STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LIX**, Special Issue 1, 2014
10th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, May 21-25, 2014

UTILISING THE SOFTWARE METRICS OF REFACTORERL TO IDENTIFY CODE CLONES IN ERLANG

VIKTÓRIA FÖRDŐS AND MELINDA TÓTH

Abstract. Code clones, the results of "copy&paste programming", are special types of bad smells. They have a negative impact on software development and maintenance lifecycle. The usual way to detect bad smells is to calculate software metrics. RefactorErl is a source code analysis and transformation tool for Erlang; it provides several software metrics to measure the complexity of the source code, and finds structures that violate some existing requirements or standards, or points out bad smells based on the results of them. In this paper we introduce an efficient, parallel, software metric based clone detection algorithm, which utilises software metrics of RefactorErl in an unusual way, for the functional programming language Erlang. We have successfully evaluated it on various open-source projects.

1. Introduction

Code clones are unwanted phenomena in the source code of several software. Although it is straightforward to create a new instance of an already existing source code fragment by copying, to identify them manually in an industrial scale software is complicated, time consuming and sometimes impossible. Therefore tools that support clone identification are highly desired in the software development and maintenance lifecycle.

Although several duplicated code detectors exist [7, 12, 18], only a few of them concentrate on functional programming languages [6, 8, 11]. The main goal of our work is to give an effective clone detector for the functional programming language Erlang [2].

Various techniques [17] have been developed to identify code clones. The general model of these algorithms is to measure the similarities in the source

Received by the editors: May 1, 2014.

2010 *Mathematics Subject Classification.* 68W40, 68W10.

1998 *CR Categories and Descriptors.* D.2.8 [**Software engineering**]: Metrics – *Software science*; D.1.3 [**Programming techniques**]: Concurrent Programming – *Parallel programming*.

Key words and phrases. software metrics, clone detection, Erlang, static analysis, bad smell, RefactorErl, accurate result.

code. This can be expressed by the similarity of the tokens, syntax or semantics.

RefactorErl [1, 5, 19] is a static source code analysis and transformation tool for Erlang. It analyses the source code and calculates lexical, syntactic and semantic information about Erlang programs. RefactorErl provides several metrics to point out bad smells and to check coding conventions.

In this paper we describe how we can utilise the metrics of RefactorErl to describe the lexical, syntactic and semantic structure of different source code parts. We introduce an algorithm to identify code clones based on the similarity and equality of these metric values.

2. Related Work

Various clone detection approaches have been proposed. The simplest algorithm is the *line-based detection* where the recurrences of source code lines are detected. This technique is not common.

The most commonly used techniques are token and syntax based methods [9, 3]. Some of the algorithms transform the source according to their characteristics over an abstract alphabet, and perform the clone detection on either this representation, or on a suffix-tree built from the representation. This technique is used by Wrangler [11] and the previous, unstable, unfinished prototype within RefactorErl. Others build a sequence database from the source code and use fingerprints for detection of clones [16].

Clone IdentifiErl [8] is a component of RefactorErl. It introduces an AST/metric based clone detection algorithm for Erlang.

Mayrand et al. use a metric based approach to identify code clones [15]. They group the used metrics into four points of comparison and define the cloning level (from exact copy to distinct functions) based on them. Although some metrics are general enough to apply this theory on functional languages as well, the method can not be applied on Erlang programs. The presented metrics do not characterise the language sufficiently.

3. Motivation and RefactorErl

RefactorErl is a source code analysis and transformation tool for the dynamically typed strict functional programming language Erlang. The main goal of the tool is to support effective software development and maintenance through refactorings and code comprehension assistance.

The tool supports more than twenty refactoring transformations, and provides several features to aid code comprehension:

- dependency analysis and visualisation,
- semantic query language to retrieve semantic information,

- investigation tracking,
- dynamic call analysis,
- data-flow detection,
- bad smell detection, etc.

The main features of the tool can be reached through a web-based interface, an interactive or a scriptable console interface or an editor plugin (Emacs or Vim).

RefactorErl provides more than thirty metrics to the users [10]. Software metrics characterise and describe some properties of the language elements. Once a language entity is similar to another language entity, the values of the measured metrics should be the same or similar. Therefore our aim is to utilise the metrics of RefactorErl to help the users to identify code clones in the software.

3.1. **Erlang.** Erlang was designed to implement highly concurrent, distributed, fault-tolerant systems with soft real-time characteristics. Although the dynamic nature of the language makes static program analysis complicated, a tool to identify relations in the source code statically is desired.

The language is declarative and functional. Pattern matching, higher-order functions, recursions, list comprehensions and other nice properties of functional languages are present in Erlang, but the language is not pure: functions may contain side-effects and sequencing.

An Erlang *function* is built from its *function clauses* describing the different ways of execution based on pattern matching and guards. A *function clause* is built from an expression sequence. The expressions in the sequence are called *top-level expressions*. Expressions can be *simple expressions* (e.g. `infix, send, function call`) or *branching expressions* (e.g. `case, if, receive, try`).

The definition of the factorial function with two function clauses is shown in Erlang source 1.

```
factorial(0) -> 1;
factorial(N) when N > 0 ->
     N1 = fact(N-1),
     N1*N.
```

Erlang source 1: The definition of function `factorial/1`

3.2. **Representation.** To calculate metrics, the source code has to be analysed at first. RefactorErl has an asynchronous parallel semantic analyser framework to handle the initial analysis, to build and to store the calculated

information. RefactorErl represents the source code in a three layered graph (called a *Semantic Program Graph*). The lexical layer is built from the tokens of the source code. After the preprocessor has finished, the syntax layer, containing the abstract syntax tree, is constructed. Later several semantic analysers build the semantic layer of the graph (e.g., module and function references, variable binding structure).

Information gathering is possible through graph traversing. RefactorErl provides well-defined querying interfaces. The algorithm presented in this paper uses information from the Semantic Program Graph to calculate the metrics and to reach the corresponding language elements.

4. Metric Based Clone Detection

In this section, we introduce a new algorithm for clone detection. Our algorithm, designed to be efficient and parallel, combines both some existing techniques and a novel filtering system. To our current knowledge, these techniques have never been used in Erlang.

Instead of designing another syntax-based algorithm, we have utilised software metrics to determine the similarity between code fragments to produce the initial clones. The set of initial clones can contain both irrelevant and false positive clones, so it is narrowed down by the filtering system. The filtering system applies another, stricter groups of metrics to determine the result set by filtering out useless clone pairs.

4.1. **Producing clone candidates.** One of the most important decisions that has to be made is the determination of the size of the smallest detectable clone. Due to the high abstraction level of Erlang, the algorithm deals with functions as units to select clones. We have observed that neither function clauses nor top-level expressions are large enough structures to characterise clones properly by software metrics.

Every function that has been loaded into RefactorErl is combined with every other function to form pairs. The set of clone candidates consists of these function pairs, and has the following cardinality for N functions:

$$\frac{N * (N - 1)}{2}$$

The algorithm determine the set of initial clones from this set.

4.2. **Determining initial clones.** Before we detail the first phase of our algorithm and show how software metrics can be used to measure similarity, we take a look at the topic of similarity in general.

4.2.1. *Similarity.* It can be said that two elements are similar to each other, if the values of their examined attributes are nearly equal to each other; and that the state of being similar depends on the studied features. Thus by changing the examined features, separate items from the same set can become similar to or absolutely different from each other. As described later, we can benefit from this vagueness. We have decided to determine the similarity among clone candidates by software metrics.

4.2.2. *Software metrics.* RefactorErl provides several ready-to-use software metrics and a Semantic Program Graph that is rich in information and easy to query. By gathering information from the Semantic Program Graph, not only lexical and syntactic but also semantic attribute based software metrics can be calculated. Each metric can be used to characterise separate features of code fragments. In general, one metric cannot characterise a code fragment completely, but several of them can. Thus, the more metrics are used, the more precise the result is.

Description	Maximum deviation
Number of alphanumeric characters located inside comments.	10
Number of comment lines inside the definition form of the function.	5
Number of unique macro applications.	2
Number of non-empty lines.	5
Average length of variable names.	3
Number of function clauses.	1
Number of guarded function clauses.	1
Number of such tokens that form the body of the function.	50

TABLE 1. List of metrics belonging to the Programming Style aspect

4.2.3. *Aspects.* Our chosen metrics can be grouped together by considering which aspect of the source code they characterise. We have formed 3 groups, called *aspects*, which are the following:

- *Programming style*: The Programming style metrics, shown in Table 1, characterise not only the layout of the source code, but also the programming style in which the application has been written. For instance, the average length of variable names is one such metric.
- *Expression*: In Erlang only expressions exist, they are the main building blocks of Erlang programs. Due to their significant dominance, we have dedicated a separate aspect to them. These metrics, shown in

Description	Maximum deviation
Number of top-level expressions.	5
Number of function calls.	5
Number of unique function calls.	2
Number of infix operator applications.	5
Number of clauses contained by branching expressions.	5
Number of guarded clauses contained by branching expressions.	4
Number of variable bindings.	4
Number of such bound variables that have no reference.	5
Number of fun expressions (lambda expressions).	2
Number of unique record references.	3
Number of expressions that have side-effects.	2

TABLE 2. List of metrics belonging to the Expression aspect

Table 2, are used to measure several features of expressions, because the way the expressions are constructed and used are absolutely descriptive. For example, the number of unused variables belongs to this aspect.

- *Flow of control*: Nearly any problem can be solved in several separate ways, so the way the control is defined is also an expressive aspect. This aspect, whose metrics are listed in Table 3, covers all of the control structures that can be used in Erlang. For example, the number of recursive calls or the number of started processes belong to it.

These aspects are independent from each other, and they describe the lexical, syntactic and semantic properties of the source code, respectively. The more aspects the elements of a clone candidate are found to be similar in, the more certain their connections are.

4.2.4. *Arbitrators.* We have assigned each aspect to be examined by separate *arbitrators*. Each arbitrator is responsible for judging each candidate as described below, after the necessary assumptions are defined.

Let $Metrics$ be the set of metrics whose elements need to be evaluated by the arbitrator. Let A and B be the elements of a clone candidate on which the metrics are evaluated.

Description	Maximum deviation
Number of such sequential or decision nodes that appear in the control flow graph of RefactorErl.	10
Number of control related decisions.	2
Number of recursive structures.	2
Number of exit points.	2
Number of computational nodes.	8
Average cardinality of nested branching expressions of top-level expressions.	2
Number of independent, maximised-length paths in the control flow graph of RefactorErl.	10
Number of message passings.	2
Number of started processes.	1
Number of raised exceptions.	2

TABLE 3. List of metrics belonging to the Flow of control aspect

If the arbitrator has detected the maximum amount of similarity between the elements of a clone candidate, then its vote is *promising*.

$$\forall M \in Metrics(M(A) = M(B)) \implies vote = promising$$

If the arbitrator has pointed out that the elements of a candidate are resembling to each other without being identical, then its vote is *perhaps*. To define it, we need to introduce a new function that assigns the maximum deviation to the given metric. The exact deviations of metrics, shown in Table 1, Table 2 and Table 3, are determined based on our empirical studies.

$$MaxDev :: Metrics \mapsto \mathbb{N}^+$$

The vote is *perhaps*, if the computed metric values of the elements of the investigated clone candidate differ from each other less than the corresponding maximum value.

$$\forall M \in Metrics(|M(A) - M(B)| \leq MaxDev(M)\ ) \implies vote = perhaps$$

If the arbitrator has found no similarities between the elements of a clone candidate, then its vote is *impossible*.

$$\exists M \in Metrics(|M(A) - M(B)| > MaxDev(M)\ ) \implies vote = impossible$$

4.2.5. *Evaluating votes.* By considering the votes of the arbitrators, 3^3 combinations can occur as a result for each clone candidate, but exactly one of them holds for each candidate at a time. However, we have found only four combinations of them (shown in Table 4) useful, which we are able to determine and also to categorise the initial clones with. Every clone candidate that does not belong to any of these four categories, is dropped; all of the remaining clone candidates form the set of initial clones.

Category	Flow of Control	Programming Style	Expression
Cat1	promising	promising	promising
Cat2	promising	perhaps	promising
Cat3	promising	impossible	promising
Cat4	promising	impossible	perhaps

TABLE 4. Categories of initial clones based on the votes of arbitrators

Why have we found only these four combinations useful? We first observe that only promising votes are present in the Flow of Control column. All of the metrics of Flow of Control aspect try to characterise the semantics of functions based on their control flow graphs, which are available within RefactorErl. If the semantics of two functions differ from each other, then we consider that these functions are likely to solve separate problems.

Furthermore, only the Programming Style column contains all of the possible votes. This is motivated by the following observations. Each programmer has his/her own style that is preserved by the source code. Although, all of the attributes of this aspect vanish after the source code has been compiled, we consider this aspect meaningful, because we have observed that usually when making clones both comments and the programming style in which the copied function has been written are preserved.

We also observe that no impossible votes can be seen in the Expression column. By considering that the main building blocks of Erlang programs are expressions, it follows that two functions built up from different expressions cannot form a clone.

An interesting but useless combination is when only the Programming Style is promising (identical), and the other votes are impossible. In this case, we have two functions, probably written by the same programmer, who either has a very unique style or no knowledge of the coding conventions.

For the Flow of Control aspect, the constraint that all of the values of these metrics must be equal to each other seemed too strict for selecting initial clones, so we have tried another solution. Specially, we modified the circumstances under which a promising vote can be given as defined below.

$\forall M \in Metrics(|M(A) - M(B)| < MaxDev(M) * 0.3\ ) \implies vote = promising$
This rule allows a smaller deviation, that depends on the original deviation. Contrary to our expectations, we observed that the cardinality of the initial clones becomes at least two times larger, but the growth of the result set is infinitesimal, thus we rejected this attempt.

4.3. **Narrowing down the set of the initial clones.** We have observed that syntax-driven algorithms produce fewer initial clones than our algorithm, but these algorithms can overlook some clones, because they have strict additional constraints that are not included in our algorithm. The constraints may originate from some information loss caused by flattening down the syntax-tree or from an over-emphasis on syntactic structure.

Considering our algorithm, it is likely to happen that the set of initial clones contains function pairs whose structures are different from each other, because the selection of initial clones is metric-driven. Thus an efficient filtering system is desired to narrow down the set of initial clones.

4.3.1. *False positive and irrelevant clones.* First, we discuss the difference between *false positive* and *irrelevant clones*. False positive clones are such clones that are not real clones in fact, whilst irrelevant clones are real clones, but they are absolutely useless. Exemplars of both kinds of clones are shown in Figure 1.

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

FIGURE 1. Exemplars of false positive and irrelevant clones

Clone detection algorithms usually focus only on false positive clones during filtering, and do not deal with irrelevant clones. Our goal was to construct an algorithm that highlights only important clones, so we have tried to filter out clones that are likely to serve no useful purpose. We have observed that the complete result of clone detection can be ruined by a huge amount of irrelevant clones, because the user is not capable of distinguishing important clones from irrelevant clones while he/she is being swamped with worthless details.

Our filtering system makes up the second phase of the algorithm, which again uses arbitrators to remove both irrelevant and false positive clones. We note that this filtering system completely differs from the filtering system that

we introduced our previous algorithm [8]. The only thing common in these filtering systems is their purpose; both filter out false positive and irrelevant clones.

4.3.2. *Arbitrators.* The arbitrators introduced in the first phase have different tasks in the second phase; they have to vote on each initial clone. An arbitrator votes *true*, if the arbitrator regards the clone as an important clone, otherwise it votes *false*. The arbitrator only votes on those initial clones that did not receive an impossible vote from this arbitrator in the first phase. Thus, the conditions being examined on an initial clone depend on the poll result of the first phase. If all of the involved arbitrators vote true, then the initial clone appears in the result set of the algorithm, otherwise it is dropped.

Why are not all of the arbitrators involved in the poll? We might lose some important clones, if all of the arbitrators poll about all of the clones. To understand the reason, consider votes of Cat4 in Table 4. In this category the forming elements of an initial clone were found to be written in different programming styles, thus it is likely that the vote of the Programming style arbitrator would be negative again. However, this negative vote should be not taken into account, because this clone has been accepted into the set of initial clones, for which the vote of Programming style was disregarded.

4.3.3. *Forming votes.* The votes of the arbitrators are determined by evaluating all of the conditions of their *condition sets*. Every *condition* is an ordered pair, whose elements are a *characterising function* and a *binary predicate*. Every characterising function queries and characterises some lexical, syntactic or semantic property (etc. number of tokens) of the given function as defined below.

$$PropertyFun :: Function \mapsto Property$$

Each binary predicate assigns a boolean value to every given pair of properties. The assigned value depends on the condition in which the binary predicate is defined.

$$PredicateFun :: Property \times Property \mapsto \{true, false\}$$

Let $Conds$ denote the condition set that needs to be evaluated, and let A and B denote the elements of the examined initial clone. We then formalise how the vote of an arbitrator is determined as follows.

$$\forall C \in Conds(C.PredicateFun(C.Prop(A), C.Prop(B))) \Leftrightarrow vote = true$$

Thus, the vote is true if and only if all of the conditions evaluate to true; otherwise the vote is false.

In the following, we detail the conditions that are currently used to form the votes of arbitrators. Although any number of conditions can belong to

a condition set, each condition set has a single condition at the moment. Before we detail the conditions that belong to Flow of Control, Expression and Programming Style respectively, we detail one special condition set.

4.3.4. *Identical Condition set.* Considering Cat1 initial clones, it can be easily seen that identical or nearly identical clones belong to this category. A clone is considered to be an identical clone, if its forming functions differ from each other only in identifiers or constants. If the first phase of the algorithm has detected a real, quite complex identical clone, then this clone should still be considered as a clone even if none of the conditions used for every other clone is true. To summarise, it can be said that Cat1 clones are exceptions and are treated differently than other clones.

In this condition set two conditions exist that are called *Identical functions* and *Complexity*. The *Identical functions* condition tries to filter out those clones whose functions differ from each other in terms of something other than their identifiers and constants. To every given function, this characterising function assigns a string that is formed as follows. The syntax tree of the function is flattened down by assigning a separate character to each token type, except to those tokens that form a function call. We preserve the name of the called function to filter out those clones that refer to different functions, because these clones cannot be identical clones. The binary predicate of this condition is a simple equivalence test, it assigns true to every pair of identical strings, otherwise it assigns false.

The *Complexity* condition tries to filter out clones that have at least one element that is not complex enough. A function is not complex enough, if none of its function clauses are complex enough. A function clause is not complex enough, if the depth of its syntax-tree is small. To every given function, the characterising function assigns the maximum depth value of its function clauses. Given two depth values, denoted by A and B respectively, the binary predicate assigns true to them, only if: $A > 4 \quad \wedge \quad B > 4$

4.3.5. *Flow of Control Condition set.* In this set, the only condition is the *Ratio of same function calls*. It tries to filter out clones whose elements call mostly separate functions, because the semantics of these clones are likely to differ from each other. To each function, this characterising function assigns a set of functions referred to directly from the given function. For every initial clone the characterising function assigns two sets of functions, which are denoted by A and B, respectively. The binary predicate assigns true to the given properties only if the following statement holds:

$$\frac{|A| + |B| - 2 * |A \cap B|}{|A \cap B|} < 0.1$$

4.3.6. *Expression Condition set.* The only condition that belongs to this set is *Maximum depth of function clauses.* It does the same as Complexity condition does, except that binary predicate is stricter. Given two depth values, denoted by A and B respectively, the binary predicate assigns true to them, only if: $A > 7 \quad \wedge \quad B > 7$

4.3.7. *Programming Style Condition set.* This set has one condition, namely *Minimum number of tokens.* It gets rid of clones that have at least one short element. To every function, the characterising function assigns the sum of the number of tokens forming its function clause bodies. Given two token counts, denoted by A and B respectively, the binary predicate assigns true to them, only if: $A \geq 20 \quad \wedge \quad B \geq 20$

4.3.8. *Filtering system.* We now describe the filtering system that uses the above condition sets. The filtering system works by narrowing down the set of initial clones to produce the result set of clones by narrowing down the set of initial clones. We have defined a flexible system that is capable of handling separate categories of initial clones by utilising arbitrators again.

Each initial clone is analysed as follows based on its category. If the initial clone is a Cat1 clone, then only the Identical Condition set is evaluated on it, otherwise the arbitrators poll any clone to which the arbitrator has not given an impossible vote during the first phase. The vote of each involved arbitrator is computed by evaluating its condition set. If more than one arbitrator is involved in the poll, then their votes are summed up by taking the conjunction of their votes. If the result is true, then the clone appears in the result set of the algorithm, otherwise it is dropped.

5. Evaluation

5.1. **Notes on implementation.** The implementation of our algorithm is extremely specialised for Erlang. The whole algorithm is designed to be efficient and parallel, because every metric value of a function is calculated in a lazy and caching way, and the processing is done by separate processes that follow a work-stealing strategy. To make the application more scalable, the number of worker processes is determined at run-time, based on the attributes, such as, the number of available processors, of the computer running the program.

During the first phase of the algorithm, worker processes are responsible for controlling polls related to clone candidates. Each process gathers an unprocessed clone candidate and manages the voting among the arbitrators. Each vote is determined by computing only the necessary amount of software metric values. The arbitrators, which have been instantiated either in the same process or in separate processes, share the same cache tables, so the

caching level is maximised to avoid calculating the same metric value twice. After all of the arbitrators have voted, two cases can happen: the candidate can become an initial clone or can be ignored during the second phase of the algorithm.

During the second phase of the algorithm, worker processes control polls related to unprocessed initial clones. The result of the poll can be determined by evaluating a logical conjunction chain, which is nested when more than one arbitrator is involved in the poll. Based on this observation, every conjunction chain can be computed by short-circuit evaluation. Thus, no pointless operation is performed.

5.2. **Evaluation.** Our algorithm performed well on several open-source projects, including Mnesia [14], Dialyzer [13] and Wrangler [11]. The well-known Bellon's benchmark [4] deals with only imperative languages and does not offer any test projects written in Erlang. Unfortunately, we have not found any alternatives specialised for Erlang, thus we had to analyse the detected clones manually. All the clones we found in the result set are considered to be real clones by us.

We have observed that the result of the algorithm is quite hard to comprehend, if a function has several occurrences. Crucially, if a function has N occurrences then $\frac{N*(N-1)}{2}$ pairs are reported by the algorithm. In the case of Mnesia, the 13 detected cloned occurrences of mnesia:val/1 function were reported as 78 pairs.

We have noticed that the new implementations of previously existing features are likely to be created by copy&pase programming (e.g: `refac_clone_evolution` and `refac_inc_sim_code` modules in Wrangler, or `dialyzer_gui` and `dialyzer_gui_wx` modules in Dialyzer). Some properties of the test runs are shown in Table 5.

Project	Line of code	Func.s (pcs)	Analysed pairs (pcs)	Initial clones (pcs)	Clones (pcs)	Execution time (sec)
Dialyzer	17 292	1 481	520 710	20 660	24	65
Mnesia	22 594	1 687	1 418 770	51 773	123	141
Wrangler	51 274	4 037	5 887 596	178 286	466	1 047

TABLE 5. Properties of test runs

Some interesting clones (found in Mnesia, Wrangler and Dialyzer respectively) are shown below.

```
First instance (found in mnesia):
%% Local function in order to avoid external function call
```

```
val(Var) ->
    case ?catch_val(Var) of
        {'EXIT', Reason} -> mnesia_lib:other_val(Var, Reason);
        Value -> Value
    end.

Second instance (found in mnesia_index):
val(Var) ->
    case ?catch_val(Var) of
        {'EXIT', _ReASoN_} -> mnesia_lib:other_val(Var, _ReASoN_);
        _VaLuE_ -> _VaLuE_
    end.

Category: 2

First instance (found in api_refac):
exported_funs(File) ->
    {ok, {_, Info}} = wrangler_ast_server:parse_annotate_file(File, true),
    case lists:keysearch(exports, 1, Info) of
        {value, {exports, Funs}} ->
            Funs;
        false ->
            []
    end.

Second instance (found in api_refac):
defined_funs(File) ->
    {ok, {_, Info}} = wrangler_ast_server:parse_annotate_file(File, true),
    case lists:keysearch(functions, 1, Info) of
        {value, {functions, Funs}} ->
            Funs;
        false ->
            []
    end.

Category: 1

First instance (found in dialyzer_utils):
keep_endian(Flags) ->
  [cerl:c_atom(X) || X <- Flags, (X =:= little) or (X =:= native)].

Second instance (found in dialyzer_utils):
keep_all(Flags) ->
  [cerl:c_atom(X) || X <- Flags,
                     (X =:= little) or (X =:= native) or (X =:= signed)].

Category: 4
```

By comparing the presented algorithm with Clone IdentifiErl, we have observed that the main difference between the presented algorithm and Clone

IdentifiErl originates from the chosen unit. Clone IdentifiErl works with top-level expressions as units, thus it can detect such clones whose elements form functions only partially. However, working with top-level expressions has a deficiency: the problem space of Clone IdentifiErl is larger by orders of magnitude.

6. Conclusion and Future work

In this paper we have described a metric based clone detection algorithm for Erlang. In the presented methods we have utilised the source code representation and the already implemented metrics of RefactorErl. Our algorithm is efficient and parallel, and introduces a filtering system to eliminate both false positive and irrelevant clones. We have evaluated our algorithm on the source code of different open source projects, and compared the algorithm with Clone IdentifiErl.

We want to further evaluate and improve our techniques. At first, we want to work out an algorithm which clones can be grouped with to make the result more comprehensive as this need has been shown above. On the other hand we are going to further study the results of our analysis and tune the algorithm by altering the number of used metrics and the parameters.

Acknowledgement

This work has been supported by Ericsson–ELTE-Soft–ELTE Software Technology Lab. The authors would like to thank Julia Lawall for her useful advices.

References

[1] RefactorErl Homepage. http://plc.inf.elte.hu/erlang.

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

[3] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 368–377, 1998.

[4] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *Software Engineering, IEEE Transactions on*, 33(9):577–591, 2007.

[5] I. Bozó, D. Horpácsi, Z. Horváth, R. Kitlei, J. Kőszegi, 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, October 2011.

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

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

[8] Viktória Fördős and Melinda Tóth. Identifying Code Clones with RefactorErl. In *Proceedings of the 13th Symposium on Programming Languages and Software Tools, ISBN 978-963-306-228-9*, pages 31–45, Szeged, Hungary, August 2013.

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

[10] R. Király and R. Kitlei. Application of complexity metrics in functional languages. In *Proceedings of 8th Joint Conference on Mathematics and Computer Science, ISBN 978-963-9056-38-1*, pages 267–282, Komrno, Slovakia, July 2010.

[11] 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, New York, NY, USA, 2009. ACM.

[12] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: a tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 20–20, Berkeley, CA, USA, 2004. USENIX Association.

[13] Tobias Lindahl and Konstantinos Sagonas. Practical type inference based on success typings. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 167–178, New York, NY, USA, 2006. ACM Press.

[14] Haakan Mattsson, Hans Nilsson, and Claes Wikstrom. 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.

[15] J. Mayrand, C. Leblanc, and E.M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Software Maintenance 1996, Proceedings., International Conference on*, pages 244–253, 1996.

[16] Susan H. Randy Smith. Detecting and Measuring Similarity in Code Clones. *IWSC*, 2009.

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

[18] Saul Schleimer, Daniel S. Wilkerson, and Alex 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.

[19] Melinda Tóth and István Bozó. 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.

ELTE-Soft Nonprofit Ltd & Department of Programming Languages and Compilers, Faculty of Informatics, Eötvös Loránd University

E-mail address: [f-viktoria,tothmelinda]@elte.hu