Acta Cybernetica **22** (2016) 553–571.

Identifying Code Clones with RefactorErl*

Viktória Fördős† and Melinda Tóth‡

Abstract

Code clones, the results of "copy&paste programming", have a negative impact on software maintenance. Therefore several tools and techniques have been developed to identify them in the source code. Most of them concentrate on imperative, well known languages, while in this paper, we give an AST/metric based clone detection algorithm for the functional programming language Erlang. We propose a standalone solution that does not overload users with results that are insignificant from the point of view of the user. We emphasise that the maintenance costs can be decreased by using our solution, because the programmers need to deal only with important issues.

1 Introduction

Duplicated code detectors [3, 7] help in the identification of clones. Various approaches [23] have been proposed, including the analysis of code tokens [17], the syntax tree built up using the tokens [22], and using different metrics [22]. The majority of these methods and algorithms have been constructed specifically for the imperative paradigm and its mainstream languages, whilst in functional programming only a few exist, such as [6] developed for the Haskell language, and [11, 15] for the Erlang language [2]. Hitherto, none of the published papers dealt with the issue of *irrelevant clones* and proposed a standalone solution to the problem of presenting only the *relevant clones* to the user.

In practice, the set of (initial) clones is too large and contains many *false positive* or *irrelevant clones*. Therefore further operations are needed to narrow down the result set to serve the user only valuable, relevant clones as results.

First of all, we should discuss the difference between *false positive* and irrelevant clones. False positive clones are not real clones, whilst irrelevant clones are real clones, but they are absolutely useless. Examples of both kinds of clones are shown in Figure 1.

Clone detection algorithms focus only on false positive clones during filtering, and do not usually deal with irrelevant clones. Our goal is to construct an algorithm which easily notices important clones. So, we have tried to filter out any

*Supported by Ericsson–ELTE-Soft–ELTE Software Technology Lab

†ELTE-Soft Ltd., E-mail: f-viktoria@elte.hu

‡Eötvös Loránd University, E-mail: tothmelinda@elte.hu

DOI: 10.14232/actacyb.22.3.2016.1

False positive clone	Irrelevant clone
`f(List) -> 1+length(List).` `g() -> self() ! message.`	`new_cg() -> #callgraph{}.` `new_plt() -> #plt{}.`

Figure 1: Examples of false positive and irrelevant clones

clones serving no useful purpose. We have observed that the complete result of the algorithm can be ruined by a huge amount of irrelevant clones, because the users are not capable of distinguishing important clones from irrelevant ones while they are being swamped with worthless details. Our filtering system is the second phase of Clone IdentifiErl, and is detailed in Section 4.4.

Clone detection is a special static analysis task and its precision is hugely influenced by the available information, therefore Clone IdentifiErl does not work directly on the source code.

RefactorErl [1, 5, 29] is a static source code analyser and transformer tool for Erlang. RefactorErl provides a representation that contains even more information about the source beyond that of the abstract syntax tree.

Contributions In this paper we introduce Clone IdentifiErl that is an AST/metric based algorithm, whose implementation exploits the advantages of RefactorErl to precisely detect clones in Erlang programs. We address the problem of serving only relevant clones as results by proposing an Erlang specific solution. Moreover, we compare our algorithm with other Erlang specific detectors and we discuss how this solution can also be tailored to efficiently deal with the typical irrelevant clones of other programming languages.

2 Related work

The clone research community has carried out significant research for the last two decades. Various clone detection approaches have been proposed. The simplest algorithm is the line-based detection [23], where the recurrences of source code lines are detected. Although the most commonly used techniques are token and syntax based methods [3, 4, 20], some approaches build a sequence database from the source code and use fingerprints for the detection of clones [24, 25]. Mayrand et al. [22] use a metric based approach to identify code clones. Mohammad et al. [19] dealt with clones that are active at runtime to determine the impact of these clones.

Only a few researchers dealt with the problem of irrelevant clones. Harsu et al. [14] have published a case study classifying the importance of clones. Jurgens et al. [16] have proposed an iterative, configurable clone detector, called ConQAT, that contains a filtering system. ConQAT can remove repetitive generated code fragments and overlapping clones by iteratively reconfiguring and rerunning its initial clone detector. Contrary to ConQAT, our approach is a standalone filtering system,

thus it can be plugged into an initial clone detector and necessitates neither iterative evaluation nor the reproduction of initial clones to filter out irrelevant clones. We have proposed a more flexible, language-independent filtering system [12] to fit the preferences of any user. This filtering system works with groups of clones to refine them based on the domain specific predicates given by the user.

Some research has been carried out to ease the comprehension of the result of duplicated code detection. Here, a key is the compactness of the result. But a duplicated code detector can only result in pairs of clones or groups of clones. The latter scenario is said to be more comprehensible. Although, if the representation of the algorithm does not aid in retrieving grouped clones, grouping the result is a further step. SeClone [18] supports automatic grouping on file-level type usage by using the Suffix Tree Clustering algorithm. Tairas et al. [27] use Latent Semantic Indexes (LSI) to group clone classes that are the result of a syntax-driven clone detection algorithm. In general, LSI uses the singular value decomposition technique to identify relations between terms and concepts in unstructured text. The proposed approach exploits LSI to reveal relationships among clone classes that are not based on syntactical structures. We have also proposed a solution [10] addressing this problem that can be used almost in any cases when the result consists of clones pairs.

3 Erlang and RefactorErl

Erlang is a declarative, dynamically typed, functional, concurrent programming language, which was designed to develop soft real-time, distributed applications.

The compilation unit of Erlang programs is called a *module*, which is built up from attributes and function definitions. The encapsulating module, the name of the function, and the arity of the function can identify a function uniquely in Erlang. Pattern matching features are a prominent way to define functions by cases. The cases of a function definition are called *function clauses*, and they are separated from each other by a `;` token. A one-arity function, which consists of two function clauses, is shown in Erlang source 1. This function will be our running example through out the paper.

A function clause is built up from either one expression, called the *top-level expression*, or a sequence of top-level expressions as defined in the Erlang grammar. There are no statements in Erlang, only expressions. Contrary to statements, every expression has a value, which is the value of its last top-level expression.

Two kinds of expressions, the *list comprehension* and the *record* expression, can be found in the implementation of almost all industrial applications. Thus our filtering system focuses on them, and we briefly introduce these expressions here.

List is a frequently used data structure in Erlang. A list comprehension is a built-in language feature of Erlang to manipulate a list, based on Zermelo-Fraenkel set theory [13]. A general list comprehension is shown in Erlang source 2, whose *generators* (`Gen1`, ... ,`GenN`) are responsible for producing the base set. The *filters* (`Filter1`, ... `FilterN`) of the list comprehension narrow the base sets. `Expr`,

```
clone_fun(L) when is_list(L)->
    ShortVar = L,
    A = 1,
    B = lists:max([I || I<-lists:seq(1, 10)]),
    (A == 1) andalso throw(badarg),
    self ! B;

clone_fun(_)->
    V = f(g(42)),
    LongVariableName = V,
    B = lists:max([J || J<-lists:seq(V, V*2)]),
    X = fun(E) -> E + B end,
    self ! X.
```

Erlang source 1: clone_fun/1 function definition form

```
[ Expr || Pattern1<-Gen1, Filter1, ... ,  PatternN<-GenN, FilterN ]
```

Erlang source 2: A general list comprehension

called a *head of the list comprehension*, is an expression, which is evaluated on every element of the generators for which all filters are true.

Due to its simplified and safe usage, a record is an important preprocessed language element of Erlang, which is similar to a struct in C. There are four kinds of record operations:

- gathering the index of a record field is a non-modifier record operation;
- accessing a value of a field is a non-modifier record operation;
- creating an instance of a record is a modifier record operation;
- modifying a value of a field is a modifier record operation.

RefactorErl supports the daily work of Erlang programmers with code comprehension and refactoring tools. It provides the ability to retrieve semantic information and metric values about the source code, to perform dependency analysis and to visualise the results of the analysis. It facilitates code reorganisation with clustering algorithms and several refactoring methods. The incremental and asynchronous analyser architecture allows the programmer to track source code changes. The tool has multiple user interfaces to choose from: a web-based interface, an interactive console or one can use Emacs or Vim with RefactorErl plugins.

The source code has to be loaded into RefactorErl in order to be analysed. While performing analyses, the tool builds a labelled, directed graph, called *Semantic Program Graph* containing lexical, syntactic and semantic information about the source code. Information from the Semantic Program Graph is gathered by the evaluation of path expressions and the traversal of the graph. The algorithm presented in this paper does use information from the Semantic Program Graph and metrics

of RefactorErl.

4 Clone IdentifiErl

In this chapter, we present a new algorithm for accurate clone detection. Our algorithm combines a number of existing techniques, but introduces also a novel filtering component, to be described in Section 4.4. To our knowledge, these techniques have never been used specifically in Erlang.

What does clone detection mean intuitively? One may try to compare every code fragment to every other. The original representation of a code fragment is too concrete, thus a generalised form of source code needs to be used. The similarity of each pair of code fragments can be represented by a matrix. The first component of our algorithm produces this matrix, which is detailed in Section 4.2. From this matrix, the *initial clones* can be extracted along diagonals. This is what the second component of our algorithm does, which is described in Section 4.3. Irrelevant clones can be found among these clones, which are removed by evaluating filters. This process is described in Section 4.4.

4.1 Unit

The unit of a clone instance has to be chosen as cautiously as possible. One of our goals was to design and construct an algorithm that can be successfully used on legacy code, so the source code of several Erlang programs were studied.

The abstraction level of Erlang is high. Due to this abstraction, an application written in Erlang is so brief that a line of Erlang code generally corresponds to 8 to 10 lines of C code. It follows that block-based algorithms cannot be used. It also follows that the size of the chosen unit should be small. Tokens and subexpressions are too small to be used efficiently and a function clause is not small enough, therefore a top-level expression becomes the unit of the algorithm.

The program text of a top-level expression is too particular, thus generalisation is needed. We convert the expressions into a formal language, which uses a formal alphabet. This formal language can hide the unneeded specialisations of the tokens.

A generalised top-level expression is a sentence over the fixed formal alphabet. Every word is produced based on the type of the token. Tokens are produced by tokenizing expressions in the same order as given by the lexical analyser. It is necessary to preserve this order to keep the characteristics of the original expression. The alphabet of the language is not injective, in order to hide unneeded differences, for example, the difference between a variable and a constant (either a number or an atom).

Example After generalisation, our running example will be as shown in Figure 2. All top-level expressions are indexed and generalised.

Index	Top-level expression	Generalised top-level expr.
	`clone_fun(L) when is_list(L)->`	
i-1	`ShortVar = L,`	`A=A`
i	`A = 1,`	`A=A`
i+1	`B = lists:max([I || I<-lists:seq(1, 10)]),`	`A=A:A([AlAvA:A(A,A)])`
i+2	`(A == 1) andalso throw(badarg),`	`(AfA)FA(A)`
i+3	`self ! B;`	`A!A`
	`clone_fun(_)->`	
j-1	`V = f(g(42)),`	`A=A(A(A))`
j	`LongVariableName = V,`	`A=A`
j+1	`B = lists:max([J || J<-lists:seq(V, V*2)]),`	`A=A:A([AlAvA:A(A,A*A)])`
j+2	`X = fun(E) -> E + B end,`	`A=x(A)zA+Ae`
j+3	`self ! X.`	`A!A`

Figure 2: The transformation part of the first component

4.2 Matrix

A code clone is usually a result of "copy&paste programming". As an example, assume that one has copied a three-unit long sequence and has modified the second unit of the sequence, but the order of the sequence has been kept unchanged.

Usually larger clones are preferred, so we want to collect the three-unit long sequence as one clone instead of collecting three one-unit long clones. To be able to do it, modifications should be handled flexibly. Our algorithm works primarily on a matrix, which is a view of the problem, with which the flexibility criteria can be satisfied. Each element of the matrix expresses the similarity between two expressions and while a clone is made by preserving the original, correct order of its elements, it is enough to focus on the diagonals of a matrix. In other words, the fragments of diagonals are completely isomorphic to the fragments of code sequences found in the code directly. We put this idea in perspective in the following subsections.

4.2.1 Introducing the matrix

Assume that every top-level expression is numbered (indexed) sequentially, as shown in Figure 2. By taking the cardinality of the indexes as the size (denoted by n), a square matrix can be constructed, whose elements express similarity between the defining rows and columns, which are the top-level expressions identified by their indexes.

The relation, denoted by *Similarity*, between two top-level expressions, has the following properties:

- *Similarity* is reflexive, namely all values are related to themselves.

	1	$i-1$	i	$i+1$	$i+2$	$i+3$	n
1	...	...	...	...	...	...	...
$j-1$	⋮	0.5	0.5	0.43	0.46	0	⋮
j	⋮	**1.0**	**1.0**	0.21	0	0	⋮
$j+1$	⋮	0.19	0.19	**0.94**	0.23	0	⋮
$j+2$	⋮	0.17	0.17	0.22	0.24	0	⋮
$j+3$	⋮	0	0	0	0	**1.0**	⋮
n	...	...	...	...	...	...	...

Figure 3: A Dice-Sørensen similarity matrix

- *Similarity* is symmetric.
- *Similarity* quantifies the similarity between two top-level expressions in a clear and distinctive manner.

If the symmetric property holds, then only the lower triangular matrix need to be computed. If the reflexive property also holds, it follows that the elements of the main diagonal do not need to be computed. With these two properties the volume of computation is greatly reduced to the following cardinality:

$$\tfrac{1}{2}n^2 - n.$$

Clone IdentifiErl uses Dice-Sørensen metric [8, 26] for determining similarity, which does satisfy the properties of *Similarity* relation, too. There is no reason why the metric should not be replaced with other string similarity metrics [28]. Let Dice-Sørensen metric be portrayed by the m function

$$m : String \times String \to [0, 1] \subset \mathbb{R}.$$

Let n be the cardinality of the top-level expressions, A be the n-sized, square matrix. Let *selecttle* be a selector function which returns the top-level expression indexed by the given index. Now the matrix can be defined as

$$A(i,j) ::= \begin{cases} m(selecttle(i), selecttle(j)) & \text{if } i,j \in [1,n], i < j; \\ 0 & \text{otherwise.} \end{cases}$$

Example Consider the code fragments shown in Figure 2 with indexes. The relevant part of the Dice-Sørensen similarity matrix is shown in Figure 3.

4.2.2 Patterns in the matrix

The clauses are clones of each other, except that line `(i-1)` differs from line `(j-1)` and line `(i+2)` also greatly differs from line `(j+2)`. Therefore, it can be said that

Index	Top-level expression	Generalised top-level expr.
	`clone_fun(L) when is_list(L)->`	
i-1	`ShortVar = L,`	`A=A`
i	`A = 1,`	`A=A`
i+1	`B = lists:max([I || I<-lists:seq(1, 10)]),`	`A=A:A([AlAvA:A(A,A)])`
i+2	`(A == 1) andalso throw(badarg),`	`(AfA)FA(A)`
i+3	`self ! B;`	`A!A`
	`clone_fun(_)->`	
j-1	`V = f(g(42)),`	`A=A(A(A))`
j	`LongVariableName = V,`	`A=A`
j+1	`B = lists:max([J || J<-lists:seq(V, V*2)]),`	`A=A:A([AlAvA:A(A,A*A)])`
j+2	`X = fun(E) -> E + B end,`	`A=x(A)zA+Ae`
j+3	`Y = lists:zip([1,2,3],[3,21]),`	`A=A:A([A,A,A],[A,A,A])`
j+4	`self ! X.`	`A!A`

Figure 4: The new definition of `clone_fun/1`

three clones are present: the first one is a one-unit long pair, namely `([i-1], [j])`, the second one is also a one-unit long pair, namely `([i+3], [j+3])`, and the third one is a two-unit long pair, namely `([i, i+1], [j, j+1])`.

The following pairs are related to each other according to relation *isClone* (which is formally defined in section 4.3):

$$\{\dots,(i-1,j),(i,j),(i+1,j+1),(i+3,j+3),\dots\} = isClone.$$

Thanks to the complexity of Erlang programs one-unit long clone pairs can still be relevant clones. However, multi-unit long clone pairs are preferred in practice.

To take another example, assume that the starting units of a k-unit long clone pair can be found at indices a and b (k is a positive, fixed integer). Then

$$\{(a+i,b+i) \mid i \in [0\dots k-1] \subset \mathbb{Z}\} \subseteq isClone.$$

As observed by Baker [3], every pair in the defined set is an element of the matrix, and based on a k-unit long clone pair one of the diagonals of the matrix can be partially formed.

There may exist clones that cannot be found among diagonals such as the following. Let us assume that the first clause of `clone_fun/1` is the same as shown in Figure 2, but its second clause contains one newly inserted top-level expression. The new definition of `clone_fun/1` is shown in Figure 4.

Clone pairs `([i+3],[j+4])` and `([i, i+1], [j, j+1])` are in different diagonals. If the instances of a clone differ from each other in that way, then the full clone cannot be collected from the same diagonal, for instance, when the cardinality of inserted, deleted or rewritten top-level expressions differ from each other.

To summarise, instead of finding any pattern in the matrix, it is enough to search in diagonals. Although a full clone cannot be collected from the same diagonal in every case, its parts can be collected from different diagonals.

4.3 Determining initial clones

While a clone may be divided into sub clones due to insertions, deletions or other kinds of modifications, it would be practical if a full clone could be gathered somehow. Therefore we need to add a new parameter, called the *invalid sequence length*. It is the maximum length of a sequence whose middle elements can differ more from each other than *threshold* would allow. This limitation to the elements is naturally needed because of the beginnings and the endings of the clones should be similar to each other. By introducing invalid sequence length, one can customise the allowable maximum deviation of a clone.

If the chosen metric is exactly a distance, its values should be normalised to $[0, 1]$ to be able to handle the threshold correctly.

Now, we are able to precisely define the *isClone* relation, which expresses whether two units are considered to be clones of each other. The Dice-Sørensen metric is portrayed by the m function, and $Threshold$ contains a non-negative real number that is less than one. Let *isClone* be a general Boolean function operating on string pairs as follows:

$$isClone : String \times String \to \mathbb{L}.$$

The truth set of this function is:

$$\lceil isClone \rceil ::= \{(a, b) \mid a \in String, b \in String, m(a, b) > Threshold\}.$$

As shown in Section 4.2, it is enough to focus only on the diagonals. Thus, if the set of diagonals is constructed first, the elements of the set can be computed in parallel, because every element of the matrix is affected by only one complete diagonal.

We calculate the initial clones [9] by traversing the diagonals in parallel. We check whether a pair of top-level expressions is a clone or not. In the former case we try to extend the initial clone candidate with a new pair of expressions. In the latter case we take into account the invalid sequence length and try to extend the candidate if it is allowed.

Working with diagonals has a deficiency: the gathered instances of a clone can overlap the natural boundaries of the clone. The overlap should be avoided if possible. So, a boundary needs to be defined as a trimming rule of the production of initial clones, as follows: every top-level expression of a clone must belong to the same function clause per instance. This rule works, because function clauses act like natural boundaries.

Example The three initial clones which are detected by the described algorithm with using 1 for invalid sequence length are shown below:

1. `LongVariableName = Var`
   and
   `ShortVar = L`

2. `A = 1,`
   `B = lists:max([I || I<-lists:seq(1, 10)]),`
   `(A == 1) andalso throw(badarg),`
   `self ! B`
   and
   `LongVariableName = Var,`
   `B = lists:max([J || J<-lists:seq(Var, Var*2)]),`
   `X = fun(E) -> E + B end,`
   `self ! X`

3. `A = 1`
   and
   `ShortVar = L`

4.4 Filtering and trimming unit

There is an important difference between one-unit long and multi-unit long clones. Due to the high abstraction level of the formal language used to tokenise the program text, and the usage of the similarity metric, lots of false positive and irrelevant clones appear in the set of initial clones if only the one-unit long clones are taken into consideration. It follows that the filters for one-unit long clones need to be stricter than the filters for the multi-unit long clones.

As described in Section 4.3, the first phase of the algorithm exploits the advantages of invalid sequence length to point out clones that cannot be found by some of the other algorithms, for instance, the algorithm [15] that primarily works on a suffix tree to gather the initial clones. This asset should be preserved, thus invalid sequence length is also used in the filtering unit to process the multi-unit long clones. During the filtering, it can happen that a multi-unit long clone is split into a one-unit long clone and the rest of the multi-unit long clone. In this case, the one-unit long clone has to be further processed by the filters that are relevant for one-unit long clones.

4.4.1 Algorithm of the filtering system

A clone appears in the result set of the algorithm only if it meets all the requirements which are stated in the corresponding filters. For all *clone* in $InitialClones$, we have:

$$\bigwedge_{Filter \in Filters} Filter(clone) \implies clone \in ResultClones.$$

Crucially, if there is a requirement (defined by one of the filters) cannot be fulfilled, the clone is dropped. That makes our filtering system easily extendible and also very efficient, because the evaluation of filters is short-circuit.

The basic idea behind the algorithm is shown in Algorithm 1, which is detailed in Algorithm 2 by focusing only on the multi-unit long clones. If the currently examined clone is a one-unit long clone, then the `FiltersForOneLongs` function is responsible for dealing with it. The `FiltersForOneLongs` function forms the conjunction of the results of the evaluated filters, which are dedicated to one-unit long clones. If the conjunction is true, then the examined clone is returned, otherwise an empty set is returned.

```
function FilteringAndTrimmingUnit(InitialClones, InvSeqLength)
    Clones ← ∅
    parallel for all Clone ∈ InitialClones do
        if isOneUnitLongClone(Clone) then
            Clones ← FiltersForOneLongs(Clone)
        else
            Clones ← FiltersForMultiLongs(Clone, InvSeqLength)
        end if
    end parallel for
    return Clones
end function
```

Algorithm 1: Filtering and trimming unit of the algorithm

The input of the algorithm is the set of the initial clones and the invalid sequence length. The execution can run in parallel on initial clones, which are given as input to Algorithm 1. Two scenarios can occur after applying the filtering system to a one-unit long clone pair: it can be accepted or removed from the final result. Applying the filtering system to a multi-unit long clone pair can result in multiple smaller, to-be-filtered clone pairs. Nevertheless to any scenario the algorithm terminates in a finite number of steps. The output of the algorithm is a set of clones, which are produced in parallel, so the result of the algorithm is comprised of the separate result sets.

A bit more explanation is needed for the `FurtherTrim` function. This function is responsible for trimming invalid items from the beginnings and endings of the given clone. The result of a trimming operation is a set, whose one-unit long clones are further filtered by the `FiltersForOneLongs` function, to check if the stricter filters, defined to deal with one-unit long clones, can be satisfied by these clones, too.

4.4.2 Description of the defined filters

In this subsection the filters are presented. They can be categorised into three groups. The first group is used on one-unit long clones, the second group deals with multi-unit long clones, and the third group is applied to every clone. The ideas behind the filters were based on separate case studies on the detected initial clones of a real life application, called Mnesia [21]. Mnesia, written in Erlang, is a database management system and belongs to the standard Erlang/OTP library. It

```
function FiltersForMultiLongs(Clone, InvSeqLength)
    Clones ← ∅
    AClone ← ⟨⟩
    InvSeqCount ← 0
    for all UnitPair ∈ Clone do
        if ∧FilterFun∈FilterFuns* FilterFun(UnitPair) then
            AClone ← AClone ⊕ ⟨UnitPair⟩
            InvSeqCount ← 0
        else
            if InvSeqCount < InvSeqLength then
                AClone ← AClone ⊕ ⟨UnitPair⟩
                InvSeqCount ← InvSeqCount + 1
            else
                Clones ← Clones ∪ FurtherTrim(AClone)
                AClone ← ⟨⟩
                InvSeqCount ← 0
            end if
        end if
    end for
    Clones ← Clones ∪ FurtherTrim(AClone)
    return Clones
end function
```

Algorithm 2: Filtering and trimming unit of the multi-unit long clones

consists of 22,594 non-empty lines of code, 31 modules, 1,687 functions and 5,393 top-level expressions. Thanks to our filtering system, all irrelevant clones, which are shown as examples of filters in this subsection, appear only in the set of initial clones found in Mnesia and are not present in the final result set.

Filters for one-unit long clones As stated earlier, stricter filters need to be used on one-unit long clones, because a huge amount of these clones are in fact useless.

The *Simple Expression filter* is responsible for dropping a pair of top-level expressions if at least one component of the pair is an atom, a number, a character, a variable, a list, a tuple, a record operation or a function application. It may seem too strict, but in practice nobody cares about clones like the following ones: `Res` and `false`.

The *Simple Match filter* is responsible for dropping a pair of top-level expressions if only one component of the pair is a match expression, or if both are match expressions with right-hand sides that would be dropped by the Simple Expression filter, or if both components are match expressions with right-hand sides whose referred functions differ from each other. An example can be the following pair: `true = ets:foldl(Insert, true, Tab)` and `DelObjs = mnesia_lib:db_get(Tab, K)`.

The *Simple Send filter* is responsible for dropping a pair of top-level expressions

if only one component of the pair is a send expression or if both components are send expressions with right-hand sides that would be dropped by the Simple Expression filter. For example, consider `ReplyTo ! {self(),Done}` and `Pid ! {self(), more, Slot}`.

Filters for multi-unit long clones First, notice that a clone is examined here in such a way that all filters are separately evaluated on all its units (see Algorithm 2) that are pairs of top-level expressions. If a pair exists that cannot satisfy all the filters, then the clone is trimmed further. While studying filters for multi-unit long clones, this notice should be kept in mind.

The longer a clone is, the more important it is. Thus, the constraints of the filters for multi-unit long clones need to be weaker and also more flexible than the constraints of the filters for one-unit long clones. Therefore, we have introduced a new filter type, called *compound filter*, which satisfies the previously stated requirements. A filter is a compound filter if it is evaluated in the following way. If the kind of the examined expression is a match expression or a send expression, then instead of the given expression, its right-hand side is taken into consideration. The algorithm benefits from using compound filters while dealing with multi-unit long clones, because these filters find the dominant part of an expression, so the evaluation of the filter shows relevant result.

The *Same Right-hand Side filter* is responsible for dropping a pair of top-level expressions if only one component of the pair is a send or match expression, or both components are send or match expressions with right-hand sides of the following kinds: atom, number, character, variable or string, or if both components are match expressions with right-hand sides whose referred functions differ from each other. For instance, consider `Dist2 = incr_node(Node, Dist)` and `Tab = element(1, Val)`.

The *Same Record Operation filter* – a compound filter – is responsible for dropping a pair of top-level expressions if only one component of the pair is a record operation or if both components are record operations which differ in the classification of the operation (either a modifier or a non-modifier) or differ in the fields of the referred record. An example is as follows: `Tid = D#decision.tid` and `Commit0 = P#participant.commit`.

Filters for any clones The following two filters focus on branching expressions or fun expressions, which are built up from clauses and are frequently used in Erlang.

The *Same Cardinality filter* – a compound filter – is responsible for dropping a pair of top-level expressions if only one component of the pair is a branching or fun expression or if both components are branching or fun expressions which differ in the cardinality of the clauses. As an example consider Figure 5.

The *Same Function Application filter* – a compound filter – is responsible for dropping a pair of top-level expressions if only one component of the pair is a branching or fun expression or if both components are branching or fun expressions

```
case Res of                    case Res of
    {'EXIT', Reason} ->            {'EXIT', {aborted, What}} -> abort(What);
        exit(Reason);              {'EXIT', What} -> abort(What);
    _ -> Res                       _ ->Res
end                            end
```

Figure 5: An example dropped by the Same Cardinality filter

which differ in the function calls considering at least one of its clause. For example, `Fun = fun(S) -> lists:suffix(S, File) end` and `Fun2 = fun(Frag, A) -> mnesia:foldr(ActivityId, Opaque, Fun, A, Frag, LockKind) end`.

The *Head of List Comprehension filter* – a compound filter – is responsible for dropping a pair of top-level expressions if only one component of the pair is a list comprehension, or if both components are list comprehensions and their head expressions differ either in the cardinality of their sub-expressions or in referred functions. For instance, consider `Dist = [{good, Node, 0} || Node <- Pool]` and `TableList = [{Tab, keep_tables} || Tab <- List]`.

Example From the three initial clones gathered from our running example, only the four-unit long clone is the result of the algorithm, the two one-unit long clones are filtered out. These clones are object lessons for irrelevant clones.

To demonstrate the necessity of both the filtering system and the stricter filters for one-unit long clones, one should consider that 346,130 one-unit long and 5,022 multi-unit long initial clones were found in Mnesia. The set of the initial clones can greatly be narrowed down by using all defined filters, thus the result of the algorithm is more comprehensible in practice. As a demonstration of the efficiency of the filtering system, consider that 351,152 initial clones were found in Mnesia, and this huge amount of initial clones was reduced to 801 by applying the filtering system. Neither irrelevant nor false positive clones were found in the result of the algorithm. A non-trivial example is shown in Figure 6.

We end this section by describing how Clone IdentifiErl should be advanced to detect clones in programs written in another programming languages. The first phase requires a parser that decomposes the analysed program into small syntactic units. These units should be tokenised to turn them into their generalised representatives by using a formal language. The remaining phases of the initial clone detection will use these generalised forms. The second phase requires more effort. At first, the categories of the irrelevant clones need to be determined with which the necessary filters can be characterised. To materialise these filters, it is almost certain that a static analyser need to be employed. Next, the filtering phase should be implemented by using the same technique as we have presented in this paper but by using the proper filters.

```
Left one (found in mnesia_loader): case ?catch_val(send_compressed) of
 {'EXIT', _} -> mnesia_lib:set(send_compressed, NoCompression),
 NoCompression; Val -> Val end

Right one (found in mnesia_controller):
    case ?catch_val(no_table_loaders) of
        {'EXIT', _} ->
            mnesia_lib:set(no_table_loaders,1),
            1;
        Val -> Val
    end
```

Figure 6: Clones from the `mnesia` library

5 Comparison with another Erlang specific algorithms

We have examined separate programs (Mnesia and the clustering application of RefactorErl) by using Clone IdentifiErl, our metric-based clone detector and a standard suffix-tree based approach. In this chapter, we discuss the lessons we learnt.

The result sets of the algorithms are slightly different. The difference originates from the separate theoretical backgrounds of the algorithms, as we will discuss in this chapter.

Regardless of the applied theories, real syntactic clones are reported by all algorithms. This is obvious if we consider that the instances of these clones are similar from any point of view. However, even if syntactic clones can be reported by all the algorithms, the metric-driven algorithm can overlook some clones that are smaller than a function.

Considering the differences between Clone IdentifiErl and the suffix-tree based algorithm, the differences originate from independent reasons. Generally speaking, Clone IdentifiErl is more powerful; it results in more clones than the suffix-tree based algorithm does.

Clone IdentifiErl is not restricted by the parameter that constraints clones to consists of at least a minimum number of tokens. Thus clones that are smaller than this minimum do not appear in the result of the suffix-tree based algorithm.

Due to the usage of string similarity metrics, Clone IdentifiErl can find not only real syntactic clones. This is a great advantage that allows the users to detect clones whose instances differ from each other because of small modifications. For instance, consider Erlang Source 3.

As described previously, the theories driving the algorithms greatly influence their results. All the algorithms work, but serve separate purposes. If those clones are in the focus that can easily be eliminated, then the usage of the suffix-tree based algorithm is advised because of its linear computational cost. If the constraints are

```
% First instance
sorensen_dice2(_E1, Attr1, _E2, Attr2) ->
 #presence{both=A,first=B,second=C,none=D}=presence(Attr1,Attr2),
 if
A+B+C+D == 0 -> 1;
true         -> 1 - 2*(A+D) / (2*A + B + C + D)
 end.

% Second instance
sorensen_dice(_E1, Attr1, _E2, Attr2) ->
 #presence{both=A, first=B, second=C} = presence(Attr1, Attr2),
 if
A+B+C == 0 -> 1;
true       -> 1 - 2*A / (2*A + B + C)
 end.
```

Erlang source 3: A clone can only be detected by using string similarity metrics

weakened and clones that are modified instances of each other are also concerned, then Clone IdentifiErl can come to the rescue. But if only clones that are functions are important, the metric-driven algorithm is the ideal candidate. We additionally note that the metric-driven algorithm is more broadly usable than it seems to be at first sight, because functions in Erlang are small; functions usually consist of a few (one to five) top-level expressions.

6 Conclusion and future works

Duplicated code detection is a special static analysis, where code clones are identified in the source code. Clones can result in several bugs and inconsistencies during software maintenance. Thus, programmers should at least be aware of their existence.

In this paper we have described and evaluated a duplicated code detection algorithm to identify code clones in Erlang programs. We have shown the main parts of Clone IdentifiErl by highlighting the filtering possibilities. We use the representation of Erlang programs defined by RefactorErl (a static analyser and transformer tool) to build the matrix and to evaluate filters.

We have discussed the problem of presenting only valuable clones to the user. We have learnt that the complete result of a duplicated code detector can be ruined if both irrelevant and relevant clones are presented. By taking the initial clones of Mnesia into consideration, we have demonstrated how serious this problem can be if the clones of a legacy code base need to be identified.

We have proposed an Erlang specific solution – the filtering system – which narrows down the set of initial clones by filtering out both irrelevant and false

positive clones. By applying the filtering system, Clone IdentifiErl can distinguish between irrelevant and relevant clones to provide a more comprehensible result to the users. To best of our knowledge, no paper have proposed a standalone solution to filter out irrelevant clones.

We want to further evaluate and improve our techniques. We are going to further study the results of our approach and tune the algorithm by altering the number of used filters.

References

[1] RefactorErl Homepage. http://refactorerl.com.

[2] Armstrong, Joe. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[3] Baker, Brenda S. A program for identifying duplicated code. In *Computer Science and Statistics: Proc. Symp. on the Interface*, pages 49–57, March 1992.

[4] Baxter, Ira D., Yahin, Andrew, Moura, Leonardo, Sant'Anna, Marcelo, and Bier, Lorraine. Clone Detection Using Abstract Syntax Trees. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 368–377. IEEE Computer Society, 1998.

[5] Bozó, I., Horpácsi, D., Horváth, Z., Kitlei, R., Kőszegi, J., Tejfel, M., and Tóth, M. RefactorErl - Source Code Analysis and Refactoring in Erlang. In *Proceedings of the 12th Symposium on Programming Languages and Software Tools*, pages 138–148, Tallin, Estonia, October 2011.

[6] Brown, Christopher and Thompson, Simon. Clone Detection and Elimination for Haskell. In Gallagher, John and Voigtlander, Janis, editors, *PEPM'10: Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 111–120, January 2010.

[7] Cordy, James R. and Roy, Chanchal K. The NiCad Clone Detector. In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*, ICPC '11, pages 219–220. IEEE Computer Society, 2011.

[8] Dice, Lee Raymond. Measures of the Amount of Ecologic Association Between Species. *Ecology*, 26(3):297–302, July 1945.

[9] Fördős, Viktória and Tóth, Melinda. Identifying Code Clones with RefactorErl. In *Proceedings of the 13th Symposium on Programming Languages and Software Tools*, pages 31–45, Szeged, Hungary, August 2013.

[10] Fördős, Viktória and Tóth, Melinda. Comprehensible presentation of clone detection results. In *To appear in AIP Conference Proceedings of the 12th International Conference of Numerical Analysis and Applied Mathematics*, Rhodes, Greece, September 2014.

[11] Fördős, Viktória and Tóth, Melinda. Utilising the software metrics of RefactorErl to identify code clones in Erlang. In *Proceedings of 10th Joint Conference on Mathematics and Computer Science*, volume LIX of *Informatica*, pages 103–118, Cluj-Napoca, Romania, May 2014. Studia Universitatis Babeş-Bolyai.

[12] Fördős, Viktória, Tóth, Melinda, and Kozsik, Tamás. Clone Wars. In *Proceedings of the 3th Workshop on Software Quality Analysis, Monitoring, Improvement and Applications*, volume 1266, pages 15–22, Lovran, Croatia, September 2014.

[13] Fraenkel, A.A. and Hillel, Y.B. *Foundations of Set Theory*. Foundations of mathematics. North-Holland Publishing Company, 1958.

[14] Harsu, Maarit, Bakota, Tibor, Siket, István, Koskimies, Kai, and Systä, Tarja. Code Clones: Good, Bad, or Ugly? In *Proceedings of 11th Symposium on Programming Languages and Software Tools and 7th Nordic Workshop on Model Driven Software Engineering*, pages 162–176, Tampere, Finland, August 2009.

[15] Huiqing, Li and Simon, Thompson. Clone detection and removal for Erlang/OTP within a refactoring environment. In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '09, pages 169–178. ACM, 2009.

[16] Juergens, Elmar and Göde, Nils. Achieving Accurate Clone Detection Results. In *Proceedings of the 4th International Workshop on Software Clones*, IWSC '10, pages 1–8, New York, NY, USA, 2010. ACM.

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

[18] Keivanloo, I., Rilling, J., and Charland, P. SeClone - A Hybrid Approach to Internet-Scale Real-Time Code Clone Search. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference*, pages 223–224, June 2011.

[19] Khan, Mohammad Asif A., Schneider, Kevin A, and Roy, Chanchal K. Active Clones: Source Code Clones at Runtime. In *Proceedings of the Eighth International Workshop on Software Clones*, volume 63 of *IWSC'14*, July 2014.

[20] Koschke, Rainer and Riemann, Ole Jan Lars. Robust Parsing of Cloned Token Sequences. In *Proceedings of the Eighth International Workshop on Software Clones*, volume 63 of *IWSC'14*, July 2014.

[21] Mattsson, Haakan, Nilsson, Hans, and Wikstrom, Claes. Mnesia - A Distributed Robust DBMS for Telecommunications Applications. In *PADL '99: Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, pages 152–163. Springer-Verlag, 1998.

[22] Mayrand, Jean, Leblanc, Claude, and Merlo, Ettore. 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, pages 244–. IEEE Computer Society, 1996.

[23] Roy, Chanchal K., Cordy, James R., and Koschke, Rainer. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Sci. Comput. Program.*, 74(7):470–495, May 2009.

[24] Schleimer, Saul, Wilkerson, Daniel S., and Aiken, Alex. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 76–85. ACM, 2003.

[25] Smith, Randy and Horwitz, Susan. Detecting and Measuring Similarity in Code Clones. In *3rd REF/TCSE International Workshop on Software Clones. Workshop proceedings of the 13th European Conference on Software Maintenance and Reengineering*, CSMR2009, pages 28–34, Kaiserslautern, Germany, March 2009. IEEE Computer Society.

[26] Sørensen, T. A method of establishing groups of equal amplitude in plant sociology based on similarity of species and its application to analyses of the vegetation on Danish commons. *Biol. Skr.*, 5:1–34, 1948.

[27] Tairas, Robert and Gray, Jeff. An Information Retrieval Process to Aid in the Analysis of Code Clones. volume 14, pages 33–56, Hingham, MA, USA, February 2009. Kluwer Academic Publishers.

[28] Thierry, Lavoie and Ettore, Merlo. About Metrics for Clone Detection. In *Proceedings of the Eighth International Workshop on Software Clones*, volume 63 of *IWSC'14*, July 2014.

[29] Tóth, Melinda and Bozó, István. Static analysis of complex software systems implemented in Erlang. In *Proceedings of the 4th Summer School Conference on Central European Functional Programming School*, CEFP'11, pages 440–498, Berlin, Heidelberg, 2012. Springer-Verlag.

Received 10th March 2014