the same functionality on a global level, although implementation details are expected to differ.
joe 3.7[7] **and nano 2.2.1**[8] are two relatively small text editors for Linux that are also expected to contain a lot of cloned functionality.

Application	# files	# functions	# lines
kcalc	8	396	7202
gcalctool	15	375	14895
bash	228	2588	119534
tcsh	74	1232	60857
joe	50	1077	38364
nano	15	407	22542

Figure 14. Characteristics of included test projects

7. Implementation

To collect test results, a calculation tool was created which was implemented using C#. Source code of all code units was manually collected and entered in a Microsoft SQL Server database table.

```
for each code unit u1
   for each code unit u2
      su1 = fetch source text of u1 from db
      su2 = fetch source text of u2 from db

      for each test t
         score = t(su1, su2)

         write tuple to table:
            <test id, u1 id, u2 id, score>
      end for
   end for
end for
```

Figure 15. Pseudo code for the calculation of test results

Source code was stored as plain text in a table with an identifier that uniquely identifies a single code unit. All tests and collected code units as described in **Results** were iterated and served as input to each test. See Figure 15.

Eventually, tuples of the form `<test_id, unit1_id, unit2_id, score>` were stored in the database. Since scores have been normalized, scores range from 0 to 1.
To calculate and display each ROC curve, all tuples belonging to a specific test were selected. Inside a loop the threshold value was incremented with 1% of total range (0 to 1) and sensitivity and specificity values were calculated for this threshold value. This produces sensitivity-specificity pairs (sensitivity = TP / (TP + FN)), specificity = TN / (TN + FP)) for approximately 100 threshold values. These pairs were plotted in ROC diagrams.
To calculate accuracy/threshold diagrams, the same loop was used to iterate over threshold values. For each threshold value inside this loop, accuracy ((TP + TN) / total) was calculated and a diagram was plotted as well.

We had great trouble finding a suitable ANSI C parser. Since C is not a standardized language, several non-standard extensions exists which are often specific to a certain compiler. No single parser seemed able to cope with all code given to it. The programs we tried include ANTLR, Elkhound/Elsa, GCC and GCCXML, among others. There were always some small features inside some code units which caused the parsing program to crash. GCC was able to parse each file in its entirety, but adapting GCC itself was considered to be too big a task and to be outside the scope of this study.
Finally, we found srcML[9], which was able to parse every piece of code we gave to it without trouble. SrcML was capable of parsing source code and produce an abstract syntax tree in the form of an XML document. From this file an in-memory abstract syntax tree could be built which could be compared to other abstract syntax trees.

Several other external tools were used to help calculate scores. To calculate scores for the Kolmogorov LZMA test, 7-Zip[10] was used, an open source program that contains a couple of different compression algorithms. In our case, we used LZMA compression. We started the program with the arguments "`a -t7z -m0=LZMA -mx=9 <filename>`", using the maximum amount of LZMA compression available.
To help extract function headers and bodies, ctags[11] was used, a program that creates an index of language objects found in source files. It was started with the arguments "`-x -s -e <filename>`", which made sure static and external headers were also included.
To extract object code for single functions out of object files as generated by a compiler, objdump[12] was used. From the output of objdump hexadecimal start address and length of each function could be extracted, which could then be used to extract binary code out of object files.

Software quality We are confident about the quality of developed software and reliability of our test results. We

[3] `http://utils.kde.org/projects/kcalc`
[4] `http://calctool.sourceforge.net`
[5] `http://www.gnu.org/software/bash`
[6] `http://www.tcsh.org`
[7] `http://joe-editor.sourceforge.net`
[8] `http://www.nano-editor.org`

[9] `http://www.sdml.info/projects/srcml`
[10] `http://www.7-zip.org`
[11] `http://ctags.sourceforge.net`
[12] `http://linux.die.net/man/1/objdump`

can not fully exclude the possibility that there are bugs in the software that cause incorrect results, but software was tested intensively and first results where checked manually, which should reduce the chance of bugs that influence test results in a major way.

Performance We did not take into account performance considerations when building our tool. This had the consequence that a single test could take minutes to finish, and executing all input combinations for a single test could take more than an hour. We did not have the goal to build a tool that would be scalable or could be used outside a research environment, but instead we wanted to obtain test results in the easiest way without spending too much time developing a tool.

8. Results

In this section, results of manual detection as originally performed by a single observers will be shown first. Second, results from multiple observers will be shown. Finally, test characteristics of automated tests are shown, which were calculated by combining manual test results with automated test scores.

8.1 Manual collection, single observer

File	
▪	gtk.c
▪	kcalcdisplay.cpp
Method	
A.1	static void display_refresh(GCDisplay *display)
B.12	bool KCalcDisplay::updateDisplay(void)
A.21	void ui_set_accuracy(int accuracy)
B.17	void KCalcDisplay::setPrecision(int precision)
A.23	static void mp_add2(const MPNumber *x, const MPNumber *y, int y_sign, MPNumber *z)
B.21	static KNumber ExecAdd(const KNumber& left_op, const KNumber & right_op)
A.18	static void solve(const char *equation)
B.7	bool CalcEngine::evalStack(void)
A.14	static void do_paste(GCDisplay *display, int cursor_start, int cursor_end, const char *text)
B.1	void KCalcDisplay::slotPaste(bool bClipboard)
A.3	void display_pop(GCDisplay *display)
B.5	void KCalcDisplay::slotHistoryBack()
A.13	void display_push(GCDisplay *display)
B.15	void KCalcDisplay::slotHistoryForward()
A.6	void mp_and(const MPNumber *x, const MPNumber *y, MPNumber *z)
B.21	static KNumber ExecAnd(const KNumber & left_op, const KNumber & right_op)

Figure 16. Semantic clones found manually in gcalctool (above) and kcalc (below).

The manual detection process resulted in 12 file-level semantic clones and 18 function-level semantic clones in three training projects. After manual detection, 11 random file-level combinations and 19 random function-level combinations (which were definitely not semantic clones) were added to introduce noise in the detection process. Without this the number of false positives and true negatives would always be 0. In total, 78 code units where selected from three different training projects, of which 54 code units were functions and 24 were files. An overview of manually detected clones can be found in figures 16, 17 and 18. See section **Discussion** why so few code units were found to be semantic clones.

File	
▪	execute_cmd.c
▪	sh.exec.c
▪	expr.c
▪	sh.exp.c
▪	error.c
▪	sh.err.c
▪	bashhist.c
▪	sh.hist.c
▪	stringlib.c
▪	tc.str.c
▪	variables.c
▪	ma.setp.c
▪	jobs.c
▪	sh.proc.c
Method	
A.4	void save_history()
B.2	void savehist(struct wordent *sp, int mflg)
A.15	void load_history()
B.9	void loadhist(Char *fname, int mflg)
A.19	int execute_command_internal(command, asynchronous, pipe_in, pipe_out, fds_to_close)
B.14	void doexec(struct command *t, int do_glob)
A.20	void sys_error(const char *format, ...)
B.23	void stderror(unsigned int id, ...)
A.11	intmax_t evalexp(expr, validp) char *expr; int *validp;
B.24	int expr(Char ***vp)

Figure 17. Semantic clones found manually in bash (above) and tcsh (below).

File	
▪	charmap.c
▪	chars.c
▪	help.c
▪	help.c
▪	utils.c
▪	utils.c
▪	pw.c
▪	prompt.c
Method	
A.8	void help_display(Screen *t)
B.4	void do_help(void(*refresh_func)(void))
A.5	void help_init(void)
B.6	void usage(void)

Figure 18. Semantic clones found manually in joe (above) and nano (below).

8.2 Manual collection, multiple observers

In figure 19, answers from the survey can be found. First, our own answers are displayed. These are the clones we originally used as gold standard. Numbers inside the table refer to functions in figures 16, 17 and 18, referred requirements can be found in figure 5. In the last three columns, answers from other observers are shown. The number of times a function is selected is displayed (3 x 11 means 3 observers chose function 11 for this requirement). In the last column, the number of times A and B match exactly as a pair with our own answer is counted, and displayed with the proportion of people agreeing (2 (24%) means that 2 users chose both A and B the same as we did, and this is 24% of total answers for this requirement).

Requirement	**Our observation**		**All observers**		
	A	*B*	*A*	*B*	*A & B*
Calculators					
DC1	11	-	6 x 11	4 x 7	2 (29%)
			1 x 18	1 x 21	
DC2	1	12	2 x 1	4 x 12	0 (0%)
			2 x 13		
			2 x 18		
DC3	6	21	7 x 6	7 x 21	7 (100%)
DC4	-	-	1 x 22		6 (86%)
DC5	16	-	4 x 16	1 x 21	4 (57%)
			1 x 6		
DC6	-	-			7 (100%)
DC7	3	5	5 x 3	4 x 5	3 (43%)
DC8	13	15	1 x 4	1 x 2	0 (0%)
			1 x 16	1 x 9	
				4 x 15	
DC9	21	17	7 x 21	5 x 17	5 (71%)
DC10	-	-	5 x 14	6 x 1	5 (71%)
DC11	17	-	7 x 17	2 x 12	4 (57%)
				1 x 1	
Shells					
SH1	19	14	4 x 19	5 x 14	3 (43%)
			1 x 11	1 x 3	
SH2	20	23	7 x 20	7 x 23	7 (100%)
SH3	-	-			7 (100%)
SH4	4	2	1 x 15	6 x 2	6 (86%)
			6 x 4		
Text editors					
TE1	8	4	7 x 8	6 x 4	6 (86%)
TE2	5	-	5 x 5	4 x 3	1 (14%)
				1 x 4	
				1 x 8	
TE3	-	-		1 x 10	6 (86%)

Figure 19. Our original observation and observations of other observers. The references ("DC1") in the first column refer to the requirements in figure 5. The numbers in the other columns refer to the functions in figure 16, 17 and 18.

Below, test characteristics of different tests are presented. All functions from the three different projects have been combined into a single set. This should be no problem since files and functions are independent units, and test outcomes are calculated for each file or function individually.

8.3 Automated test results

8.3.1 Edit distance

Results for the edit distance test are shown below. As it turns out, this test does not perform better than flipping a coin (visible as the diagonal line going to the points (0, 0) and (1, 1) in figure 17). For each test, the Delong Delong Clarke-Pearson method [28] is used to compare curves.

In figure 21, "Area" is the area under the curve. This area has to be significantly bigger than 0.5 to make this test a good predictor for semantic clones. The farther the most upper-left point of the curve comes to the point (0, 1), the better the test is at predicting semantic clones.

With a p value of 0.30, this test does not perform significantly better than chance, based on a 95% confidence interval. When the number 0.5 is not included in the range of the 95% confidence interval (in this case 0.36 to 0.74) the test is significant. Edit distance as performed on tokens instead of separate characters returned similar results.

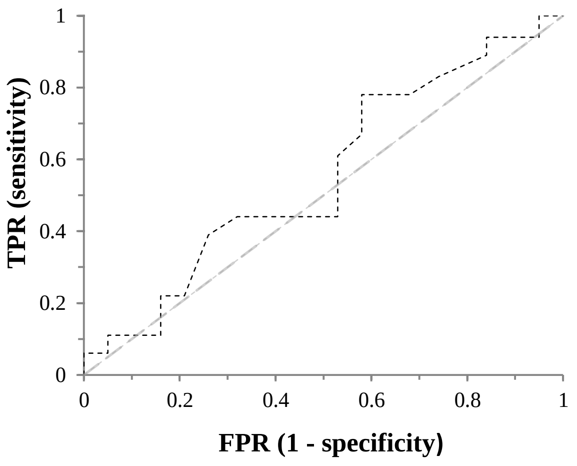


Figure 20. ROC curve of edit distance test. TPR = True Positive Rate = sensitivity = recall = TP / (TP + FN), FPR = False Positive Rate = FP / (TN + FP).

Area	0.55
p	0.30
95% C.I.	0.36 - 0.74

Figure 21. Test characteristics.

8.3.2 Exact Token Match

With a p value < 0.0001, this test performs significantly better than chance.

Excluding keywords decreases test performance, as can be seen from the accuracy/threshold graph below. Without keywords, maximum accuracy is approximately 70%, while with keywords, maximum accuracy is 78.4%.

The figure below shows that this optimal accuracy (with keywords) is reached at threshold values of approximately 0.94 and 0.96.

	With keywords	*Without keywords*
Area	0.83	0.76
P	**0.00**	**0.00**
95% C.I.	0.69 - 0.97	0.60 – 0.91

Figure 22. Test characteristics

The numbers inside the binary classification matrix below depend on the threshold value chosen, see **definition 4.x (Code clone)**. The lower the score for a test, the more likely it is the two input units are a code clone, according to that test. The threshold value is the cut-off point of the test. Below this value, test outcome is considered to be positive (a clone is detected).

Confusion matrix (functional level)		*Manual detection*	
		yes	*No*
Test outcome	*yes*	*15*	*5*
	no	*3*	*14*

Figure 23. Confusion matrix of the exact token match test at a threshold value of 0.94.

In figure 25, numbers for a threshold value of 0.94 are shown.

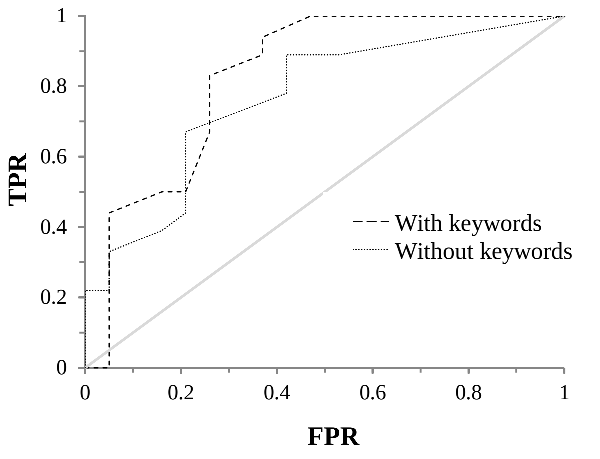


Figure 24. ROC curve of exact token match test with and without keyword matching.

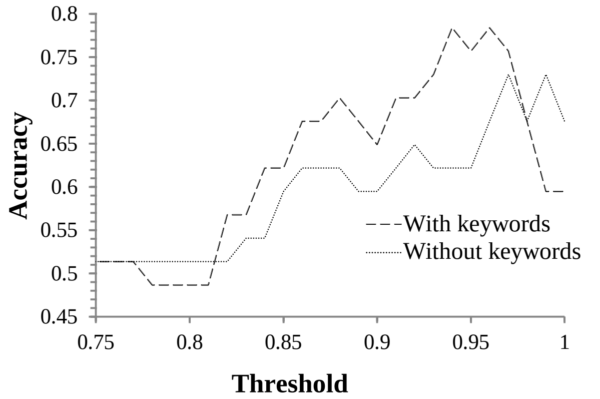


Figure 25. Accuracy/threshold curve

Test characteristics calculated from this confusion matrix are shown in figure 26.

Sensitivity (recall)	83%
Specificity	73.7%
Accuracy	78.4%
PPV (precision)	75%

Figure 26. Test characteristics at a threshold value of 0.94. PPV = Positive Predictive Value.

8.3.3 Kolmogorov LZMA

With a p value of 0.79, this test does not perform significantly better than chance. Performing compression on object code gives similar results, but is not shown in the graph below.

Area	0.42
P	0.79
95% C.I.	0.23 - 0.61

Figure 27. Test characteristics

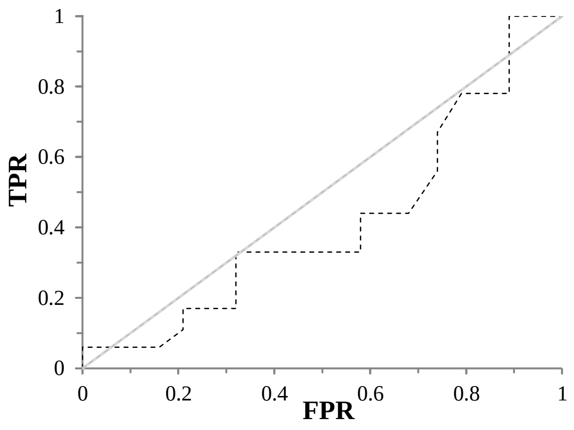


Figure 28. ROC curve of Kolmogorov LZMA

8.3.4 Metric Vector

With a p value of 0.38 this test does not perform significantly better than chance. We originally added all metrics as described by Kontogiannis, but other combinations of metrics with different weighing factors gave similar results.

Area	0.53
p	0.38
95% C.I.	0.34 – 0.72

Figure 29. Test characteristics

8.3.5 Dictionary Lookup

With a p value of 0.00, this test is a significant predictor for semantic clones. The area shows that this test does not perform better than the exact token match test (dotted line), but worse. See section **Discussion** for more details.

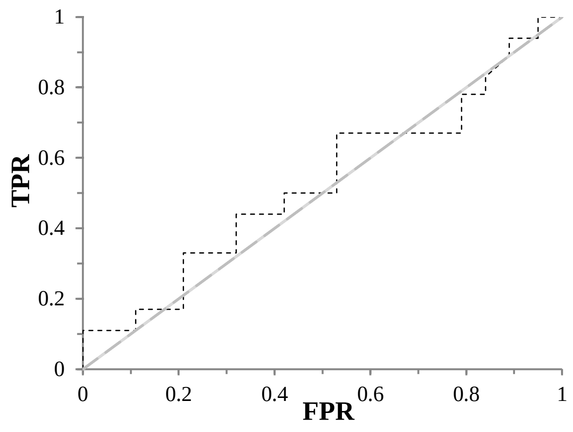


Figure 30. ROC curve of metric vector

Area	0.74
P	**0.00**
95% C.I.	0.57 - 0.90

Figure 31. Test characteristics

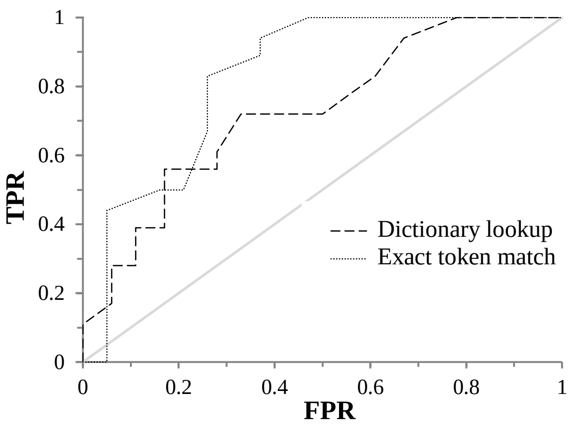


Figure 32. ROC curve of dictionary lookup test

8.3.6 AST Comparison

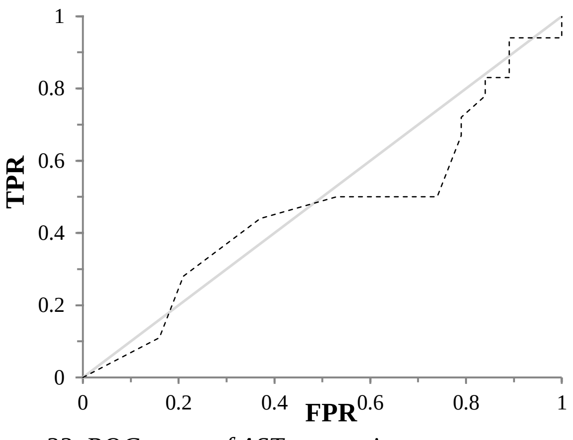


Figure 33. ROC curve of AST comparison test

With a p value of 0.53, this test does not perform significantly better than chance.

Area	0.49
P	0.53
95% C.I.	0.29 - 0.69

Figure 34. Test characteristics

9. Discussion

The only test that returns significant results is the exact token match test. This test has a sensitivity of 83%, a specificity of 73.7% and a positive predictive value of 75%, given our test cases. This means that it correctly identifies 83% of all semantic clones in our test sample, and missed 17% of known semantic clones in our sample. For all non-semantic clones in our sample, the chance that this test recognizes them correctly as not being clones is 73.7%, which means that 26.3% of all non-semantic clone pairs were incorrectly identified as clones. The positive predictive value of this test states that of all clones as positively identified by this test, 75% of cases are indeed semantic clones. Extending this test by a dictionary match decreased test performance.

These numbers are somewhat disappointing since we expected the exact token match to return higher scores, and we expected that adding a dictionary lookup would increase test performance, instead of decreasing it. We are also surprised to find out that other tests are not capable of detecting semantic clones, given the fact that we limited our selection of test cases to rather trivial examples, and left out more complex ones.

Low scores We can think of a number of possible reasons for these low scores. As shown in our survey, it is extremely difficult even for human observers to correctly classify two pieces of code without knowledge from the design of the complete system. This contextual knowledge is completely missing in our automated tests. One could argue that some kind of structural knowledge is available after parsing each function and building an AST, but given results of our AST comparison test this is not enough. Besides this, precision and recall scores such as ours are not uncommon in classic information retrieval experiments (for instance, in [3] and [34]).
All forms of edit distance tests (on characters and trees) do not return significant results. This may be explained by the fact that two semantic clones can fulfill the same requirement through two completely different implementations, which results in two functions that look completely different in every aspect. The metric vector test did not perform as well as expected. No individual metric performed significantly well and combining several metrics into a single vector did not improve test performance. Chosen metrics apparently tell nothing about semantic similarity, at least not in our collection of test cases.

Manual detection process Semantic clones were originally detected by a single observer. This raised questions about the external validity and reliability of our gold standard.
We tried to address this issue by asking multiple independent observers to judge the same code samples. A questionnaire was prepared which should be completed in no longer than half an hour. This turned out to be harder than expected, because the total number of semantic clones as selected by us was simply too large to put in a questionnaire all at the same time. This forced us to make a selection of clones, which raised the question how objective our selection was. We tried as hard as possible to prevent selection bias in our questionnaire.

As it turns out, multiple independent observers are more able to identify semantic clones within a short period of time, although large differences in agreement between different requirements exist. Most observers are able to correctly identify the most trivial cases, which we believe means that people generally have an intuitive notion of a semantic clone and are capable of detecting them given a certain set of functions. In cases where there exists disagreement we are capable of exactly pointing out why we believe other observers judged wrongly. This is mostly caused by a lack of contextual information and because some unrelated functions in our survey showed so much resemblance that they were accidentally confused.
For instance, someone responded that according to him the requirement “after every calculation, the result is displayed on screen” belongs to the function `static void solve(const char *equation)`. Although understandable, this function does not take care of something *after* every calculation, but *is* the calculation itself. Besides that, in this function no result is displayed but the result is only calculated.
In another example, implementations of the requirement “a help screen can be shown” were sometimes confused with implementations of the requirement “after entering ‘--help’ on the command line, usage is printed”. Function headers and requirements look alike on the surface but goals are nevertheless completely different.

We could have taken an approach in which we completely replaced our own observation with the one from the survey. A binary classification setup ensures that current manual results can safely be replaced by other observations while leaving automated test results unaffected. In this sense, there is no need to defend our own observation since test characteristics could also have been calculated from any observation, even if we disagreed with them. We decided not to take this approach because we believe that manual results from other observers are not necessarily an improvement over our own test results, as shown above.

Trivial requirements Supplying requirements with functions in our survey may raise questions why observers were not allowed to reconstruct these requirements from functions themselves, or why we did not use requirements as supplied by the original authors of the software. Starting with the latter, none of the open source projects we studied contained any documentation on requirements of any kind.
A setup in which users are asked to reconstruct requirements from given source code themselves without supplying them would also have been possible. A survey would then contain two lists of functions which observers should connect themselves. This reduces the risk of influencing observers through given requirements. However, tryouts with our survey showed that supplying only function bodies is not enough for observers to judge code samples. More information on these functions is needed. Another reason to add them is that without requirements and with only a list of two functions, each observer is free to maintain his own definition of a semantic clone, which makes comparison between results from different observers almost impossible.

We believe requirements as supplied by us are trivial, and that other observers would think of the same requirements as we did. Exactly how trivial most requirements are can be seen in figure 5. We believe that anyone would have thought of the requirement “a binary AND-operation can be performed on two numbers” given the corresponding function. For all other requirements, we believe this also applies.
Without given requirements, some observers would apply a definition of semantic clones that was too strict for our purposes. These respondents would restrict to a mathematical definition, in which only the most simple functions were marked as semantic clone candidates (e.g. adding and subtracting numbers). We originally planned to use the same mathematical definition of semantic clones, in which a semantic clone is defined as two functions causing the same change in external state and side effects (bi-simulation), but soon found out that this leads to a number of clones that is too small to be useful. With this definition, the number of clones found in all 6 projects would be limited to less than 5.

Trivial cases We decided not to add syntactic clones (copies of the same code, as in the multiple precision arithmetic library in kcalc and gcalctool) because this would make our tests appear better than they actually are. Adding syntactic clones increases sensitivity, specificity and positive predictive value for a test. This is due to the fact that for syntactic clones, every automated test will automatically return 0. For instance, edit distance between two identical pieces of code is always 0. The same is true for all other tests as described in this study. Since we defined two pieces of syntactically identical code to be semantic clones as well, adding these cases automatically leads to an increase in the number of true positives, which in turn increases sensitivity, specificity and positive predictive value for this test.

We only selected trivial cases for our gold standard. These cases are either obvious a semantic clone (on which most other observers agree) or obviously not a semantic clone

(two random unrelated functions). This selection of cases is a gross simplification as compared to real-world systems. In the end, a single definition of a semantic clone may not do justice to the complexity of real-life examples.
Note that adding cases which humans consider to be trivial does not influence whether or not cases possess characteristics from which a computer can draw a trivial conclusion. In other words, even if humans find a case trivial, a computer may still have a hard time finding out if they are semantic clones or not.
And even with a single definition ("fulfilling the same requirements"), there may be disagreement among different observers which function implements exactly which requirement. Types of disagreement can even be divided into three categories: (1) Observers disagree among each other, resulting in an average score which does not fit in a single category; (2) Observers agree but agree to be unsure; (3) Observers want to classify cases in categories other than "yes" or "no" (maybe into different types of semantic clones). Ways to solve this disagreement include merging uncertain cases in "yes" or "no" categories, adding new categories or limiting selection of cases to trivial ones. We chose the latter. For further discussion on an approach that takes a more complex definition of a semantic clone into account, see section **Future work**.

Original goal Our original goal was to get threshold values from training data and to apply them to a real-world production system. We performed a small experiment with optimal threshold values for the exact token match test on a real production system. We printed the top 10 functions with the smallest distance according to this test, but results did not include any significant matches. We therefore concluded to focus on optimization of techniques instead of trying to design a technique that performs well on real-world examples.

Limitations in test setup Using a binary classification matrix and calculating corresponding test characteristics from this matrix is a proven method for the evaluation of binary classification tests [31]. There is no doubt about the correctness of the calculation of test characteristics from a binary classification matrix, given a certain gold standard (in this case, multiple independent human observers). Internal validity of our study is ensured by this test setup.

The biggest problem of our test setup is the limited ability to extrapolate test results to untested cases (external validity). If tests perform well on a set of limited code units, there is no guarantee that these tests will perform equally well on untested cases. Another problem with our results is that it only contains projects from three different domains (desktop calculators, shells and text editors). One could argue that this should have been more.
To what extent we can generalize results to other domains, and whether selection of these domains influences test results is uncertain. There are, however, a number of reasons why it should be possible to extrapolate our test results to untested cases: (1) since we did not include any domain-specific knowledge in any of our tests, all tests could be considered domain-independent and we therefore have no reason to believe that chosen domains influence test results, (2) we did include 3 totally unrelated domains which is already a kind of random selection and (3) we included a wide range of different types of functions, which makes sure our conclusions do not apply only to functions of a certain length or type.
Another limitation of our test setup is that manual observers were only allowed to select semantic clones from a given set of functions. This could result in a selection bias in which "false negatives" (cases we overlooked but are indeed semantic clones) are missed. If we missed functions or we did not include them in our survey, they could not be checked by other observers. Ideally, all observers should look through the entire source code and report all clones found. This approach has the problem that it has taken us almost a week to complete. We tried to address this problem by asking the original developers of the software if they knew of any places in their code and their functional equivalent (e.g. kcalc and gcalctool) where there exist functional duplicates. This resulted in a library which was used in both programs, which we did not find interesting enough to include in our analysis. It remains difficult to prove how representative our sample of test cases is, but since we checked complete source of all projects we have confidence in our own results, and given our results, most independent observers agree with our findings.

The number of selected clones (12 files and 18 functions) and non-clones (11 files and 19 functions) may also pose a threat to validity. Increasing the number of files and functions will increase the validity and reliability of test results. With too little test results, graphs will not have enough detail to make sensible statements about test performance. Besides this, calculation of test significance is influenced by the number of input values (statistical tests almost never return significant results when the number of values is too low). This issue does not pose a threat because we have a test that performs significantly positive and the same number of test cases are used for all tests.

Imperfect gold standard Even after consulting multiple independent observers and the original authors of the software, our gold standard remains imperfect. In this sense, the term "gold standard" is misleading. A better name would have been "reference test". No matter how many cases we add, there are always additional cases we did not check. As long as automated techniques do not get close to sensitivity scores of 100%, and there are no other compelling reasons why other gold standard would have been better, we believe that ours is good enough for this study.
In medicine, imperfect gold standards are very common. The first test to detect an aortic dissection (a serious medical condition in which a tear in the wall of the aorta causes blood to flow between tissue layers), was an aortogram. This test has a sensitivity of 83% and a

specificity of 87%. This means that the “ideal” test of that moment missed 17% of all cases! There were simply no better alternatives at that time, and all other tests for the same condition were compared to this one.

We fully acknowledge the limitations of our study. This study determines the ability of tests to detect semantic clones, given a limited set of test cases. We do not have the intention to be totally complete in our selection of cases, nor do we aim to provide a definite reference in the field of semantic clones. The definition of a semantic clone as stated by us is by no means intended as final, and we welcome a discussion on this definition and on results of our study.

10. Conclusion

Our results show that, given our input set of code units, the only test capable of predicting semantic clones statistically significant (Sensitivity 75% with a 95% confidence interval) is the exact token match test. Adding a dictionary lookup to words which were not directly matched did not increase its performance. The exact token match test performs best when keywords are included. This means that every other test is not a significantly better predictor than pure chance.

Our initial judgment was mostly confirmed by the questionnaire we sent to independent observers. We can easily advocate cases on which there exists mutual disagreement between observers or disagreement between observers and our own judgment. Each individual case can be shown to be trivial, but may require more contextual knowledge of the underlying system.

11. Future work

Extended confusion matrix As explained in the previous section, a problem with our binary classification scheme is that in reality, only the most trivial cases can be included in our analysis which can be classified as either being definitely a semantic clone, or definitely *not* being a semantic clone.

Type of test granularity		Manual detection				
		Def. yes	*Prob. yes*	*Unc.*	*Prob. no*	*Def. no*
Test	*Cat. 1*					
	Cat. 2					
	Cat. 3					
	Cat. 4					
	Cat. 5					

Figure 35. A more complicated classification matrix. Def. = definitely, prob. = probably, unc. = uncertain.

Several examples of combinations of code units have been found which could not easily be classified by humans in these two categories. These examples would belong in categories like “probably yes”, “probably no” or “uncertain”. Due to the nature of our test setup (a 2 x 2 table), these categories were not included in our study. This choice is not without problems, as shown in the previous section.

To make it possible to also include uncertain cases, a more advanced test setup should be used. This is shown in figure 35. In this table, uncertainty categories are introduced. This would solve the problem of cases on which even humans do not disagree, or on which humans agree that the answer is uncertain. In case of disagreeing observers, an average answer could be selected (1 x definitely yes and 1 x definitely no = uncertain).

In this extended setup, each category has upper and lower limits. For instance, category 1 contains all scores which lie between 0 and 10, category 2 all cases which lie between 10 and 20, etc. This setup enables the inclusion of nontrivial cases in the analysis.
Another option is to include different types of semantic clones. For instance, categories like “not functionally related, “loosely functionally related”, “syntactic clone”, “same base functionality” and “semantic clone” could be used. These categories would appear in the columns of figure 35, instead of probability categories.

Adding more cases and observers Reliability of the gold standard could be further increased by adding more code clones. Now less than 100 code clones have been identified and were included in our study. The more this number is increased, the more reliable the gold standard becomes. All added cases should of course be judged by independent observers. Increasing the number of observers will also increase reliability.

Improving case selection in gold standard Given the nature of our test setup, tests were only executed on pre-selected cases. As shown in the discussion, this naturally raises questions about the external validity of our study.

In a future study, all observers should receive a complete copy of the source code. Each observer should be given the assignment to look through the entire source and identify semantic clones, without being restricted to a limited selection of clones as assembled by us. Requirements should not be supplied in advance. This approach has taken us a week to complete, but would nevertheless be the best way to create a gold standard. Each observer should look for cases in each category as described in the previous section (“probably” and “uncertain” also included). Then, calculations can be performed which combine data from different observers into a single gold standard (e.g. Kappa score).

In a future study, more open source projects from different domains should be added. Limiting the number of domains to 3 (desktop calculators, shells and text editors) makes it harder to extrapolate results to other domains. We stated before that we do not know which influence our selection of domains has on test results. What we are certain about is that increasing the number of domains will enable us to generalize the result over the domains we included additionally.

Combining multiple tests Multiple test results can be combined into a single score to improve test performance. We already tried to apply this on the metric vector test, in which a couple of different metrics can be combined in a couple of different ways. Each test can be multiplied with a weighing factor, to increase influence of one test over the others. Combining multiple tests only works when tests are "orthogonal" enough to each other to amplify each other. One test that uses a certain measure and another test that uses a derivative form of this measure look for essentially the same value and therefore cannot amplify each other. For example, if one test counts the number of lines and another tests the number of tokens, these values will be closely correlated to each other and combining these values will probably not improve test results. If, on the other hand, one test calculates the cyclomatic complexity and another test performs an exact token match, these values may be orthogonally enough to amplify each other. A future study could include calculating test characteristics for combined tests.

Adding a domain ontology test Checking only for exactly matching tokens has shown to be a significant predictor for semantic clones, but improvements are possible. A semantic web is a collection of words from a specific domain (such as desktop calculators) which are connected with links, which are also named. This leads to a web of connected terms. When two code units are semantically related, expected is that words mapped on this semantic web are more closely located to each other than with unrelated functions. See figure 36 for a small example of a semantic web of desktop calculators.

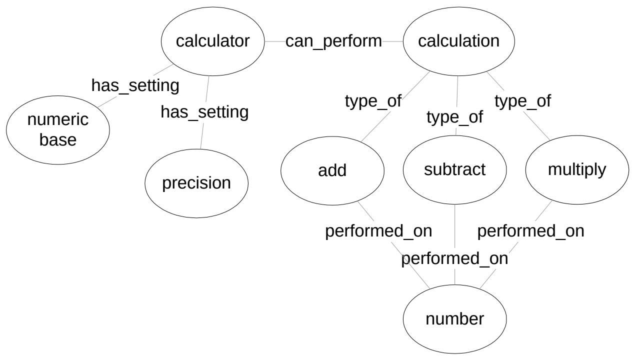


Figure 36. Example of a part of a semantic web of a desktop calculator.

12. Acknowledgements

We would like to thank Dik Prenger, Frans Kremer and Daniela Dupré-Vasaru at Philips Healthcare, Jurgen Vinju and Paul Griffioen at CWI for their valuable input. We would also like to thank all people who answered our semantic clone survey.

13. References

[1] J. Ferrante, K. J. Ottenstein and J. D. Warren. The program dependence graph and its use in optimization. In *TOPLAS*, 9(3), p. 319-349, 1987.

[2] C. K. Roy, J. R. Cordy and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. In *Science Of Computer Programming*, 74(7), p. 470-495, 2009.

[3] K. Kontoginannis. Evaluation Experiments on the Detection of Programming Patterns Using Software Metrics. In *WCRE '97: Proceedings of the Fourth Working Conference on Reverse Engineering*, p. 44, 1997.

[4] S. S. Skiena. *The algorithm design manual*. Springer-Verlag, New York, 1998.

[5] H. Machida. The Clone Space as a Metric Space. In *Acta Applicandae Mathematicae*, volume *52*: p. 297-304, 1998.

[6] M. Gabel, J. Lingxiao and S. Zhendong. Scalable detection of semantic clones. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, p. 321-330, 2008.

[7] F. Rysselberghe and S. Demeyer. Evaluating clone detection techniques from a refactoring perspective. In *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*, p. 336-339, 2004.

[8] M. Bruntink, A. van Deursen, T. Tourwé and R van Engelen. An Evaluation of Clone Detection Techniques for Identifying Crosscutting Concerns. In *ICSM: Procedings of the International Conference on Software Maintenance*, p. 200-209, 2004.

[9] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna and L. Bier. Clone detection using abstract syntax trees. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, p. 368, 1998.

[10] Walenstein, N. Jyoti, J. Li, Y. Yang and A. Lakhotia. Problems Creating Task-relevant Clone Detection Reference Data. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, p. 285, 2003.

[11] R. J. Solomonoff, A preliminary report on a general theory of inductive inference, 1960.

[12] L. Jiang, G. Misherghi, Z. Su and S. Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *Proceedings of ICSE, 2007*, p. 96-105.

[13] J. Krinke. Identifying similar code with program dependence graphs, 2001.

[14] T. Kamiya, S. Kusumoto and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. In *IEEE Transaction of Software Engineering*, 28(7), p. 654-670, 2002.

[15] I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. Soviet Physics Doklady, 10, p. 707-710, 1966.

[16] R. Cilibrasi and P. Vitányi. Clustering by compression. In *IEEE Transactions on Information Theory*, 51(4), p. 1523-1545, 2005.

[17] M. Cebrián and M. Alfonseca. *The normalized compression distance is resistant to noise, IEEE,* 2007.

[18] R. Geiger, B. Fluri, H.C. Gall and M. Pingzer. Relation of code clones and change couplings. In *FASE'06: Proceedings of the 9th International Conference of Fundamental Approaches to Software Engineering*, p 411-425, 2006.

[19] S. Giesecke. Generic modelling of code clones. In *Proceedings of Duplication, Redundancy and Similarity in Software,* 2006.

[20] D. Gitchell and N. Tran. Sim: a utility for detecting similarity in computer programs. In *ACM SIGCSE Bulletin*, 31(1), p. 266-270, 1999.

[21] B. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *WCRE'95*: *Proceedings of the Second Working Conference on Reverse Engineering*, p. 86-95, 1995.

[22] J. Mayrand, C. Leblanc and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *ICSM'96: Proceedings of the 12th International Conference on Software Maintenance,* p. 244-253, 1996.

[23] S. Ducasse, M. Rieger and S. Demeyer. A language independent approach for detecting duplicated code. In *ICSM '99: Proceedings of the International Conference on Software Maintenance*, p. 109-118, 1999.

[24] A. Marzal and E. Vidal. Computation of Normalized Edit Distance and Applications. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(9), p. 926-932, 1993.

[25] L. Yujian and L. Bo. A Normalized Levenshtein Distance Metric. In *IEEE Transactions of Pattern Analysis and Machine Intelligence*, 29(6), p. 1091-1095, 2007.

[26] T. Sager, A. Bernstein, M. Pinzger and C. Keifer. Detecting similar Java classes using tree algorithms. In *Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR'06)*, pp. 65-71, 2006.

[27] Y. Wuu. Identifying syntactic differences between two programs. In *Software Practice and Experience,* 21(7), 1991.

[28] Comparing the Areas Under Two or More Correlated Receiver Operating Characteristic Curves: A Nonparametric Approach. E.R. DeLong, D.M. DeLong and D.L. Clarke-Pearson. In *Biometrics* 44, p. 837-45, 1988.

[29] Z. Li and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI*, 2004.

[30] T. Fawcett. An introduction to ROC analysis. In *Pattern Recognition Letters* 27, p. 861–874, 2006.

[31] D.G. Altman and J.M. Bland. Diagnostic tests. 1: Sensitivity and specificity". In *British Medical Journal (BMJ)* 308 (6943), p. 1552, 1994.

[32] H. Basit, S. Pugliesi, W. Smyth, A. Turpin and S. Jarzabek. Efficient token based clone detection with flexible tokenization. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'07),* p. 513-515, 2007.

[33] John Johnson. Substring matching for clone detection and change tracking. In *Proceedings of the 10th International Conference on Software Maintenance*, p. 120-126, Canada, 1994.

[34] M. de Marneffe, C. D. Manning, C. Potts. "Was it good? It was provocative." Learning the meaning of scalar adjectives. Stanford University, Linguisics Department, 2010.