Annales Univ. Sci. Budapest., Sect. Comp. 57 (2024) 143–165

GRAPH-BASED DUPLICATED CODE DETECTION WITH RefactorErl

István Bozó, Zsófia Erdei and Melinda Tóth
(Budapest, Hungary)

Communicated by Zoltán Horváth
(Received January 9, 2024; accepted June 16, 2024)

Abstract. Code duplicates are created for various reasons such as code reuse by copying existing fragments of code (copy-and-paste programming). Considering the huge amount of duplicated code and its maintenance cost in large software systems, it is crucial to detect code clones.
In this paper we give a graph-based algorithm which uses the semantic program graph generated by the tool RefactorErl, a source code comprehension and refactoring tool for the programming language Erlang, to find different types of code clones in the source code. The presented algorithm was able to efficiently detect not only textually identical code fragments (Type I) but also copied and slightly modified code fragments, such as changed, added or removed expressions, in addition to variations in identifiers, literals, types, whitespace characters, layout and comments (Type II, Type III).

1. Introduction

The term code duplicate stands for snippets of codes that have identical or similar code fragments to it in the source code. Two code fragments may be

Key words and phrases: Static analysis, Erlang, duplicated code detection.
2010 Mathematics Subject Classification: 68W99, 68N18.
Project no. TKP2021-NVA-29 has been implemented with the support provided by the Ministry of Culture and Innovation of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme.
This work is a detailed version of a MaCS 2020 presentation.

duplicates of each other without being identical character by character. Even code sequences that are semantically identical, but not syntactically, could be considered code duplicates. Previous studies show, that a significant fraction (between 7% and 23%) of the code in a typical large software system consists of code clones [12]. Code duplicates are created for various reasons such as code reuse by copying existing fragments of code (copy-and-paste programming). While such cloning is often intentional and can be useful in many ways, generally in the long term the presence of duplicate code makes software maintenance more difficult and as a consequence more expensive [7]. For example by reusing existing functions with slight changes in variables or data structures, not only do we increase the chance of bug occurrences, but also if an instance of duplicate code is changed, its clones have to be modified as well. Duplicated fragments can also significantly increase the work to be done when adding new features to our software, and make code quality analysis and code comprehension more tedious as well. In addition, many other software development tasks such as understanding source code, analyzing code quality (fewer clones can mean better quality code), checking for plagiarism, investigating copyright infringement, analyzing software evolution, and debugging may require finding syntactically or semantically similar code snippets. Thus, duplicate code recognition is an important and valuable part of software analysis.

There exist several tools using several different approaches and methods that can be used to identify code duplicates within our source code. Such tools are called duplicated code detectors. The majority of these tools focuses on a concrete language, but there are also some attempts of creating general tools for finding code duplicates in any type of source code. These tools most often use pattern matching on the raw source code, mostly using a sliding window algorithm or analyzing a sequence of tokens generated from the source code. These tools can identify completely identical snippets of codes, but they are not able to reliably used for identification of code clones that are slightly different. The more sophisticated tools exist mostly for mainstream languages, the more successful methods analyze the syntax tree built up using the tokens combined with different metrics of the code.

To find Type II and Type III duplicates, we not only need to abstract away identifier names and literals but also have to account for inserted, deleted or modified lines in the code clones. The graph representation that RefactorErl provides contains even more information about the source beyond that of the abstract syntax tree. Our algorithm exploits these advantages to find similar subgraphs in the semantic program graph, grouping possible code clone candidates together and filtering the code fragments for better results.

The rest of the paper is structured as follows. Section 2 introduces code clones and Section 3 presents a categorization of clone detection algorithms and the corresponding related works. In Section 4 we briefly introduce our

target language, Erlang, and the framework we used in the analysis, RefactorErl. Section 5 describes our clone detection algorithm and Section 6 evaluates the algorithm on multiple open source modules and demonstrates the resulted clones we found. Finally, Section 7 concludes the paper and presents possible future directions.

2. Background

According to the code clone survey written by Chanchal K. Roy, James R. Cordy, and Rainer Koschke similarity of code fragments can be defined on different levels [12]. Two code fragments can be clones of each other based on the similarity of their program text or based on functionalities without being textually identical. In this paper we use the following classification for code clone types based on the kind of similarity they have:

Type I: Identical code fragments in terms of textual representation without whitespace characters and comments. We can see, on the examples shown below (1 and 2) the two code snippets are in terms of textual representation identical not considering the whitespace characters and comments.

Listing 1. Types of code duplicates

```
foo(A) -> A + 2. % comment
```

Listing 2. Types of code duplicates

```
foo(A) ->
    A + 2.
```

These types of duplicates are the easiest to detect even when using text-based methods.

Type II: Structurally or syntactically identical fragments, except identifiers, literals, types, whitespace characters and comments. In these types of clones statements, control structures, and the structure of expressions are the same. In the following example we can see such code clones.

```
foo(A) -> A + 2. % comment

foo(B) ->
    B + 3.
```

Type III: Code fragments are further modified: statements, expressions can be changed, added or removed. These types of duplicates are usually created

by modifying the copied code snippet. After the modification, the two snippets of code do not necessarily match syntactically or semantically, but we can say they are similar. Type III duplicates are harder to detect with token or metric-based methods.

```
foo(A) ->
    B = A + 2, % comment
    C = B - 1.

foo(A) ->
    B = A + 2, % comment
    C = bar(B),
    D = C - 1.
```

Type IV: Functionally equivalent code fragments that have the same pre- and post-conditions are called semantic clones. These code fragments do the same computations but might have different syntactic structures. The detection of Type IV clones is the hardest even after having a great deal of background knowledge about the program construction and software design.

```
sum(A) -> sum_impl(A, 0).
sum_impl([], V) -> V;
sum_impl([H|T], V) -> sum_implt(T, H + V).

sum(A) ->
    lists:foldl(fun(E,A) -> E + A end, 0, A).
```

3. Related works

Considering the huge amount of duplicated code and its maintenance cost of large software systems, it is crucial to detect code clones. Fortunately, there are multiple research studies to find clones. Once said clones are identified, they can be removed through source code refactoring.

There are many different tools for aiding code clone detection in the source code, but only a few of them were developed for commercial use. The more sophisticated tools heavily depend on the features of the analyzed programming language and able to detect different types of clones based on the used detection techniques.

There are several ways to find duplicated code. In the next section, we will introduce some possible approaches. Most tools used in practice do not rely on one, but several of these methods.

Text based methods. Several clone detection techniques are based on pure text-based methods. In this approach, the target source program is considered a sequence of strings. Two or more code fragments are similar if their textual representation, not considering the whitespace characters and comments, are alike. In these types of methods, generally little or no transformation is performed on the source code before starting the actual comparison. Since the raw source code is directly used in the clone detection process, one advantage of this method is that it is language-independent. But it is very sensitive for any change in the code clones like inserting or deleting code fragments. Mostly Type I duplicates can be identified by such methods.

Token-based methods. In the token-based detection approach, the entire source code is transformed into a sequence of tokens. This sequence is then scanned for similar chains of tokens. The original code parts corresponding the duplicated subsequences are returned as code clones. In these types of methods, transformations can be used to achieve better results. For example, whitespace characters and comments are usually removed during the tokenization and identifiers and literals can be replaced by placeholder tokes. For this reason, token-based methods can be more powerful compared to textual approaches. With such methods, type I. and most Type II clones can be identified. Token-based techniques are also used in the area of plagiarism detection. Some well-known plagiarism detection tools such as Winnowing [13] and JPlag [11] are based on token-based techniques.

Metrics based methods. Metrics based methods instead of comparing code directly, the source code is divided into syntactic units. For each unit, a value is generated based on its metrics, e.g., the number of tokens, the number of lines, the layout of expressions, etc. Two syntactic units with the same metrics are identified to be possible code clones. Several clone detection techniques use various software metrics for detecting similar code. Mayrand et al. [10] use a metric based approach to identify code clones. Metrics are calculated from names, layout, expression and (simple) control-flow of functions. A clone is defined only as a pair of whole function bodies that have similar metrics values. With this method, partially similar units are not detected.

AST based methods. Syntax-based methods are using an abstract syntax tree as a representation. An abstract syntax tree (AST) is a tree representation of the abstract (simplified) syntactic structure of source code written in some programming language. Code clone detection tools can use this representation to find similar subtrees in the AST and return the corresponding source code as clone pairs or clone classes. AST based detection techniques are dependent on the programming language. Techniques like consistent renaming can be used to abstract away the differences in identifier names and literals. This approach enables Type II clones to be detected. Detecting clones by comparing AST-s is desirable, but it is hard to scale.

The first non-text-based code clone detection method for the Erlang was developed for the Wrangler [1, 9] refactoring framework developed by H. Li and S. Thompson. This method uses a hybrid mechanism, a token-based technique to identify possible code duplicates that are later examined for semantic matching using the annotated AST.

A similar clone detection tool is built into the functionality of the HaRe [8] framework used for refactoring Haskell programs. The algorithm [3] is implemented in the framework by Christopher Brown and Simon Thompson, but in contrast to the Wrangler method, works purely on AST analysis and monadic analysis, and also allows for refactoring (elimination of clones) once duplicates are found.

4. Erlang and RefactorErl

Erlang [4] is a general-purpose, dynamically typed, concurrent functional programming language, which enables developers to write highly scalable soft real-time systems. Erlang was originally designed for developing telecommunication software, since then it is also widely used in the world of banking, chat services and database management systems. Due to its robustness and fault tolerance, it is suitable for the development of large-scale distributed systems. Erlang programs run within a virtual machine (Erlang VM or node), so programs written in Erlang are platform-independent. The standard library of Erlang is called OTP (Open Telecom Platform), the Erlang runtime environment and OTP are collectively called Erlang/OTP.

The module system of Erlang allows us to divide the program into smaller units: modules. Each Erlang program consists of modules, where each module is contained in a file with a *.erl* extension. The module declaration is located at the beginning of the modules. The module declaration is followed by the export declaration part, which is a list of functions defined in the module but intended to be used outside the module, and the function definitions.

A function declaration is a sequence of function clauses separated by semicolons, and terminated by a period. Each clause consists of a head, an optional guard condition, and a body. The head of a clause contains the name of the function, followed by the list of arguments, separated by commas. Each argument of the function is a valid pattern. The cases of a function definition are called function clauses, and they are separated from each other by a semicolon (`;`) token. The number of arguments is the arity of the function. A function is uniquely identified by the module name, its name and arity. That is, two functions defined with the same name and in the same module, but with different arities are two different functions.

A function clause is built up from either one expression, called the top-level expression, or a sequence of top-level expressions. The value of a function is the value of its last top-level expression.

Erlang is a dynamically typed language. This means that the type of expressions are not checked at compile time, only at runtime. For example, the expression $2 + "2"$ in the source code, does not raise a compile-time error. It causes a runtime exception.

RefactorErl [14, 2] is a static analyzer and transformer tool for Erlang, developed at Eötvös Loránd University. The tool uses static code analysis techniques and provides a wide range of features, like data flow analysis, dynamic function call detection, side-effect analysis, a user level query language to query semantic information or structural complexity metrics about Erlang programs, dependency examination among functions or modules, function call graph with information about dynamic calls, etc. The tool has multiple user interfaces to choose from: a web-based interface, an interactive console, and also supported by plugins for Emacs or Vim.

Static analysis is a method for examining the source code that is performed without actually executing programs. Static analysis can also be a useful method for debugging, code checking, software visualization, testing, and code comprehension. For static analysis, a representation of the source code is essential. The effectiveness of the analysis significantly depends on the chosen representation.

RefactorErl builds an abstract syntax tree from the source code during the initial analysis, and complements it with additional semantic information, thus creating a Semantic Program Graph (SPG) [6]. During our clone detection analysis we have used the SPG.

5. Algorithm for graph based clone detection

In this section, we present a new, graph-based clone detection method for Erlang programs. Our algorithm uses the semantic program graph of RefactorErl to classify possible clone groups and filters the results to identify clones.

The algorithm uses values computed for each vertex to represent the similarity of nodes. These values are derived from the attributes of the vertices of the SPG themselves. Based on these vertex values, another value is computed for each edge in the graph. A hash table is used to store these edge values, where each bucket contains a set of edges that are considered similar based on the attributes. This step serves as the initial phase of the partitioning process.

The similarity graphs is constructed based on the hash table in the following manner: any bucket containing more than two edges is combined into

an isomorphism vertex. The attributes of this isomorphism vertex are derived from the original vertices associated with the edges. These edges are created when the endpoints of a pair of vertices match, meaning the starting attribute of one vertex corresponds to the ending attribute of the second vertex, and vice versa.

The components of the similarity graph determine the code clone groups. Once the code clone groups have been generated the relevant clones need to be filtered out. For this, we use metrics that are calculated from the clone candidates using the number of subexpressions, tokens, variables, leaf expressions, guards, function calls etc. Code snippets are considered clones if they have have similar metrics values.

The attributes used to compute the edge values can greatly influence the results. If we target to find not only Type I clones (identical code fragments), then we need to abstract away identifier names and literals. Using the semantic program graph as our representation, we have access to different attributes describing the properties of the nodes. Our method is based on grouping subgraphs of the SPG together based on their similarity of structure and the corresponding attributes of their nodes.

5.1. The semantic program graph

RefactorErl represents an Erlang program with a directed labelled graph, called the Semantic Program Graph (SPG). The SPG has several type of nodes with different attributes describing the properties of the nodes. After the source code is analysed, this graph is stored in a database.

An SPG is a structurally weakly connected, directed graph whose vertices maps to the lexical, syntactic and semantic elements of a program. This graph includes all the information that can be statically extracted from the source code with different analyses. The SPG can be divided into three layers in terms of types of stored information. The first is the lexical layer, the second is the syntactic layer and the third is the semantic layer. The lexical layer contains both the original and the preprocessed tokens of the programs storing the Erlang source code as it is, preserving its formatting, while the syntactic layer stores the abstract syntax tree of the preprocessed source code. The semantic layer contains the results of the different kinds of semantic analyses, such as side-effect analysis, data-flow analysis, dynamic function reference analysis, control-flow analysis.

To better illustrate the structure of the SPG, we present below a brief example code snippet 3 and the corresponding SPG constructed by RefactorErl, which is shown in Figure 1. Our example Erlang module has a module declaration, an export list that defines its interface, and a simple function definition. The function takes an argument and increments it by one. As we can see on

Figure 1, the SPG built during source code analysis, unlike AST, is not a tree but can contain circles and even loops, but it always has a root.

Listing 3. Example Erlang code snippet

```
-module(example).
-export([inc/1]).

inc(A) ->
    A+1.
```

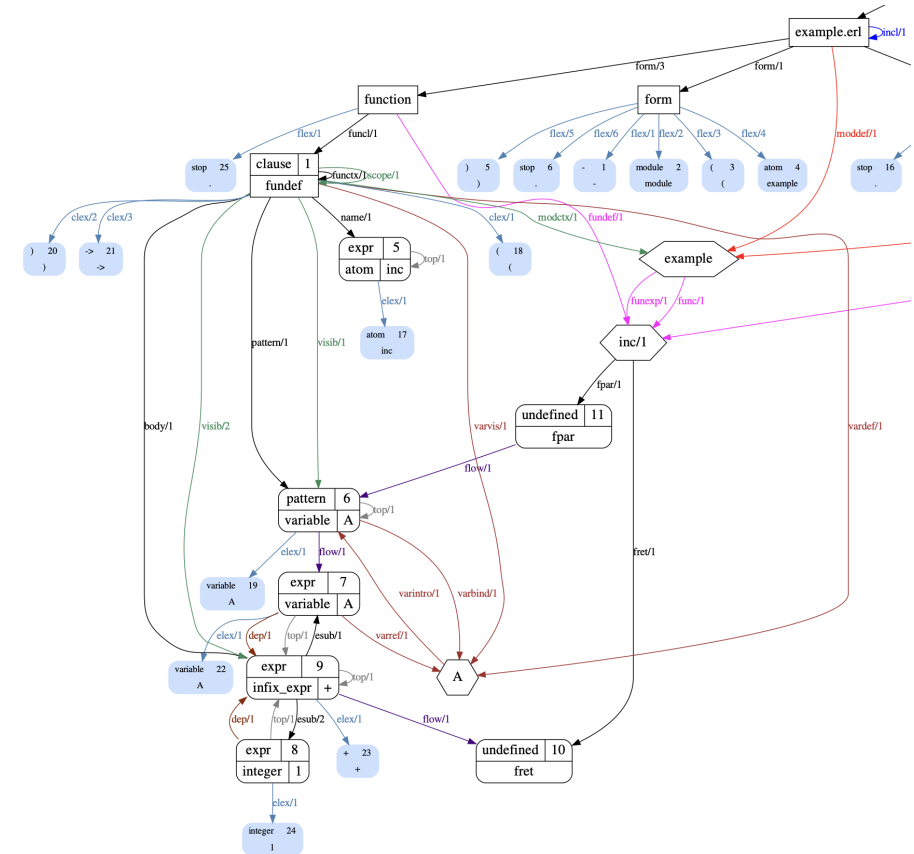

Figure 1. SPG example

5.2. Similarity graph

The first important step of determining the possible sets of code clones is building the similarity graph. When using the graph representation, code clones show up as similar subgraphs in the SPG. Graph similarity is expressed by graph isomorphism. Finding isomorphic subgraphs in an arbitrary graph is an NP-complete problem in its generalized form. While it would be possible to find the exact code clones in the SPG by comparing every possible subgraph - it would not be feasible. Instead, we will use an inexact method to group similar subgraphs together.

The similarity graph is a disconnected directed graph, where every component determines a set of similar subgraphs. The corresponding source code of these similar subgraphs is returned as possible sets of clone candidates.

5.2.1. Initial classification of vertices and edges

Algorithm 1 demonstrates how we build the table containing the classified edges. First, the SPG is traversed while labels are generated for the nodes. These labels will determine the baseline classification of the nodes. Each node will be assigned a label based on a selected set of attributes of the node. If two vertices have the same label, they will belong to the same group. The vertices of the SPG may have many attributes, but we only need a well-chosen subset of them to determine the labels. For example, using the "Id" attribute of the nodes for determining the label would be pointless, since that attribute is unique for every vertex, every single vertex would get into a separate class (resulting in no code clones). If we target to find not only Type I clones (identical code fragments), then we also need to abstract away identifier names and literals. In this representation using the SPG, we omit the attributes containing these data for the label generation. Choosing the set of attributes to generate the labels from enables one to do the parameterized matching. The function that determines the labels can be thought of as a hash function where the parameters are a list of attributes and the result is the computed label.

$$label(a_1, a_2, ..a_n) = v$$

For example, in the implemented prototype algorithm, we use the type of an expression as a label of a node representing an expression. However, in a case of a variable we do not use the name of the variable, because then the algorithm would not be able to detect Type II clones where a variable is renamed.

After the labels of the vertices have been determined, we can start the classification of the edges. An edge of a directed graph $G = (V, E)$ can be represented as an ordered pair of vertices $(u, v) \in E$. We can simply assign labels to the edges by concatenating the already calculated labels of the vertices.

Since the graph is directed, the order of the labels does matter, we will get a different edge label for an edge (u, v) from an edge (v, u) (unless the labels assigned to v and u are the same).

With the help of the edge labels, we can classify the edges into a hash table. We can use the edges and the computed labels as key-value pairs. Thus, the edges with the same labels will be placed in the same bucket.

Algorithm 1 Classification of edges

Funct BuildHashTab(G)

1: $(V, E, \alpha, \beta) := G$
2: $Tab := \emptyset$
3: **for** $(v_1, v_2) \in E$: **do**
4: $\quad Key := hash((v_1, v_2))$
5: $\quad put(Tab, \{Key, (v_1, v_2)\})$
6: **end for**
7: Return Tab

Since the SPG representation of even very simple modules is too big for visualization, to demonstrate the general algorithm we will show a simple example of a general directed graph (not an actual SPG) with node labels shown in Figure 2. In this example, we can assume that the classification of the vertices is already done and they are labelled according to their corresponding attributes. Table 2 shows the table containing the classified edges.

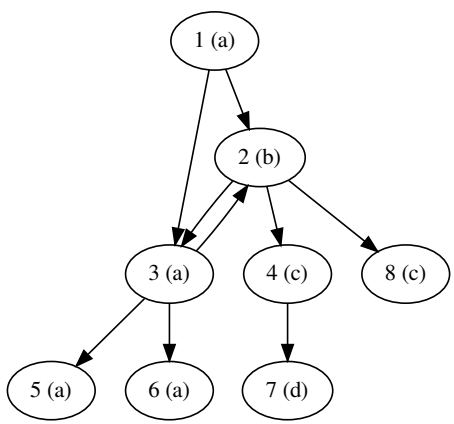

Figure 2. General example graph

Algorithm 2 demonstrates how we build the hash table from the given example graph and Table 2 shows the built similarity graph. First we iterate through the buckets of the hash table and for every bucket which contains two

[aa]	(1,3)	(3,5)	(3,6)
[ab]	(1,2)	(3,2)	
[ba]	(2,3)		
[bc]	(2,4)	(2,8)	
[cd]	(4,7)		

Table 1. Classification of the edges

aa	({1,3},{3,5,6})
ab	({1,3},{2})
cd	({2},{4,8})

Table 2. Edges of the similarity graph

or more elements, a new vertex is created in the similarity graph. All buckets containing two or more elements are grouped based on the number of contained elements. We iterate through the buckets of the hash table and combine the containing edges head and tail vertices as follows: if the bucket originally contained the $[(v_1, u_1), (v_2, u_2)...(v_n, u_n)]$ edges, we shrink the $V = (v_1; v_2; ...v_n)$ vertices and the $U = (u_1; u_2; ...u_n)$ vertices into two vertices, and they will determine a new (U, V) edge of the similarity graph.

Algorithm 2 Building of the similarity graph

Funct build_sim_graph(Tab)

```
1:  SimGraph := ∅
2:  for Bucket ∈ Tab : do
3:     EdgeList := value(Bucket)
4:     FromSet, ToSet := ∅
5:     for (u, v) ∈ EdgeList : do
6:        add(FromSet, u)
7:        add(ToSet, v)
8:     end for
9:     add_super_vertex(SimGraph, FromSet)
10:    add_super_vertex(SimGraph, ToSet)
11:    add_edge(SimGraph, {FromSet, ToSet})
12: end for
13: return SimGraph
```

5.3. Detecting initial clones

Once the similarity graph has been built, we can determine the initial sets of possible code clones. Every connected component of the similarity graph determines a set of subgraphs in the SPG. These subgraphs are not necessarily

isomorphic, some may only match to a part of the whole component. Since the non-leaf nodes of the SPG represent non-terminals, not every subgraph of the SPG will represent a code fragment in the source code. For example if two code snippets contain a case expression the syntax tree of the snippets can be somewhat similar, but that does not necessarily mean that they are duplicates. Consider the following examples: 4

Listing 4. Snippet A

```
not_beach(X) ->
    case X of
        0 -> zero;
        1 -> one;
        _ -> other
    end.

%%%%

beach(Temperature) ->
    case Temperature of
        {celsius, N} when N >= 20, N =< 45 ->
            'favorable';
        {fahrenheit, N} when N >= 68, N =< 113 ->
            'favorable in the US';
        _ -> 'avoid beach'
    end.
```

Both of these code snippets have a case expression in their syntax tree, but they have different patterns, guards, and clauses. The similarity graph generated from the code will contain the edges that overlap, but will not contain any leaves.

These can be filtered out in the component level by checking whether or not the subgraph contains a leaf node. We can also filter out components containing too few nodes. These components represent code fragments too small to be relevant as code duplicates.

5.4. Filtering relevant clones

Once the code clone groups have been generated we need to filter out the relevant clones. Code clone groups often contain pieces of code whose SPG-s do not overlap sufficiently with each other to be considered true clones. To measure the similarity between code snippets we use various metrics and properties. Metrics are calculated from the number of subexpressions, tokens, variables, leaf expressions, guards, function calls etc. Code snippets are considered clones

if they have have similar metrics values. The algorithm can be parameterized to how strictly to filter out potential code clones based on these metrics. If you want to detect type III. duplicates, you should not set very low thresholds for the similarity metrics, because inserting and deleting code snippets will cause many of these metrics to vary, but this can also result in a significant increase in false positives. In Section 6 we showed an example of the found Type III clones (Figure 5).

If the duplicate group is a case, if or try expression, the number and type of conditions or guard expressions is also considered. For example, if we assume that the code snippets previously presented 4 were placed in a clone group at the initial categorization, it is determined during the refinement process that they are not actually duplicates.

6. Recognized types of code duplicates

In this chapter, we will present some types of similarities that can be effectively recognized by the algorithm. Let us consider the Type I or text-identical duplicates first. Since the similarity graph is obtained by processing the semantic program graph and the same graph structure belongs to two identical code snippets in the SPG, the attributes of the vertices (except for the unique "Id" value) will also be the same. In this way, the algorithm always recognizes such code snippets as duplicates. In the code snippet below, we provide such an example.

Listing 5. Type I clones

```
double([]) -> [];
double([X | XS]) -> [2*X | double(XS)].
```

Let us consider that the previous identical code fragment (Listing 5) is defined in two different loaded modules. The subgraphs corresponding to the code snippets were grouped in one component of the similarity graph. After processing the similarity graph, the two fragments were flagged as code clones. The following Figure 3 shows the subgraphs corresponding to the code snippets in Listing 5. To differentiate between the two isomorphic subgraphs, we have also shown in the figure the original unique identifiers of the nodes in the SPG.

In the case of Type II duplicates, as previously mentioned, it is necessary to abstract away identifier names and literals. Since the variables, literals and constants are not distinguished by name or value when constructing the similarity graph, Type II code clones will correspond to subgraphs with the same structure so they will be grouped in the same component. In Figure 4 we can see such an example with the corresponding code fragments presented in Listing 6.

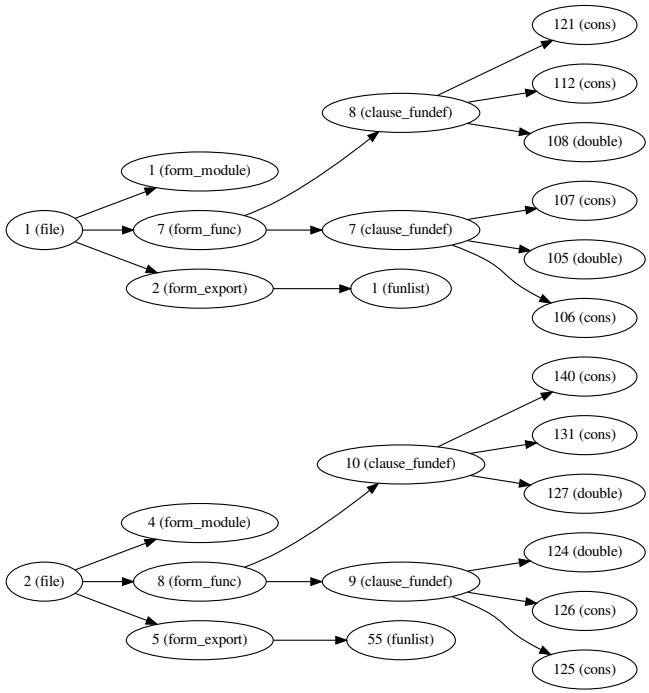

Figure 3. Type I code clones

Listing 6. Type II clones

```
foo(A) ->
    Fun1 = fun(Par1) ->
               {A,B} = Par1,
               A+B
           end,
    Fun1(A).

    ...

bar() ->
    Fun2 = fun(Par2) ->
               {C,D} = Par2,
               C+D
           end,
    Fun2({1,2}).
```

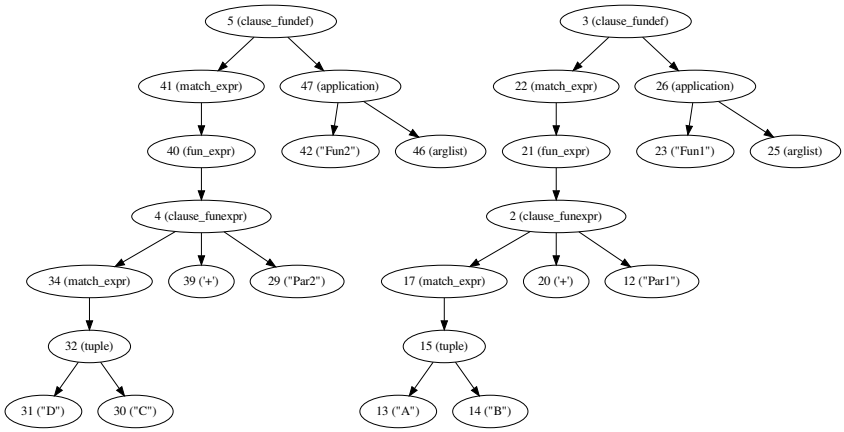

Figure 4. Type II code clones

In Type III code clones the code fragments might be slightly different: statements, expressions are changed, added or removed. These types of duplicates are usually created by modifying the copied code snippet. After the modification, the two snippets of code do not necessarily match syntactically or semantically, but we can say they are similar. In these cases the subgraphs corresponding to the code clones only partially overlap. The code snippets in Listing 7 [4] below show such an example. As we can see, both fragments contain a case expression, but in the first example includes one extra case that is missing from the second example. As we can see in the corresponding subgraphs shown in Figure 5, the subgraphs mostly overlap. To highlight the difference in the subgraphs, the edges missing from the subgraph corresponding to the second code snippet is coloured red.

Listing 7. Type III code clones

```
beach1(Temperature) ->
    case Temperature of
        {celsius, N} when N >= 20, N =< 45 ->
            'favorable';
        {kelvin, N} when N >= 293, N =< 318 ->
            'scientifically-favorable';
        {fahrenheit, N} when N >= 68, N =< 113 ->
            'favorable-in-the-US';
        _ ->
            'avoid-beach'
    end.
```

```
beach2(Temperature) ->
    case Temperature of
        {celsius, N} when N >= 20, N =< 45 ->
            'favorable';
        {fahrenheit, N} when N >= 68, N =< 113 ->
            'favorable-in-the-US';
        _ ->
            'avoid-beach'
    end.
```

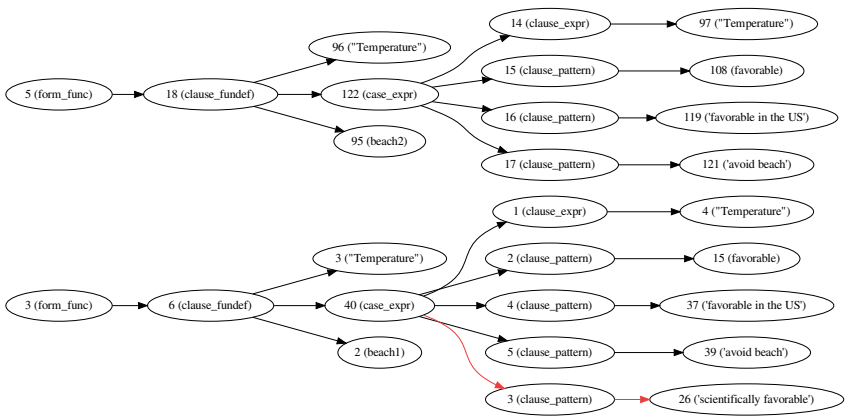

Figure 5. Type III code duplicates

In addition to our small examples, we also tested the prototype algorithm adapted to the RefactorErl graph on several open source libraries. The first examined library was the source code of Mnesia DBMS [5] (Database Management Systems). Mnesia is a distributed, soft real-time database management system written in Erlang. The source code of Mnesia is easily accessible as part of the OTP framework and has a relatively large code base - more than 22 thousands of lines of source code. Because of this, we found the Mnesia source code suitable for testing the algorithm. Since the algorithm does not require further processing of the raw source code and only uses the SPG built by RefactorErl to analyze the code base, the prototype of the algorithm processed the code base of Mnesia in a few minutes.

The Mnesia source code did not contain textually-identical (Type I) duplicates, but the algorithm identified some parametrical identical ones. Such an example can be seen in Listing 8 and Listing 9 source code snippets.

Listing 8. Code clones in the mnesia_lib modul

```
%15 (clause_block)
% ...
        case Storage of
            disc_only_copies ->
                dets:select(Tab, Pat);
            {ext, Alias, Mod} ->
                Mod:select(Alias, Tab, Pat);
            _ -> ets:select(Tab, Pat)
        end
% ...
```

Listing 9. Code clones in the mnesia_lib modul

```
%23 (clause_block)
% ...
        case Storage of
            disc_only_copies ->
                dets:match_object(Tab, Pat);
            {ext, Alias, Mod} ->
                Mod:select(
                    Alias, Tab, [{Pat, [], ['$_']}]);
            _ -> ets:match_object(Tab, Pat)
        end
% ...
```

Listing 10. Type 3 clones in mnesia

```
% From mnesia example 1

lock_record(Tid, Ts, Tab, Key, LockKind)
                                when is_atom(Tab) ->
    Store = Ts#tidstore.store,
    Oid = {Tab, Key},
    case LockKind of
        read ->
            mnesia_locker:rlock(Tid, Store, Oid);
        write ->
            mnesia_locker:wlock(Tid, Store, Oid);
        sticky_write ->
            mnesia_locker:sticky_wlock(Tid, Store, Oid);
        none ->
            [];
        _ ->
```

```
                abort({bad_type, Tab, LockKind})
    end;

% From mnesia example 2

lock_table(Tid, Ts, Tab, LockKind) when is_atom(Tab) ->
  Store = Ts#tidstore.store,
  case LockKind of
    read ->
        mnesia_locker:rlock_table(Tid,Store,Tab);
    write ->
        mnesia_locker:wlock_table(Tid,Store,Tab);
    load ->
        mnesia_locker:load_lock_table(Tid,Store,Tab);
    sticky_write ->
        mnesia_locker:sticky_wlock_table(Tid,Store,Tab);
    none ->
        [];
    _ ->
      abort({bad_type, Tab, LockKind})
  end;
```

Our prototype algorithm could be configured to find duplicates on different levels. We tested the algorithm to find duplicate clauses, which are a bigger units, and on the level of expressions, which are much smaller units. When searching for duplicate clauses, there were fewer results, but the analysis yielded many subgraphs that although indeed isomorphic, did not cover syntactically correct code snippets. Extending there subgraphs to the AST we got subtrees where the structure of the upper part of the tree was largely the same among the code clones grouped together, there were cases where the difference between the found code snippets was too large at the token level to consider them real clones. However, when searching for expressions, the duplicates were broken up into smaller pieces as demonstrated in examples 10, these were less useful than finding the full code snippet. The prototype algorithm was examined on multiple open-source modules like crypto, mnesia and eunit. Our analysis found in crypto 17 groups and 117 code snippets, in eunit 36 groups and 105 code snippets, in mnesia 111 groups and 492 code snippets and in ssl 151 groups and 1248 code snippets. It is worth noting that the large amount of duplicate candidates in ssl mainly originates from macro calls and map data structures (Listing 11). After manual inspection of the clone groups and duplicates found in the modules, in crypto 87%, in eunit 83% in mnesia 70% and in ssl 72% of overall of the results could be considered duplicates. Table 6 shows the number of duplicate groups, duplicates and the number of false positive results of our

prototype algorithm broken down into the examined modules.

Listing 11. Similar maps in SSL module

```
...

124893 (assoc_map_expr)
    #{id       => tls_dist_server_sup,
      start    => {tls_dist_server_sup,
                   start_link, []},
      restart  => permanent,
      shutdown => 4000,
      modules  => [tls_dist_server_sup],
      type     => supervisor
     }
161191 (assoc_map_expr)
    #{id       => ssl_listen_tracker_sup,
      start    => {ssl_listen_tracker_sup,
                   start_link, []},
      restart  => permanent,
      shutdown => 4000,
      modules  => [ssl_listen_tracker_sup],
      type     => supervisor
     }
161217 (assoc_map_expr)
    #{id       => tls_server_session_ticket,
      start    => {tls_server_session_ticket_sup,
                   start_link, []},
      restart  => permanent,
      shutdown => 4000,
      modules  => [tls_server_session_ticket_sup],
      type     => supervisor
     }
...
```

	LOC	# groups	# candidates	% of duplicates
crypto	3321	17	117	87.17 %
eunit	4029	36	105	83.81 %
mnesia	24526	111	492	70.73 %
ssl	34096	151	1248	72.01 %

Table 3. Module statistics

7. Conclusion and future works

Code duplicates are generated by various reasons such as code reuse by copying existing fragments of code (copy-and-paste programming). While it is possible that such cloning is intentional and can be useful in some, generally in the long term the presence of duplicate code makes software maintenance more difficult and thus it is considered a bad practice. Studies show that code duplicates not only inflate maintenance costs but also considered defect-prone as inconsistent changes to code duplicates can lead to unexpected behavior [12].

Several tools exist using different approaches and methods that can be used to identify code duplicates within the source code. The majority of these tools focuses on a specific language. There are also some attempts of creating general tools for code duplicate identification in source codes written in an arbitrary language. These methods are mostly based on pattern matching on the raw source code mostly using a sliding window algorithm or analyzing a sequence of tokens generated from the source code. These methods can be used to identify completely identical snippets of code, but they cannot be reliably used to detect code clones that are slightly different (have been modified). The more refined tools for identifying code duplicates exist mostly for mainstream languages, in the context of functional programming, only a few of them are available.

In this paper, we presented a graph-based algorithm which can find different types of code clones in the source code. We use the representation of Erlang programs defined by RefactorErl (a static analyser and transformer tool) to find code clones based on their corresponding subgraphs in the SPG. The algorithm was able to efficiently detect Type I, Type II and most Type III clones in the source code, as we have demonstrated with several examples. Since the semantic program graph provided by RefactorErl is a complete and easy to examine the representation of the source code, the algorithm presented in the paper does not require further processing of the raw source code. Due to the nature of the graph structure, it provided easily accessible information for the duplicate code detection than if we would have chosen a different representation. The algorithm uses an edge labelling function classify the edges of the SPG and builds a similarity graph that determines a degree of similarity between the subgraphs. With this method, we can group the similar subgraphs of the SPG, each set determining a class of code clones. By selecting which attributes of the nodes of the SPG the labelling function uses, we can influence what types of code duplicates the algorithm can detect.

The prototype algorithm was examined on multiple open-source modules like crypto, mnesia and eunit. We tested the algorithm to identify duplicates on the clause as well as on the expression level. While searching for duplicate clauses, we obtained fewer results. However, the analysis revealed numerous subgraphs that were isomorphic but did did not cover syntactically correct

code snippets. By expanding these subgraphs to the AST, we obtained subtrees where the upper portion of the tree was largely the same among the code clones grouped together, there were cases where the difference between the found code snippets was too large at the token level to consider them real clones. On the other hand, when searching for expressions, the duplicates were often fragmented into smaller components, these were less useful than finding the full code snippet. After manual inspection of the clone groups and duplicates found in the modules, in crypto 87%, in eunit 83% in mnesia 70% and in ssl 72% of overall of the results could be considered duplicates.

In the future, we will further evaluate and improve our methods. One obvious development is the refinement of the filtering method used on the set code clones. Since the similarity of subgraphs of the SPG is mainly based on the syntactic structure, but they do not always correspond to actual clones in the source code. Sometimes some false-positive results are also included in the result. Better filtering of the initial sets of subgraphs would help filter out irrelevant clones thus improve the accuracy of the results.

References

[1] Wrangler, cs.kent.ac.uk/projects/wrangler/Wrangler/Home.html [Acc. 04.10.2020].

[2] **Bozó, I., D. Horpácsi, Z. Horváth, R. Kitlei, J. Köszegi, M. Tejfel and M. Tóth,** Refactorerl - source code analysis and refactoring in erlang, in: *Proceedings of the 12th Symposium on Programming Languages and Software Tools, ISBN 978-9949-23-178-2*, pages 138–148, Tallin, Estonia, 10 2011.

[3] **Brown, C. and S. Thompson,** Clone detection and elimination for haskell, in: *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '10, pages 111–120, New York, NY, USA, 2010. ACM.

[4] **Cesarini, F. and S. Thompson,** *Erlang Programming: A Concurrent Approach to Software Development*, O'Reilly Media, 2009.

[5] Mnesia DBMS, http://erlang.org/doc/apps/mnesia/Mnesia_overview.html GitHub, 2019.

[6] **Horváth, Z., L. Lövei, T. Kozsik, R. Kitlei, A. Nagyné Víg, T. Nagy, T., M. Tóth and R. Király,** Modeling semantic knowledge in Erlang for refactoring, in: *Knowledge Engineering: Principles and Techniques, Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, KEPT 2009*, volume 54(2009) Sp. Issue of *Studia Universitatis Babeş-Bolyai, Series Informatica*, pages 7–16, Cluj-Napoca, Romania, 7 2009.

[7] **Jürgens, E., F. Deissenboeck, B. Hummel and S. Wagner,** Do code clones matter? *CoRR*, abs/1701.05472, 2017.

[8] **Li, H. and S. Thompson,** Tool support for refactoring functional programs, in: *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '08, page 199–203, New York, NY, USA, 2008. Association for Computing Machinery.

[9] **Li, H. and S. Thompson,** Similar code detection and elimination for erlang programs, in: Manuel Carro and Ricardo Peña, editors, *Practical Aspects of Declarative Languages*, pages 104–118, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[10] **Mayrand, J., C. Leblanc and E. Merlo,** Experiment on the automatic detection of function clones in a software system using metrics, in: *Proceedings of the 1996 International Conference on Software Maintenance*, ICSM '96, page 244, USA, 1996. IEEE Computer Society.

[11] **Prechelt, L., G. Malpohl and M. Philippsen,** Finding plagiarisms among a set of programs with jplag, *Journal of Universal Computer Science*, **8** (2002), 1016–1038.

[12] **Roy, C.K., J.R. Cordy and R. Koschke,** Comparison and evaluation of code clone detection techniques and tools: A qualitative approach, *Sci. Comput. Program.*, **74(7)** (2009), 470–495.

[13] **Schleimer, S., D.S. Wilkerson and A. Aiken,** Winnowing: Local algorithms for document fingerprinting, in: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 76–85, New York, NY, USA, 2003. ACM.

[14] **Tóth, M. and I. Bozó,** Static analysis of complex software systems implemented in erlang, Central European Functional Programming Summer School – Fourth Summer School, CEFP 2011, Revisited Selected Lectures, Lecture Notes in Computer Science (LNCS), Vol. 7241, pp. 451–514, Springer-Verlag, ISSN: 0302-9743, 2012.

I. Bozó, Zs. Erdei and M. Tóth
ELTE, Eötvös Loránd University
Budapest
Hungary
bozo_i@inf.elte.hu
zsanart@inf.elte.hu
toth_m@inf.elte.hu