

# Tdk dolgozat

**Viktória Fördős**



EÖTVÖS LORÁND UNIVERSITY  
FACULTY OF INFORMATICS  
DEPARTMENT OF PROGRAMMING  
LANGUAGES AND COMPILERS

---

# Detecting code duplicates with RefactorErl <sup>1</sup>

*Supervisor:*

**Melinda Tóth**  
Assistant lecturer  
ELTE-IK, PNYF

*Author:*

**Viktória Fördős**  
Software Information Technology MSc,  
Specialization in Software Technology  
night school, 2nd class

Budapest, 2013.

---

<sup>1</sup>Supported by Ericsson-ELTE-Soft-ELTE Software Technology Lab &  
TÁMOP-4.2.2/B-10/1-2010-0030

## **Abstract**

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 life-cycle.

Most theories have already been converted to software tools, called duplicated code detectors, but these solutions are based on methods and algorithms, which have been constructed specifically for the imperative paradigm and its mainstream languages, whilst in functional programming only a few exist. If the Erlang programming language is taken into consideration, only one usable solution is present, which can be used on legacy code. Therefore my goal is to elaborate the missing approaches for Erlang to fade away the lack of different duplicated code detectors.

In my thesis, I review the topics related to code clones, and I present my general algorithm that was constructed by abstracting from existing approaches, with which I highlight the common parts of different theories. Besides studying the proposed techniques and trying to adapt them to Erlang, I present a new algorithm for accurate clone detection, which has been successfully evaluated on open-source projects. Furthermore, I try to adapt an existing approach to the characteristics of Erlang and I also present a sketch of an algorithm that may efficiently be used to detect plagiarism. Hereupon, I also discuss and compare the constructed approaches in detail.

# Contents

|                                                          |           |
|----------------------------------------------------------|-----------|
| <b>Introduction</b>                                      | <b>4</b>  |
| <b>1 Clone detection</b>                                 | <b>7</b>  |
| 1.1 Occurrence of clones . . . . .                       | 7         |
| 1.2 Duplicated code detectors . . . . .                  | 8         |
| 1.2.1 Applications . . . . .                             | 8         |
| 1.3 Related principles of software development . . . . . | 9         |
| 1.3.1 DRY . . . . .                                      | 9         |
| 1.3.2 Rule of three . . . . .                            | 9         |
| 1.4 The general algorithm . . . . .                      | 10        |
| 1.4.1 Producing clone candidates . . . . .               | 10        |
| 1.4.2 Determining initial clones . . . . .               | 12        |
| 1.4.3 Filtering and trimming initial clones . . . . .    | 13        |
| 1.5 Existing approaches . . . . .                        | 14        |
| 1.5.1 Producing clone candidates . . . . .               | 14        |
| 1.5.2 Determining initial clones . . . . .               | 17        |
| 1.5.3 Filtering and trimming initial clones . . . . .    | 19        |
| 1.6 Moss . . . . .                                       | 19        |
| 1.6.1 Winnowing algorithm . . . . .                      | 20        |
| <b>2 Erlang &amp; RefactorErl</b>                        | <b>22</b> |
| 2.1 Erlang . . . . .                                     | 22        |
| 2.2 RefactorErl . . . . .                                | 24        |

|          |                                                            |           |
|----------|------------------------------------------------------------|-----------|
| <b>3</b> | <b>Algorithms</b>                                          | <b>26</b> |
| 3.1      | AST/metric based approach . . . . .                        | 26        |
| 3.1.1    | Unit . . . . .                                             | 27        |
| 3.1.2    | Matrix . . . . .                                           | 29        |
| 3.1.3    | Determining initial clones . . . . .                       | 33        |
| 3.1.4    | Providing only accurate results . . . . .                  | 39        |
| 3.1.5    | Filtering and trimming unit . . . . .                      | 41        |
| 3.2      | Metric based approach . . . . .                            | 49        |
| 3.2.1    | Reviewing the original solution . . . . .                  | 49        |
| 3.2.2    | Adapting to Erlang . . . . .                               | 53        |
| 3.3      | Control-flow based approach . . . . .                      | 57        |
| <b>4</b> | <b>Realisations</b>                                        | <b>59</b> |
| 4.1      | Solutions based on the AST/metric based approach . . . . . | 59        |
| 4.1.1    | Producing clone candidates . . . . .                       | 59        |
| 4.1.2    | Determining initial clones . . . . .                       | 61        |
| 4.1.3    | Filtering and trimming initial clones . . . . .            | 65        |
| 4.2      | Realising the metric based approach . . . . .              | 66        |
| 4.2.1    | Producing clone candidates . . . . .                       | 66        |
| 4.2.2    | Determining the cloning level . . . . .                    | 67        |
| 4.3      | Optimisations . . . . .                                    | 68        |
| 4.4      | Utilities . . . . .                                        | 69        |
| 4.4.1    | General unit generator . . . . .                           | 69        |
| 4.4.2    | General caching mechanism . . . . .                        | 71        |
| <b>5</b> | <b>Measurements &amp; evaluations</b>                      | <b>73</b> |
| 5.1      | Measurements . . . . .                                     | 73        |
| 5.1.1    | Configurations . . . . .                                   | 73        |
| 5.1.2    | Measurement methodology . . . . .                          | 74        |
| 5.1.3    | Mnesia - Subject of the detectors . . . . .                | 74        |
| 5.1.4    | Back-ends of RefactorErl . . . . .                         | 74        |
| 5.1.5    | Variants of the AST/metric based approach . . . . .        | 75        |
| 5.1.6    | Assessment of the measurements . . . . .                   | 78        |

|          |                                                                  |            |
|----------|------------------------------------------------------------------|------------|
| 5.2      | Evaluations . . . . .                                            | 79         |
| 5.2.1    | AST/metric based approach . . . . .                              | 79         |
| 5.2.2    | Metric based approach . . . . .                                  | 80         |
| <b>6</b> | <b>Comparison with Wrangler</b>                                  | <b>84</b>  |
| 6.1      | The clone detector of Wrangler . . . . .                         | 84         |
| 6.2      | Method of comparison . . . . .                                   | 85         |
| 6.3      | Assessment of the results . . . . .                              | 86         |
| <b>7</b> | <b>Enhancement possibilities</b>                                 | <b>88</b>  |
| 7.1      | Decreasing the cardinality of initial clones . . . . .           | 88         |
| 7.2      | Replacing the technique of the initial clones detector . . . . . | 89         |
| 7.3      | Self-organising general caching mechanism . . . . .              | 90         |
| 7.4      | Exploiting GPU instead of CPU . . . . .                          | 90         |
|          | <b>Conclusion</b>                                                | <b>92</b>  |
| <b>A</b> | <b>Used configurations</b>                                       | <b>94</b>  |
| A.1      | Config A . . . . .                                               | 94         |
| A.2      | Config B . . . . .                                               | 95         |
| <b>B</b> | <b>Some interesting detected clones</b>                          | <b>96</b>  |
|          | <b>List of algorithms</b>                                        | <b>100</b> |
|          | <b>Bibliography</b>                                              | <b>105</b> |

# Introduction

Duplicated codes or clones [38] are fragments of source code within a software which are similar or identical to each other. The occurrence of clones indicates bad design or development methodology, it greatly reduces the code quality and increases the risk of errors, hence the maintenance costs.

The manual localisation of clones can be a tough challenge for any development team, even in small programs, but in large-scale software it is almost an impossible task, because every part of the source code becomes a potential duplicate, and has to be compared with every other part. This results in a number of comparisons growing exponentially with the size of the source code.

Clone detectors [15, 20, 31, 37, 39] are helping with the identification of duplicates. The applications are based on different theories for the analysis of the source code, such as code tokens analysis [24], syntax tree analysis [11], or combining several metrics [34]. Not surprisingly, given the practical interest of clone detection, the majority of these methods and algorithms have been proposed specifically for mainstream programming languages, which are all imperative languages.

## Motivation

In the context of functional programming, only a very few clone detectors are available, such one [14] developed for Haskell, and [29] for Erlang.

Erlang [4, 9] is a lesser-known declarative programming language, but it is widespread used by several companies for various purposes, such as telecommunication, gambling and on-line betting. Therefore millions lines of

legacy code exist, written in Erlang, and are likely to contain a huge number of clones. Clones can result in several bugs and inconsistency during software maintenance, thus programmers should be aware of the existing clones in the software at least. My goal is to propose approaches that are capable of handling real-world code in Erlang.

In my thesis, I review several techniques proposed to point out clones, and I try to specialise one of them on the characteristics of Erlang, in order to provide more accurate results. I use the representation of Erlang programs defined by RefactorErl to construct approaches. Furthermore, I present a novel algorithm that is capable of handling real-world code. I describe the parts of my algorithm in detail, and I assess various realisations of my algorithm. I evaluate both the specialised algorithm and mine on an open-source project.

## Structure and notation of the thesis

The rest of the thesis is arranged as follows. In Chapter 1, concepts related to clones are reviewed, then I introduce my own abstract, general algorithm of duplicated code detectors, finally some published techniques are summarised by using the introduced abstract algorithm. In Chapter 2, I briefly introduce the characteristic properties of the Erlang programming language and the important features of RefactorErl. Three algorithms are presented in Chapter 3, one of which is my own algorithm. The realisations of these algorithms, their optimisations and some general utilities are shown in Chapter 4. In Chapter 5, I evaluate the presented algorithms. In Chapter 6, the existing clone detector for Erlang is compared with my approach. I propose some optimisations to my algorithm in Chapter 7. Finally I summarise the main results of my thesis and conclude my work.

This thesis deals with an area of computer science not familiar to many readers, hence many notions are presented in the following manner, and referred for example, by D0.1. Additionally, I assume that the reader is familiar with Erlang.

**D0.1. Sample definition.** *Description of the definition.*

Algorithms presented in the thesis are described using pseudo-code according to the following.

```
if  $i \geq maxval$  then
    # This is a comment
     $i \leftarrow 0$ 
else
    if  $i + k \leq maxval$  then
         $i \leftarrow i + k$ 
    end if
end if
```

**Algorithm 0.1:** Algorithm Example

Erlang source fragments are shown in the following manner.

```
quick_sort([]) ->
    [];
quick_sort([H | T]) ->
    quick_sort([ X || X <- T, X < H ]) ++ [H]
    ++ quick_sort([ X || X <- T, X >= H ]).
```

Erlang source 1: Example of Erlang source

Code clones are shown in the following manner.

```
% First instance
f()-> this_is_the_first_instance_of_the_clone.

% Second instance
f()-> this_is_the_second_instance_of_the_clone.
```

Erlang source 2: Example clone consists of two instances

# Chapter 1

## Clone detection

In this chapter general notions and problems are presented regarding to clones. I present my own abstract algorithm to detect code clones in Section 1.4, which is used to review several published techniques in Section 1.5.

### 1.1 Occurrence of clones

**D1.1. Codebase.** *The sum of source code maintained by an organization is called codebase.*

**D1.2. Duplicated code.** *In information technology, duplicated code (also called code duplicate, duplicate, code clone or simply clone) is a fragment of a source code that occurs multiple times within the codebase.*

**D1.3. Identical clones.** *Clones whose source fragments are identical to each other are called identical clones.*

**D1.4. Similar clones.** *Clones which can be refactored into each other by generalising its source are called similar clones. Thus, identical clones are special similar clones.*

There are multiple reasons for the occurrence of clones.

Most clones are the results of “copy&paste programming”. Programmers easily make the mistake of copying and adapting existing implementations

of algorithms instead of generalising them and providing thus a general algorithm with specialisations for the new task and the existing ones.

Duplicates can also be the result of plagiarism. Students learning computer science usually must develop many applications during exams or as homework, and usually it is expected that the software is the student's own intellectual work. However, occasionally the students do not fulfil this expectation, but rather copy the work of fellow students, and submit them as their own work, sometimes even without the knowledge of the original author. The original source is usually modified to hide from copy detection (with more or less success), and the copies have an almost identical functionality.

## 1.2 Duplicated code detectors

**D1.5. Duplicated code detector.** *A duplicated code detector (also called duplicates detector, clone identifier, clone detector, or duplicated code detection software) is an application for detecting duplicated code in the codebase.*

There are several duplicated code detectors with different functionalities. The programming language and the programming paradigm of the source code, the properties of the duplicates, the implemented matching algorithm may result in differences in the result of the duplicated code detection.

### 1.2.1 Applications

**D1.6. Anti-unification.** *The generalised source fragment of a group of clones, with which the clones can be refactored into each other, is called anti-unification.*

**D1.7. Duplicate elimination.** *Duplicate elimination is the manual or automated process where by clones are replaced by instances of their anti-unifications, leaving the behaviour invariant.*

A duplicated code detector can be applied in many areas, including plagiarism detection. They are also excellent in finding duplicates in legacy source

code acquired from different organisations, providing better code quality, smaller file sizes and better maintenance. Indeed, by eliminating duplicates the possibility of errors is also reduced, since all maintenance operation can be performed on a single code fragment.

By using duplicated code detectors, the creation of duplicates can also be avoided. Let us assume that the revision control system used by the codebase supports creation of new versions with preconditions. Setting a precondition on the repository to be duplication-free eliminates the possibility of uploading clones, hence the degradation of code quality. (The precondition can be verified by performing analysis on the revision.)

We can also choose to retain the duplicates, but in this case, particular attention should be paid to apply changes either to all or no occurrence of the duplicate. This can also be monitored using duplicate detection software.

## 1.3 Related principles of software development

### 1.3.1 DRY

According to the *DRY* (*Do not Repeat Yourself*) principle [22], representation of all pieces of a knowledge must be unique and unambiguous inside the codebase. If the DRY principle is followed, modifications of the codebase can be applied easily, quickly and safely, because every modification must only be applied in one location except some cases (e.g.: adding a constructor to a data type). If the codebase contains clones, the foundations of the principle have been breached.

### 1.3.2 Rule of three

The *Rule of Three* principle was introduced by Martin Fowler in his widely-known book, titled Refactoring [32]. It is more permissive than the DRY principle by allowing single duplicates in the code. As a refactoring guideline, it suggests that the duplicated code segment must only be extracted to a subroutine when used for the third time. This means that finding clones in

the codebase does not necessarily mean the violation of the Rule of Three principle.

## 1.4 The general algorithm

Several duplicated code detection techniques [38] were published in the past decades. By unifying those algorithms, an abstract, general algorithm can be constructed, which highlights the common parts of the different methods, as shown in Algorithm 1.1.

It relies on three functions:

- `CandidatesGenerator` produces the set of clone candidates,
- `InitialClonesDetector` creates the set of potential clones,
- `FilteringAndTrimmingUnit` forms the resulting clone set.

|                                                                                                                                                                                                                                                                                                                    |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> <b>function</b> DUPLICATEDCODEDETECTOR(<i>Sources</i>)   <i>Candidates</i> ← CANDIDATESGENERATOR(<i>Sources</i>)   <i>InitialClones</i> ← INITIALCLONESDETECTOR(<i>Candidates</i>)   <i>Clones</i> ← FILTERINGANDTRIMMINGUNIT(<i>InitialClones</i>)   <b>return</b> <i>Clones</i> <b>end function</b> </pre> |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

**Algorithm 1.1:** General duplicated code detector

It is noteworthy that the algorithm is the composition of three functions, hence it cannot be easily parallelised.

### 1.4.1 Producing clone candidates

**D1.8. Clone candidate.** *A clone candidate is a set of possible duplicate occurrences derived from source code fragments that have been transformed to the inner data representation of the matching algorithm, from which the algorithm produces the initial clones D1.9.*

Duplicated code detecting algorithms usually do not operate directly on the source code, but rather on an abstract inner representation. The

**CandidatesGenerator** function is responsible for the creation of the abstract representation and the clone candidates. The signature of the function is the following:

$$CandidatesGenerator : Sources \rightarrow Candidates,$$

where **Sources** is a set of **Source**, where **Source** represents the content of a source code, as a sequence of **Tokens**. The string of **Token** represents a fragment of a source code. The described type is shown structurally below:

$$Sources = set(Source) = set(seq(Token))$$

**CandidatesGenerator** assigns the following computable value to each **S**:

$$CandidatesGenerator(S) = Combine(Transform(S)), S \subseteq Sources$$

where **Transform** is a function that is responsible for transforming the original source code to the transformed inner representation;

$$Transform : Sources \rightarrow TransformedSources$$

The type of the inner representation is **TransformedSources**, which is a power set, called **TransformedSource**, whose elements are called **Unit**. **Unit** is a sequence, which is built up from transformed **Token**, called **TransformedToken**. The **TransformedSources** type is described structurally below:

$$\begin{aligned} TransformedSources &= set(TransformedSource) = set(set(Unit)) \\ &= set(set(seq(TransformedToken))) \end{aligned}$$

**Combine** is responsible for composing the transformed source to form the clone candidates.

$$Combine : TransformedSources \rightarrow Candidates$$

**Candidates** is a set of **Candidate**, where each **Candidate** is a set, whose elements represent the instances of a clone. The type of **Candidates** is described structurally below:

$$Candidates = set(Candidate) = set(set(seq(Unit)))$$

## 1.4.2 Determining initial clones

**D1.9. Initial clone.** *The initial clones are the subset of clone candidates (D1.8) which is derived using predefined rulesets, procedures and heuristics by the **InitialClonesDetector** operation.*

Usually the set of clone candidates is too large to be amenable to detailed analysis, therefore algorithms try to narrow down the set for example, by applying predefined rules and heuristics of the used technique. The **InitialClonesDetector** operation is responsible for this, and has the following signature:

$$InitialClonesDetector : Candidates \rightarrow Candidates$$

The operation results in a set, denoted by **InitialClones**, which is a subset of **Candidates**, provided by the **CandidatesGenerator**. This result is the subject of further analysis. However, in practice the number of elements in the initial clone set is still too large, and therefore unusable. Further selection may be applied using different technique.

$$\begin{aligned} InitialClonesDetector(Cands) &= InitialClones \\ Cands, InitialClones &\in Candidates \\ InitialClones &\subseteq Cands \end{aligned}$$

### 1.4.3 Filtering and trimming initial clones

**D1.10. Filter evaluation.** *The evaluation of a filter is the production of the solution set on the set of the initial clones:*

$$filter : Candidates \rightarrow \mathbb{L}, [filter] ::= \{c | c \in Candidates, filter(c) = true\}.$$

The `FilteringAndTrimmingUnit` component is responsible for further narrowing down and transforming the initial clones, producing the final result of the algorithm. This component is optional, if the set of the initial clones is narrow enough, then the ordering rule of the `FilteringAndTrimmingUnit` should be the identity function. A solution is shown in Section 3.2, which consists of two components.

Usually clone detectors are built up from different techniques to strengthen their efficiency and to refine their results. In practice, clone detectors utilise separate techniques for the second and the third component.

$$FilteringAndTrimmingUnit : Candidates \rightarrow Candidates$$

Evaluating filters (D1.10) on the set of initial clones provides a subset on which trimming is performed to split large clones into multiple instances according to predefined rules as follows:

$$FilteringAndTrimmingUnit = FilteringUnit \circ TrimmingUnit$$

The trimming unit has a predefined, finite set of trimming functions, as mentioned above. Each trimming function is responsible for examining clones from the aspect of its rule. Such rule can be for example, that an instance of a duplicate must occur in the body of a function. Whether an instance violates this rule, the instance and all its occurrences has to be split into two distinct set along `Units`.

Another example is presented in Section 3.1, where the applied techniques

of the first two components are very abstract, so the trimming unit is essential to narrow down the scope of the duplicate instances along **Units**.

**FilteringAndTrimmingUnit** results in a set, in which every element contains the occurrences of a single clone. This is the result of the algorithm.

## 1.5 Existing approaches

The techniques used by a duplicated code detection software are the concrete realisation of the components presented in the previous section. The techniques for each component can be chosen independently from each other and varied as long as their composition is well-typed. In this section, I show techniques that can be used as the realisation of each component.

### 1.5.1 Producing clone candidates

**D1.11. Unit of an instance.** *The units of an instance (of a duplicate) are code fragments. The unit means the same as type **Unit** represents, which is described in Section 1.4.1.*

The technique for choosing clone candidates impacts significantly the result of the algorithm. In practice, *units of an instance* (D1.11) can be either a line in the source code, a lexical or syntactical unit, or even a subtree of a syntax tree.

The choice of the unit determines the smallest possible code segment that can be detected as a duplicate. If the unit is too large, the search becomes insensitive to small sized duplicates. If it is too small, the size of the clones increases, hence the amount of clone candidates may become unmanageable, due to a combinatorial explosion, and the majority of the detected clones become useless. For example, if function definition has been chosen as unit, then clones that are only partially form function definitions become undetectable.

After selecting the unit, both the source code transformation method (defined as **Transform** in Section 1.4.1) and the composition of the clones (defined as **Combine**) must be specified. The transformation must be carried

out in such way that the generated item can still be identified in the original source code. The composition is not necessarily defined in the algorithm as a *general composition method* exists.

**D1.12. General composition method.** *Let us take a sequence of unit (D1.11) by preserving the order among its elements and by cutting along a meaningful boundary. This results in sub-sequences that can be considered as a set of sequences. The elements of this set are then organized into pairs according to some predefined strategy. These pairs are the resulting clone candidates.*

In practice, a sub-sequence can be matched to a syntactic structure (e.g.: function, token). The following techniques can be utilised to construct the first component of a clone detector.

### Line-based detection

The simplest approach for clone detection is *line-based detection*. It occurs when the selected unit is the line of the source code. The content of the line, represented as string, should be hashed, as working with hash values is more efficient than working with the original strings. For the production of clone candidates, the general composition (D1.12) can be used.

However, this method is rarely used, because it is not efficient enough and can find such clones that are not form a syntactic structure, therefore I will not discuss it here.

### Token-based detection

**D1.13. Suffix tree.** *A suffix tree is a data structure that represents the suffixes of a given string.*

**D1.14. Generalised suffix tree.** *A generalised suffix tree (also called extended suffix tree) is a data structure that represents the suffixes of a given set of strings.*

Token-based clone detection uses lexical units of the source code as base units. Tokens are transformed according to their characteristics over an ab-

stract alphabet. The transformation function is not required to be injective, but the length of each word of the alphabet should be minimised. If the transformation is based on the type of the lexical unit, the code fragments created by simple renaming automatically become identical.

After transformation, the general method (D1.12) can be used for composition, but there is also a different solution. An extended suffix tree (D1.14) can be constructed from transformed tokens. This construction results in that the suffix tree becomes the set of clone candidates, where all the occurrences of each duplicate can be gathered as a sub-suffix tree [21].

This technique is used by Wrangler [29] and the previous, unstable, unfinished prototype within RefactorErl.

## Syntax based detection

**D1.15. Fingerprint.** *A fingerprint is a (nearly) unique generated bit sequence that is significantly smaller than the original value. It is generated by a fingerprinting function, which does not need to be injective.*

**D1.16. Sequence database.** *The sequence database is a special collection. Contrary to regular database systems, where queries return exact results, the queries in sequence databases are usually based on probability measures and return sequences that are similar to the given sequence.*

Syntax based detectors usually come in two variants: *Block-based methods* use program blocks as the unit of the instance, whilst *function-based methods* use function bodies. The source code is partitioned and transformed according to the chosen unit. Usually the transformation results in a *hash value* or *fingerprint* (D1.15). Even a sequence database (D1.16) can be constructed from the transformed units.

In the latter approach further composition is usually not required, since it is quite rare for duplicates to reach beyond the scope of a function. Composition would only increase the amount of candidates, without any meaningful difference in the result. Another possibility is to pair each unit with every other. If the unit of clone candidates is a smaller syntactical unit, the general composition (D1.12) may also be used.

The algorithm described in [36] operates on the blocks created from the sequence of statements, and uses fingerprints and sequence database to detect clones.

### Detection based on the abstract syntax tree

The initialising step of AST based detectors is the construction of the abstract syntax tree. A sub tree representing a syntactical structure can be chosen as a unit. If no transformation is performed, every possible matching of the units will produce the set of clone candidates. This case corresponds to the general *subgraph isomorphism problem* [17], which is NP-complete and not specified in this thesis.

Another possibility is the transformation of the sub-trees into simple values, thereby flattening down the syntax tree, and using the general composition to produce the candidates. In case the syntactical structure chosen for the units is too large, it is suggested to pair every unit with every other to form the clone candidates. The constructed abstract syntax tree should also be preserved, as the information inside it may be used by the detection algorithm to provide more precise results.

This technique is used by [34] having functions as base units, and is further addressed in Section 3.2.

### 1.5.2 Determining initial clones

**D1.17. Threshold.** *A threshold holds the lower border of the similarity. If the similarity of a candidate is greater than the threshold, then the candidate is acceptable during the evaluation of candidates.*

The initial clone production component of the general algorithm can be seen as a set of boolean functions aiming to create a subset of clone candidates. The function must in all circumstances be evaluated on pairs of the clone candidates. The transformation method selected in the previous component limits the scope of the functions that can be chosen for this purpose.

In case the transformation used a suffix tree or a sequence database,

the initial clone generator is naturally a search algorithm working on the data structure introduced by the transformation. In the case of a suffix tree, the algorithm uses the properties of the built data structure, described in Section 1.5.1. In the case of a sequence database, there are multiple choices, including the usage of the probability queries mentioned in D1.16, and other data mining methods specialized for sequential database, as described in [31].

In case the transformation results in binary or text values (e.g., hash-code, fingerprint, word above an alphabet), the simplest possible matching function is the equality test. However, equality is usually too strict operation for initial clone detection. Rather, it is better to use *string similarity metrics*, for example, *Levenshtein* [28] and *Dice-Sørensen* [16, 40]. These metrics express how two strings look alike. Metrics depend on a *threshold value* (see D1.17) and are addressed in detail in Section 3.1.3.

The signature of a general boolean function operates on string pairs is:

$$f : String \times String \rightarrow \mathbb{L}$$

The metric is the  $m$  function, and **Threshold** contains a previously defined non-negative real number less than one.

$$m : String \times String \rightarrow [0, 1] \in \mathbb{R}$$

The truth set of this function is:

$$\llbracket f \rrbracket ::= \{(a, b) | a \in String, b \in String, m(a, b) > Threshold\}$$

In case the abstract syntax tree has been constructed during the transformation, metrics describing the source code represented by the syntax tree can be used. The metrics may characterise the structural, syntactic or semantic properties, complexity, even some properties of the control-flow. As with string metrics, a threshold must be defined for each metric separately.

### 1.5.3 Filtering and trimming initial clones

The set of applicable trimming and filtering algorithms is influenced by the first two components of the clone detector.

In case the transformation results in binary or text values (e.g., hash-code, fingerprint, word above an alphabet), the set of initial clones may contain many false positive matches, due, for example to a non-injective transformation. Therefore during filtering all decisions according to the initial clones must be made by considering their original form and environment.

If the initial clones have been produced based on non-specific source code metrics, it is almost certain that trimming transformations are almost certainly needed. A frequent trimming condition is when an occurrence of a clone overlaps a syntactical structure. For example, a duplicate is found that covers the definition of two functions positioned one above the other in the same source file. This clone should be cut along the function definition limiters.

In case the function based approach is used with source code specific metrics, the third component of the algorithm can be skipped, replacing the `FilteringAndTrimmingUnit` function with the identity function.

## 1.6 Moss

The clone detector Moss [39] has been developed at Stanford University since 1994. Its main application area is the detection of program plagiarism by students. The developers claim that demonstrates the efficiency and accuracy of the system:

The algorithm behind moss is a significant improvement over other cheating detection algorithms (at least, over those known to us). [6]

The efficiency, the effectiveness and the speed of Moss are mostly due to the production of document fingerprints. It was found that the fingerprint should include not only the properties derived from the document, but also properties of the fingerprint itself, including the document and the position

from which the fingerprint originates. This kind of fingerprint is known as *local fingerprint*. For the creation of local fingerprints, the Winnowing algorithm was created.

### 1.6.1 Winnowing algorithm

The input of the algorithm is a document, from which fingerprint was created, and two additional parameters: the *guarantee threshold* value and the *noise threshold* value. The output is a list of hashes with position information.

The guarantee threshold ( $t$ ) contains a length value, above which all clones are guaranteed to be in the results. The noise threshold ( $k$ ) is the minimal length, below which no clones are collected into the result. With these parameters the size of the window of the algorithm ( $w$ ) can be calculated.

$$w = t - k + 1$$

The steps of the algorithm are broadly the following.

1. Take the content of the document, from which fingerprint is generated.
2. Remove all unwanted elements from the text, for example, whitespaces.
3. Produce a sequence of  $(k + 1)$ -grams of the cleaned text.
4. Compute numeric hash values for every element of the sequence. After the computation has finished, select those hash values which are divisible by a fixed  $p$  value. The order of the sequence must also be preserved.
5. Process the selected hash values with a  $w$  sized window by choosing the item whose value is less than every other values occurring in the same window and then slide the window by a unit. Append the chosen item to the end of a list which contains already chosen items with its values, only if the value of the currently chosen item is not equal with the value of the last list item, otherwise drop the chosen item.

This selection method is described in detail in [39]. The list of items with minimum values is the fingerprint of the document. Every element of the list contains the hash value and the hash position indexed from zero.

## Chapter 2

# Erlang & RefactorErl

In this chapter, I briefly discuss the the characteristic properties of the Erlang programming language. Hereupon, the important features of RefactorErl are highlighted.

### 2.1 Erlang

Erlang is a declarative, dynamically typed, functional programming language which was designed to develop soft real-time applications. The language offers built-in support for distribution and concurrency based on asynchronous *message passing*, and a synchronous *message receiving* construction.

The compilation unit of Erlang programs is called a *module*, which is built up from *attribute forms* and *function definition forms* that are ended by the `.` token. Only the function definition forms are important for the presented algorithms, so only those are detailed in the following.

The encapsulating module, the name of the function, and the arity of the function can identify uniquely a function in Erlang. As in other functional languages, *pattern matching features* are prominent way to define functions by case. The cases of a function definition are called *function clauses*, and they are separated from each other by the `;` token. If a function is called, then the control passes to the first function clause of its definition: matching the arguments and satisfying its guards. A one-arity function which consists

of four function clauses is shown in Erlang source 3.

```
fibonacci(List) when is_list(List) ->
    [fibonacci(ListItem) || ListItem <- List ];
fibonacci(0) ->
    0;
fibonacci(1) ->
    1;
fibonacci(N) when is_integer(N) andalso N>1 ->
    DummyVar = 1,
    fibonacci(N-1) + fibonacci(N-2) .
```

Erlang source 3: fibonacci/1 function definition form

A function clause is built up from either one expression, called *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, expressions have a value.

Other branching expressions, such as **case**, **if**, **receive** and **try** expression, are also built up from clauses. For these clauses the properties which are defined for function clauses analogously hold.

Two kinds of expressions, namely the *list comprehension* and the *record* expression, can be found in the implementation of almost all industrial applications, thus my algorithm focuses on them, so I briefly introduce these expressions here.

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

Erlang source 4: A general list comprehension

*List* is a frequently used data structure in Erlang, therefore Erlang helps programmers handle lists easily. A list comprehension is a built-in language feature of Erlang to manipulate a list, based on Zermelo-Fraenkel set theory [19]. A general list comprehension is shown in Erlang source 4, whose *generators* (**Gen1**, ... **GenN**) are responsible for producing the base set. The *filters* of the list comprehension, namely **Filter1**, ... **FilterN**, narrow the

base sets. **Expr**, called as 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* which can also be classified as follows:

- 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.

## 2.2 RefactorErl

Clone detection is a special static analysis and its precision is hugely influenced by the available information, therefore most duplicated code detection software do not work directly on the source code, but rather on an abstract representation derived from it. Such representations include the series of tokens and the abstract syntax tree, from which crucial information for the analysis can be retrieved faster and more efficiently. RefactorErl [7, 13, 43] is a static source code analyser and transformation tool for Erlang, that provides a representation that contains even more information about the source beyond that of the abstract syntax tree.

The main aim of RefactorErl is to support the daily work of Erlang programmers with both 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 also facilitates code reorganisation with clustering algorithms and several refactoring methods. The incremental and asynchronous analyser architecture allows the programmer to track source code changes with limited resources. RefactorErl offers multiple user interfaces to choose from. It is possible to access a web-based interface, use Emacs or Vim with RefactorErl plugins, or an interactive console.

Besides its several static analysis features, RefactorErl provides source code metrics to check coding conventions and to point out bad smells in the source code. It has an automatic analyser mode to check the values of certain metrics during software development and maintenance and report warnings when some of them violate the predefined rules [25].

RefactorErl provides a user level Semantic Query Language [42] to gather information about the source code. Metrics are built into this query language, thus users can easily check the values of certain metrics for a given language element. For example, you can check the modules violating the design rule: “A module should not contain functions with more than forty lines of code!” by evaluating the following query: `mods[.funcs[loc > 40]]`.

The source code has to be loaded into RefactorErl in order to be analysed. The code is first transformed into token series using whitespace- and layout- preserving lexical analysis, and is then passed on to the RefactorErl preprocessor. The preprocessor provides the standard abilities (inclusion of header files, macro expansions, record handling, conditional compilation, and line control) and also preserves the the original forms of the preprocessed elements. After preprocessing, the abstract syntax tree is constructed from the token series based on the syntactical rules of Erlang. Next, semantic analysers decorate the syntax tree with semantic attributes and links, resulting in a graph, called the *Semantic Program Graph*, which is the internal data model of RefactorErl.

The labelled vertices of the Semantic Program Graph are the lexical, syntactic and semantic units of the source code, while the directed edges between them represent the lexical, syntactic and semantic relations, for instance, from an expression to a function an edge labelled with *funlref* is directed, if the expression calls the function. Information from the Semantic Program Graph is gathered by the evaluation of path expressions and traversal of the graph. For this purpose, RefactorErl provides a complete, high-level API.

The algorithms presented in this thesis use information from the Semantic Program Graph and metrics of RefactorErl.

# Chapter 3

## Algorithms

In this chapter, I present three clone detection algorithms. All of the algorithms are specialised to Erlang and take advantage of RefactorErl.

### 3.1 AST/metric based approach

In this section, I present a new algorithm [18] for accurate clone detection. My algorithm combines a number of existing techniques, but introduces a novel filtering component, as described in Section 3.1.5. As to my current knowledge these techniques have never been used specifically in Erlang. A publication about this algorithm has been submitted to the *13th Symposium on Programming Languages and Software Tools*.

What does clone detection mean intuitively? One may try to compare every code fragment to every other. The original representation of a code is too particular, thus generalisation is needed and the generalised form of code needs to be used. The similarity of each pair of code fragments can be represented by a matrix. The first component of my algorithm produces this matrix, which is detailed in Section 3.1.2. From this matrix, the initial clones can be extracted along diagonals. This is what the second component of my algorithm does, which is described in Section 3.1.3. Irrelevant clones can be found among these clones, which are removed by evaluating filters. The process is the third component of the algorithm, which is described in

Section 3.1.5. The ideas behind each filter have been based on case studies using Mnesia [2, 33].

The components of the algorithm are the same as the components of the general algorithm shown in Section 1.4.

The components are the followings:

- **CandidatesGenerator** produces the set of clone candidates;
- **InitialClonesDetector** creates the set of potential clones;
- **FilteringAndTrimmingUnit** forms the resulting clone set.

A one-arity function which consists of two function clauses is shown in Erlang source 5. This function will be my running example through out this section.

```
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 5: clone\_fun/1 function definition form

### 3.1.1 Unit

The generalisation part of the first component of the algorithm is described in this section.

#### Choosing the unit

The unit of a clone instance (D1.11) has to be chosen as cautiously as possible. One of my 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 extremely high. Due to this abstraction, an application written in Erlang is so brief that 1 line of Erlang code can be expressed with 8 to 10 lines of C code generally. It follows that block-based algorithms cannot be used. It also follows that the size of the chosen unit should be small.

Tokens and sub-expressions are small enough to be selected. However, they are too small to be used efficiently. A function clause is not small enough, therefore the top-level expression becomes the unit of the algorithm.

### Transforming the unit

The program text of a top-level expression is considered too particular, thus generalisation is needed. A good idea is to use a formal alphabet over a formal language which can hide unneeded specialisations of the tokens. Algorithm 3.1 shall be used for generalising.

```

function TRANSFORMWITHALPHABET(TopLevelExpression)
    #  $\epsilon$  is the empty word
    Word  $\leftarrow \epsilon$ 
    for all Token  $\in$  TOKENIZER(TopLevelExpression) do
        # '.' operator expresses the concatenation between words
        Word  $\leftarrow$  Word · WORDOVERALPHABET(Token)
    end for
    return Word
end function

```

**Algorithm 3.1:** Algorithm of the alphabet

A generalised top-level expression is a sentence over the fixed alphabet that is made of the concatenation of words. Every word is produced by the function `WordOverAlphabet` 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 original characteristics of an 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, my running example will be the same as shown in Table 3.1. What we can see there, that every top-level expression got indexed and generalised.

### 3.1.2 Matrix

How do code clones occur? Usually, they are the result of “copy&paste programming”. For example, let us 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.

Usually larger clones are preferred, so we want to collect the three-unit long sequence as one clone instead of collecting two one-unit long clones. To be able to do it, modifications should be handled flexibly. My 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 its original, correct order of its elements, the diagonals of a matrix are enough to be focused on. In other words, the fragments of diagonals are completely isomorphic to the fragments of code sequences found in the code directly. I put this idea in perspective in the following subsections.

#### Introducing the matrix

Let us assume that every top-level expression is numbered (indexed) sequentially, as shown in Table 3.1. 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, satisfies the following properties:

| Index | Top-level expression                        | Generalised top-level expression |
|-------|---------------------------------------------|----------------------------------|
| i-1   | clone_fun(L) when is_list(L) ->             |                                  |
| i     | ShortVar = L,                               | A=A                              |
|       | A = 1,                                      | A=A                              |
| i+1   | B = lists:max([I    I<-lists:seq(1, 10)]),  | A=A:A([A1A∇A:A(A,A)])            |
| i+2   | (A == 1) andalso throw(badarg),             | (AfA)FA(A)                       |
| i+3   | self ! B;                                   | A!A                              |
| j-1   | clone_fun(_) ->                             |                                  |
| j     | V = f(g(42)),                               | A=A(A(A))                        |
|       | LongVariableName = V,                       | A=A                              |
| j+1   | B = lists:max([J    J<-lists:seq(V, V*2)]), | A=A:A([A1A∇A:A(A,A*A)])          |
| j+2   | X = fun(E) -> E + B end,                    | A=x(A)zA+Ae                      |
| j+3   | self ! X.                                   | A!A                              |

Table 3.1: Demonstrating the transformation part of the first component

- *Similarity* is a binary relation.
- *Similarity* is reflexive, namely all values are related to themselves.
- *Similarity* is symmetric.
- *Similarity* expresses the equivalence of two top-level expressions in a significant manner.
- The values are between 0 and 1.

If the symmetric property holds, then only the elements of 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:

$$\frac{1}{2}n^2 - n$$

The algorithm uses Dice-Sørensen metric [16, 40] for determining similarity, which does satisfy the properties of *Similarity* relation, too. I see no reason why the metric should not be replaced with other string similarity metric. If the new metric is exactly a distance, its values should be normalised into  $[0, 1]$  first.

Let Dice-Sørensen metric be portrayed by the  $m$  function

$$m : String \times String \rightarrow [0 \dots 1] \subset \mathbb{R}$$

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 exactly defined:

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

The distance of Levenstein [28] can also be selected to use by the algorithm, hence it needs to be normalized. The normalised values comes from

$$\begin{array}{c}
1 \quad i-1 \quad i \quad i+1 \quad i+2 \quad i+3 \quad n \\
\begin{array}{c} 1 \\ j-1 \\ j \\ j+1 \\ j+2 \\ j+3 \\ n \end{array} \left( \begin{array}{cccccc} \dots & \dots & \dots & \dots & \dots & \dots \\ \vdots & 0.5 & 0.5 & 0.43 & 0.46 & 0 \\ \vdots & \mathbf{1.0} & \mathbf{1.0} & 0.21 & 0 & 0 \\ \vdots & 0.19 & 0.19 & \mathbf{0.94} & 0.23 & 0 \\ \vdots & 0.17 & 0.17 & 0.22 & 0.24 & 0 \\ \vdots & 0 & 0 & 0 & 0 & \mathbf{1.0} \\ \dots & \dots & \dots & \dots & \dots & \dots \end{array} \right)
\end{array}$$

Figure 3.1: Similarities are represented by a matrix

the following equation. Let  $m$  be the value of the Levensthein distance computed for  $s_1$  and  $s_2$  which are strings and whose lengths are  $a$  and  $b$ . We have

$$\text{norm}(m) ::= \frac{m}{\max(a, b)}, \quad \text{if } m \in [\max(a, b) - \min(a, b), \max(a, b)]$$

### Example

Let us consider the following code fragments that are shown in Table 3.1 with indexes. By using Dice-Sørensen metric the matrix can be constructed, whose relevant part is shown in Figure 3.1.

### Patterns in the matrix

What we expect, that the clauses are clones of each other, except that the  $(i-1)$ -th line differs from the  $(j-1)$ -th line and the  $(i+2)$ -th line also differs greatly from  $(j+2)$ -th line. Therefore, it can be said, that 3 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  $([i, i+1], [j, j+1])$ .

What we expect, that the following pairs are related to each other according to relation *isClone*:

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

In practice, the one-unit long clone pairs are not interesting and multi-unit long clone pairs should be focused on.

Let us assume that the starting units of a  $k$ -unit long clone pair can be found on the  $a$ -th, and  $b$ -th indexes ( $k$  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 Baxter [10], 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.

What we cannot find in inter-diagonals (inside a diagonal) is the following. Let us assume that the first clause of `clone_fun/1` is the same as shown in Table 3.1, but its second clause contains one newly inserted top-level expression. The new definition of `clone_fun/1` is shown in Table 3.2.

The  $([i+3], [j+4])$  clone pair and the  $([i, i+1], [j, j+1])$  clone pair 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.

### 3.1.3 Determining initial clones

In this section, I describe a parallel, efficient algorithm for determining initial clones.

As demonstrated in a previous example with Table 3.2, 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 I need to add a new parameter, called the *invalid sequence length*. An invalid sequence length 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

| Index | Top-level expression                        | Generalised top-level expression |
|-------|---------------------------------------------|----------------------------------|
| i-1   | clone_fun(L) when is_list(L) ->             |                                  |
| i     | ShortVar = L,                               | A=A                              |
|       | 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                              |
| j-1   | clone_fun(_) ->                             |                                  |
| j     | V = f(g(42)),                               | A=A(A(A))                        |
|       | 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                              |

Table 3.2: The new definition of clone\_fun/1

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.

Now, I am able to define exactly 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 previously defined non-negative real number less than one. Let *isClone* be a general Boolean function operating on string pairs.

$$isClone : String \times String \rightarrow \mathbb{L}$$

The truth set of this function is:

$$[isClone] ::= \{(a, b) \mid a \in String, b \in String, m(a, b) > Threshold\}$$

As shown in Section 3.1.2, 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.

Let *n* be the cardinality of the top-level expressions, then the set of diagonals is the following:

$$Diagonals ::= \{((i, 1), (i + 1, 2) \dots, (n, (n - i + 1))) \mid i \in [2 \dots n]\}$$

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.

Algorithm 3.2 for calculating the initial clones is detailed below. The inputs of the algorithm are the followings:

- *N* is the cardinality of the top-level expressions,
- *T* is the value of the threshold,

- *InvSeqLength* is the maximum length of a sequence which is built-up with invalid items.

The output of the algorithm is a set of the initial clones which are produced in parallel, so a union is needed to be constructed from them by “adding” them together.

Every diagonal is calculated in parallel in Algorithm 3.2, where the local variables are independent from each other, so no interference can happen between the parallel processes.

The algorithm never enters an infinite loop; it always terminates due to the iterations which are based on items, which are pre-calculated, cannot be expanded and also the cardinality of the items decreases at each iteration.

- **SelectTlePairs** is a function returning a pair of indexed top-level expressions, whose indexes are given as input.
- **isClone** is a function which determines if the given pair can form a clone by examining whether the calculated metric is greater than the given threshold.
- **isSameClone** is a function which determines if the given pair can be appended to the actual clone (**InitialClone**). A pair can be appended to the given clone only if the top-level expressions of the resulting clone belong to the same function clause per instance. This limitation is needed, because as mentioned above, overlapping must be avoided.
- **Trim** is a function which trims the beginnings and endings of the given pair. It is needed because invalid items may occur in the forming sequence of a clone. Invalid items are only allowed in the middle of the sequence, as described above, so the beginnings and endings of a pair must be cut out.

```

function INITIALCLONESBASEONDIAGONALS( $N, T, InvSeqLength$ )
   $Diagonals \leftarrow \{ \langle (i, 1), (i + 1, 2) \dots, (N, (N - i + 1)) \rangle \mid i \in [2 \dots N] \}$ 
  parallel for all  $Diagonal \in Diagonals$  do
     $InitialClones \leftarrow \emptyset$ 
    #  $\langle \rangle$  is the empty sequence
     $InitialClone \leftarrow \langle \rangle$ 
     $InvSeqCount \leftarrow 0$ 
    for all  $Index \in Diagonal$  do
       $TlePair \leftarrow \text{SELECTTLEPAIRS}(Index)$ 
      if ISCLONE( $TlePair, T$ ) then
        if ISSAMECLONE( $InitialClone, TlePair$ ) then
          #  $\oplus$  operator express the concatenation between sequences
           $InitialClone \leftarrow InitialClone \oplus \langle TlePair \rangle$ 
           $InvSeqCount \leftarrow 0$ 
        else
           $InitialClones \leftarrow InitialClones \cup \text{TRIM}(InitialClone)$ 
           $InitialClone \leftarrow \langle TlePair \rangle$ 
           $InvSeqCount \leftarrow 0$ 
        end if
      else
        if  $InvSeqCount < InvSeqLength \wedge$ 
          ISSAMECLONE( $InitialClone, TlePair$ ) then
           $InitialClone \leftarrow InitialClone \oplus \langle TlePair \rangle$ 
           $InvSeqCount \leftarrow InvSeqCount + 1$ 
        else
           $InitialClones \leftarrow InitialClones \cup \text{TRIM}(InitialClone)$ 
           $InitialClone \leftarrow \langle \rangle$ 
           $InvSeqCount \leftarrow 0$ 
        end if
      end if
    end for
    return  $InitialClones$ 
  end parallel for
end function

```

**Algorithm 3.2:** Parallel algorithm of the initial clones detector

The main cases of the algorithm are detailed below.

- If both `isClone` and `isSameClone` returns true, the actual clone is extended with the examined pair and the cardinality of invalid items becomes zero.
- If `isClone` returns true `isSameClone` returns false, the trimmed actual clone is added to the result set, the examined pair becomes a one unit long clone and the cardinality of invalid items becomes zero.
- If `isClone` returns false and `isSameClone` returns true, the followings can happen. If the actual clone is not empty and the cardinality of invalid items is less than the allowed one, then the trimmed actual clone is extended with the examined pair and the cardinality of invalid items is increased. Otherwise the trimmed actual clone is added to the result set, the actual clone becomes empty and the cardinality of invalid items becomes zero.
- If `isClone` returns false and `isSameClone` returns false, the trimmed actual clone is added to the result set, the actual clone becomes empty and the cardinality of invalid items becomes zero.

## Example

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

```

1. % First instance
   LongVariableName = Var

   % Second instance
   ShortVar = L

2. % First instance
   A = 1,
   B = lists:max([I || I<-lists:seq(1, 10)]),
   (A == 1) andalso throw(badarg),
   self ! B}

```

```

% Second instance
LongVariableName = Var,
B = lists:max([J || J<-lists:seq(Var, Var*2)]),
X = fun(E) -> E + B end,
self ! X

3. % First instance
A = 1

% Second instance
ShortVar = L

```

### Alternative method

A more efficient way might be to exploit transitivity in calculating the elements of the matrix. In order to do so, we need to replace the string similarity metric with a transitive relation.

Let us assume that *isClone* is a binary relation between duplicates. Two items are duplicates of each other if the value of the Dice-Sørensen metric (*DC*) computed for them is greater then 0.3. The relation *isClone* is not transitive because of the intransitivity of the string similarity metric, consider the following example:

$$\begin{aligned}
DC("aabb", "bbcc") &= 0.33 > 0.3 &\implies & ("aabb", "bbcc") \in isClone, \\
DC("bbcc", "ccdd") &= 0.33 > 0.3 &\implies & ("bbcc", "ccdd") \in isClone, \\
DC("aabb", "ccdd") &= 0.0 < 0.3 &\implies & ("bbcc", "ccdd") \notin isClone.
\end{aligned}$$

#### 3.1.4 Providing only accurate results

Before I describe the final part of the AST/metric based approach, we should discuss a general problem of presenting only valuable 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 notice 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 3.2.

| False positive clone                                                                                          | Irrelevant clone                                                                                            |
|---------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------|
| <pre>% First instance f(List) -&gt;     1+length(List).  % Second instance g() -&gt;     Pid ! message.</pre> | <pre>% First instance new_cg() -&gt;     #callgraph{}.  % Second instance new_plt() -&gt;     #plt{}.</pre> |

Figure 3.2: Exemplars of false positive and irrelevant clones

Clone detection algorithms focus only on false positive clones during filtering, and do not deal with irrelevant clones usually. My goal was to construct such an algorithm which users can easily notice important clones with, so I have tried to filter out any clones serving no useful purpose. I have observed that the complete result of the algorithm 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.

Hereupon, none of the published papers proposed a standalone solution to this problem. Jurgens et al. have proposed an iterative, configurable clone detector, called ConQAT, that contains a filtering system [23]. ConQAT can remove repetitive generated code fragments and overlapping clones by iteratively reconfiguring and rerunning its initial clone detector. Contrary to ConQAT, my approach, which is detailed in Section 3.1.5, is a standalone filtering system, thus it can be plugged into an initial clone detector and necessitates neither iterative evaluation nor reproductions of initial clones to filter out irrelevant clones. Although my approach is an Erlang specific solution, that can also be tailored to efficiently deal with the typical useless clones of other programming languages.

### 3.1.5 Filtering and trimming unit

As the part of the AST/metric based approach a parallel algorithm for a new filtering system has been introduced, which is detailed in this section.

There exists 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 3.1.3, the initial clone detector of the algorithm exploits the advantages of invalid sequence length to point out such a clone, which cannot be found by regular algorithms, for instance, an algorithm 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 processed also by the filters that are relevant for one-unit long clones.

As mentioned in Section 2.2, RefactorErl provides ready-to-use source code metrics and a Semantic Program Graph which is rich in information and easy to query. Thus every filter concentrates on only one characteristic of the code, computed by using the libraries of RefactorErl. The filters are detailed in Section 3.1.5.

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 any of the requirements of the filters cannot be fulfilled, the clone is dropped. That makes my 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 3.3, which is detailed in Algorithm 3.4 by focusing only on the one-unit long clones and detailed in Algorithm 3.5 by focusing only on the multi-unit long clones.

```

function FILTERINGANDTRIMMINGUNIT(InitialClones, InvSeqLength)
  Clones  $\leftarrow \emptyset$ 
  parallel for all Clone  $\in$  InitialClones do
    if ISONEUNITLONGCLONE(Clone) then
      Clones  $\leftarrow$  FILTERSFORONELONGS(Clone)
    else
      Clones  $\leftarrow$  FILTERSFORMULTILONGS(Clone, InvSeqLength)
    end if
  end parallel for
  return Clones
end function

```

**Algorithm 3.3:** Filtering and trimming unit of the algorithm

```

function FILTERSFORONELONGS(Clone)
  if  $\wedge_{FilterFun \in FilterFuns1}$  FILTERFUN(Clone) then
    return {Clone}
  else
    return  $\{\emptyset\}$ 
  end if
end function

```

**Algorithm 3.4:** Filtering and trimming unit of the one unit long clones

The input of the algorithm is the set of the initial clones and the invalid sequence length which is built-up from invalid items. The execution can run in parallel on initial clones, which are given as input to Algorithm 3.3. The algorithm always terminates due to the number of iterations are based on the amount of the given clones. 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

```

function FILTERSFORMULTILONGS(Clone, InvSeqLength)
  Clones  $\leftarrow \emptyset$ 
  AClone  $\leftarrow \langle \rangle$ 
  InvSeqCount  $\leftarrow 0$ 
  for all UnitPair  $\in$  Clone do
    if  $\wedge_{FilterFun \in FilterFuns} FILTERFUN(UnitPair)$  then
      AClone  $\leftarrow AClone \oplus \langle UnitPair \rangle$ 
      InvSeqCount  $\leftarrow 0$ 
    else
      if InvSeqCount < InvSeqLength then
        AClone  $\leftarrow AClone \oplus \langle UnitPair \rangle$ 
        InvSeqCount  $\leftarrow InvSeqCount + 1$ 
      else
        Clones  $\leftarrow Clones \cup FURTHERTRIM(AClone)$ 
        AClone  $\leftarrow \langle \rangle$ 
        InvSeqCount  $\leftarrow 0$ 
      end if
    end if
  end for
  Clones  $\leftarrow Clones \cup FURTHERTRIM(AClone)$ 
  return Clones
end function

```

**Algorithm 3.5:** Filtering and trimming unit of the multi-unit long clones

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.

### Description of the defined filters

In this section the filters used by the algorithm are presented. The filters 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 [2, 33]. Mnesia, written in Erlang, is a database management system and belongs to the standard Erlang/OTP library. Thanks to our filtering system, all irrelevant

clones, which are shown as exemplars of filters in this section, 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

The *Simple Expression filter* is responsible for dropping a pair of top-level expressions, if 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 at least. It may seem too strict, but, in practice, nobody cares about clones like the following one shown in Erlang source 6.

```
% First instance
Res

% Second instance
false
```

Erlang source 6: An example for the Simple Expression filter

The *Simple Match filter* is responsible for dropping a pair of top-level expressions in the following cases:

- only one component of the pair is a match expression;
- both are match expressions with right-hand sides that would be dropped by the Simple Expression filter;
- both components are match expressions with such right-hand sides, whose referred functions differ from each other.

An exemplar is shown in Erlang source 7.

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. An example is shown in Erlang source 8.

```

% First instance
true = ets:foldl(Insert, true, Tab)

% Second instance
DelObjs = mnesia_lib:db_get(Tab, K)

```

Erlang source 7: An example for the Simple Match filter

```

% First instance
ReplyTo ! {self(), Done}

% Second instance
Pid ! {self(), more, Slot}

```

Erlang source 8: An example for the Simple Send filter

### 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 3.5), 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, I have introduced a new filter type, called *compound filter* (D3.1), which satisfies the previously stated requirements. 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.

**D3.1. Compound filter.** *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 *Same Right-hand Side filter* is responsible for dropping a pair of

top-level expressions in the following cases:

- only one component of the pair is a send or match expression;
- both components are send or match expressions with right-hand sides of the following kinds: atom, number, character, variable or string;
- both components are match expressions with such right-hand sides, whose referred functions differ from each other.

For instance, consider Erlang source 9.

```
% First instance
Dist2 = incr_node(Node, Dist)

% Second instance
Tab = element(1, Val)
```

Erlang source 9: An example for the Same Right-hand Side filter

The *Same Record Operation filter* - a compound filter (D3.1) - is responsible for dropping a pair of top-level expressions in the following cases:

- only one component of the pair is a record operation;
- both components are record operations which differ in the classification of the operation (either a modifier or a non-modifier);
- both components are record operations which differ in the fields of the referred record.

For example, study Erlang source 10.

```
% First instance
Tid = D#decision.tid

% Second instance
Commit0 = P#participant.commit
```

Erlang source 10: An example for the Same Record Operation filter

## Filters for any clones

Every filter belonging to this group is evaluated both on one-unit long clones and on multi-unit long 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. An example is shown in Erlang source 11.

```
% First instance
case Res of
    {'EXIT', Reason} ->
        exit(Reason);
    _ -> Res
end

% Second instance
case Res of
    {'EXIT', {aborted, What}} -> abort(What);
    {'EXIT', What} -> abort(What);
    _ -> Res
end
```

Erlang source 11: An example for the Same Cardinality filter

The *Same Function Applications 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 a branching or fun expressions which differ in the function calls considering at least one of their clauses. An example is shown in Erlang source 12.

The *Head of List Comprehensions 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 the referred functions. An example is shown in Erlang source 13.

```

% First instance
Fun = fun(S) -> lists:suffix(S, File) end

% Second instance
Fun2 = fun(Frag, A) -> mnesia:foldr(ActivityId, Opaque, Fun,
    (cont.)A, Frag, LockKind) end

```

Erlang source 12: An example for the Same Function Applications filter

```

% First instance
Dist = [{good, Node, 0} || Node <- Pool]

% Second instance
TableList = [{Tab, keep_tables} || Tab <- List]

```

Erlang source 13: An example for the Head of List Comprehensions filter

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.

## 3.2 Metric based approach

In this section an atypical approach [34] is reviewed, then its adaptation to Erlang is shown.

### 3.2.1 Reviewing the original solution

In [34] an atypical idea is demonstrated, using different source code metrics to find duplicates in mainstream imperative programming languages. This technique works with function definitions as units, comparing every function with every other.

#### Point of comparison

Every candidate is examined from four different viewpoints, which are called *point of comparison*. These views focused on:

- *name view*: What is the name of the function?
- *layout view*: How is the layout of the function organised?
- *expression view*: What kind of expressions are used and how are the expressions handled to compose the function?
- *control flow view*: Which attributes do the used control structures have?

In each view, several metrics are defined with thresholds, such as the number of function calls, or the average length of the variable names. The full list of metrics is detailed below.

#### List of original metrics

These lists are copied from [34].

##### Function Metrics for Layout

1. Number of alphanumeric characters found in the comments located in the declaration section.
2. Number of alphanumeric characters found in the comments located in the executable section.

3. Number of logical comments within a function.
4. Number of lines of code within a function. A line of code is defined as a line that is not empty.
5. Mean number of characters of all variables that are used in the function.

### **Function Metrics for Expressions**

1. Total number of call sites in the function. This metric takes into account repetitive calls to the same function.
2. Number of distinct functions which are called by a given function.
3. Arithmetic mean of the complexity of all the decisions in a function.
4. Number of declarative statements within a function.
5. Number of executable statements within a function.

### **Function Metrics for Control flow graph**

1. Number of arcs found in the control graph.
2. Number of decisions in the control graph of a function.
3. Mean span of the branches of conditional arcs. This metric is expressed in number of unit arcs.
4. Number of arc crossings in the control graph.
5. Number of backward arcs in the control graph.
6. Number of nodes in the control graph where the flow stops or returns to a calling software unit.
7. Number of nodes in the control graph.
8. Arithmetic mean of the nesting level of the control graph of a function.
9. Number of paths in the control flow.

10. Number of control statements.
11. Number of breaches of structure, based on the principles of structured programming

### Values related to the point of comparison

Three values related to a point of comparison are *eq*, *sim* and *neg*. Given two functions, denoted by  $f_1$ ,  $f_2$ , the set of the metrics, denoted by *Metrics*, related to a point of comparison, and a function, denoted by  $\Delta$ , which assigns the corresponding threshold to the given metric.

- **Equal**, denoted by *eq*:  $f_1$  and  $f_2$  are equal for a point of comparison if the following holds:

$$\forall m \in Metrics : m(f_1) = m(f_2)$$

“Two functions are equal for a point of comparison if all metrics related to that point of comparison are equal in both functions.” [34]

- **Similar**, denoted by *sim*:  $f_1$  and  $f_2$  are similar for a point of comparison, if the following holds:

$$\forall m \in Metrics : |m(f_1) - m(f_2)| \leq \Delta(m)$$

“Two functions are similar for a point of comparison if the absolute difference is less than or equal to the delta threshold defined for each metric in the point of comparison.” [34]

- **Distinct**, denoted by *neg*:  $f_1$  and  $f_2$  are distinct for a point of comparison, if the following holds:

$$\exists m \in Metrics : |m(f_1) - m(f_2)| > \Delta(m)$$

“Two functions are distinct for a point of comparison if there is at least one metric where the absolute difference is greater than the delta value.” [34]

## Ordinal scale

An ordinal scale, called *cloning level*, is defined with the values related to the point of comparisons, and has 8 steps from “Exact Copy” to “Distinct Control Flow” as shown in Figure 3.3 :

| Scale                     | Name | Layout | Expr | Control |
|---------------------------|------|--------|------|---------|
| Distinct Control Flow (8) | X    | X      | X    | neq     |
| Similar Control Flow (7)  | X    | X      | X    | sim     |
| Distinct Expression (6)   | X    | X      | neq  | eq      |
| Similar Expression (5)    | X    | X      | sim  | eq      |
| Distinct Layout (4)       | X    | neq    | eq   | eq      |
| Similar Layout (3)        | X    | sim    | eq   | eq      |
| Distinct Name (2)         | neq  | eq     | eq   | eq      |
| Exact Copy(1)             | eq   | eq     | eq   | eq      |

Figure 3.3: Ordinal scale and points of comparison ([34])

## Determining the cloning level

The cloning level can be calculated for every candidate (function pair), as shown in Algorithm 3.6.

```

function DETERMININGCLONINGLEVEL(Candidates)
  Clones  $\leftarrow \emptyset$ 
  for all Candidate  $\in$  Candidates do
    CName  $\leftarrow$  COMP(name, Candidate)
    CLay  $\leftarrow$  COMP(layout, Candidate)
    CExpr  $\leftarrow$  COMP(expr, Candidate)
    CControl  $\leftarrow$  COMP(control, Candidate)

    Level  $\leftarrow$  SCALE(CName, CLay, CExpr, CControl)
    Clones  $\leftarrow$  Clones  $\cup \{ \langle$  Candidate, Level  $\rangle \}$ 
  end for
  return Clones
end function

```

**Algorithm 3.6:** Determining the cloning level

The input of the algorithm is a set whose elements are ordered pairs. An element of the set is a candidate, that is, a function pair. Function **Comp** determines the value related to the given point of comparison for the given candidate. Function **Scale** determines the cloning level by matching the relevant row from Figure 3.3 with the given values.

The output of the algorithm is a set whose elements are ordered pairs. An element of the set is a pair whose first component is the examined candidate and the second component is the cloning level. This is the result of the duplicated code detector.

### 3.2.2 Adapting to Erlang

A specialisation of [34] is demonstrated using language features of Erlang.

#### Problems

At first blush, the specialisation seems easy, but several questions have to be addressed. A large gap exists between imperative languages and functional languages, in which almost all the problems originated.

First of all, the algorithm is designed to deal with such sources which are written in an imperative language. The paradigm behind functional languages is declarative. It can be said intuitively that the main question in imperative paradigm is “How to do?”, while in declarative paradigm it is “What to do?”.

The level of abstraction in declarative languages are much more abstract than imperative languages, thus, metrics characterise the sources in a different manner.

Finally, there is no statement in Erlang, only expressions. It follows that half of the metrics in the Layout view, half of the metrics in the Expression view, and some metrics in the Control view become useless, and cannot be replaced by anything along the same meaning.

## Changing metrics

Every group of metrics tries to characterise the code in a different sense and keeping this sense was one of my goal. The Name view has not been changed, but others have. If it deemed important, the new metric is referred by its order as shown in Section 3.2.1.

### The new metrics in the Layout view are:

- Volume of comments instead of No. 1 in the original Layout view,
- Number of comments instead of No. 2 in the original Layout view,
- Number of macros instead of No. 3 in the original Layout view.

The other metrics (No. 4 and No. 5) are unchanged in this view. The new metrics tries still to characterise the layout.

### The new metrics in the Expression view are:

- Number of variable bindings instead of No. 4 in the original Expression view.
- Number of record references.
- Number of fun expression definitions.

The “Average complexity of decisions” metric was not defined exactly in the original paper, so I have defined it as the sum of cardinalities of any clause which can be found in the analysed entity. The No. 1 and No. 5 metrics are removed, the No. 2 metric is unchanged.

### The new metrics in the Control flow view are:

- Number of message passings.
- Number of triggered exceptions.

The “Number of loops” metric is re-interpreted as the sum of the cardinality of calling the `lists` module, which is part of the standard library of Erlang and used to manipulate lists, and the cardinality of the list comprehensions. This sum is increased by one if the function is recursive. The reasons are: almost all the functions defined in the `lists` module iterate through a list, and, while a list comprehension is being evaluated, an iteration through its generators is performed.

The No. 3, No. 4, No. 10. and No. 11 metrics are removed.

### Changing deltas

As described previously, several problems occur due to the difference between the paradigms, one of the difference being the level of abstraction, which effects greatly the computed metric values. Therefore the given deltas have to be replaced.

Due to the difference of abstraction level there are cases when the replacement is less then the original value. For example, for the “Number of paths in the control flow” metric, the given delta was 100, which is an extremely high value in the context of Erlang language. Instead of the given value, 10 will be used for this delta.

Due to the difference of coding habits, there are cases when the replacement is greater then the original value. For example, for the “Number of loops” metric, the given delta was 2, which is considered too small, so the value becomes 5, because almost all the problems are solved recursively with functional programming languages.

### Dealing with units

In [34] article the function entity is chosen as a unit of algorithm. The characteristics of functions written in C differ from the characteristics of functions implemented in Erlang. For example, Erlang introduces function clauses, as shown in Section 2.1, and this is an unknown concept in C. Thus, two different approaches are taken into count: a function clause or a function can be used as a unit.

Function clauses might seem to be the ideal choice at first sight, but there are really short (1 or 2 lines long) function clauses in Erlang programs usually. However, I decided to try out both functions and function clauses as units.

### **Predicted results**

This algorithm relies only on metrics and it does not use any syntactical properties of the candidates to determine the cloning level. This strategy is a double-edged sword. However no real clone is dropped due to the dissimilarity of their syntactical structures, but several false positive clones appear in the result. The more metrics are defined, the less false positive clones appear in the result, because every metric adds only one new aspect. The more aspects are given, the more accurate the results become.

The growth of false positive clones is especially expected if function clauses are used as units. In Erlang, there are many really short function clauses and plenty of them are base cases of recursions. They usually consist of only one simple top-level expression. These clauses have nearly no characteristic, thus metrics cannot evince them, therefore they are considered clones by this approach. This might lead into a situation when the full result of the algorithm may become useless and untraceable.

### 3.3 Control-flow based approach

In this section, I show a control flow based approach, which is only a sketch at the moment, where the control flow is used in new ways. Usually, similar sub-graphs of the control-flow graph are matched against each other, using data-flow information to refine the results. These solutions [26, 27] are usually very successful but can omit hidden clones. The new approach is based on the regular activities which are done by plagiarizing students. These activities are:

- inlining some functions,
- extracting some functions,
- renaming entities inside a module,
- changing the order of entities inside a module,
- changing the order of the parameters in the head of a function,
- moving functions between modules,
- dividing a body of a function into new functions,
- aggregating all the clauses of a function into a new function by using a *case* expression.

These activities can be detected by reorganising the source of the module along the control flow. From the reorganised module, almost all the differences caused by those activities automatically disappear. Further generalisation may be done on the reorganised module to vanish automatically differences which are caused by renaming entities.

To find clones such matchings are advised to be used which tries not only match the entire module against to others, but it tries to match sub-sequences of the module against to others. For this purpose, data mining methods specialized for sequential database might be used, which are described in [31].

This kind of matching is needed, because a whole module is rarely cloned without any modification in industrial applications. If plagiarism is focused, then it becomes more likely.

The reorganisation of modules can be used as a base order for my algorithm, which is detailed in Section 3.1. Instead of gathering the units in the same order as it can be got from the lexical analyser, the units are collected in the same order as it can be determined by the reorganised module.

# Chapter 4

## Realisations

This chapter details the realisations of the AST/metric based approach (Section 4.1) and also the metric based approach (Section 4.2). The optimisation steps (Section 4.3) and utilities (Section 4.4) are also reviewed.

### 4.1 Solutions based on the AST/metric based approach

The architecture of the AST/metric based approach is described in this section. It is based on the abstract algorithm of this solution, which is described in Section 3.1. The main components are:

- `CandidatesGenerator` produces the set of clone candidates;
- `InitialClonesDetector` creates the set of potential clones;
- `FilteringAndTrimmingUnit` forms the resulting clone set.

#### 4.1.1 Producing clone candidates

**D4.1. Identifier of an entity.** *Syntactic and semantic information can only be queried from `RefactorErl` about an entity if its identifier is known. Therefore, to be able to always exactly identify the units, their identifiers are preserved during the whole algorithm.*

**D4.2. ETS table.** *A ETS table is an efficient, key-value store which provides concurrent reading and writing for Erlang processes. The lifetime of the table is the lifetime of the process which owns it.*

The initial clone detector of this algorithm works with generalised units (top-level expressions), therefore conversions are needed. The General unit generator, described in Section 4.4.1 is able to perform the conversion. Afterwards, the ETS table (D4.2) contains the identifier of the top-level expression (D4.1), the identifier of its parent and its generalised value in each row for every top-level expression found in the Semantic Program Graph. The identifier of the parent is used for trimming by the initial clone detector.

Every row in the ETS table has a unique index, which is given in the same order as top-level expressions are gathered from the Semantic Program Graph. This allows the preservation of the layout of the sources. The values of the indexes are assigned from a sequence whose first element is 1. A diagonal is a list of indexes and one of the descriptors of clone candidates.

The subject of the duplicated code analysis can either be every top-level expression found in the Semantic Program Graph, or those defined by the user. The latter case means that only some expressions need to be matched against every expression in the following way. Let us assume that those expressions have indexes ranging from  $m$  to  $m+k$ . The  $n$ -by- $n$  matrix, denoted by  $A_{n,n}$ , has to be reduced to a  $k$ -by- $n$  submatrix, denoted by  $S_{k,n}$ , by deleting every row whose index is either between 1 and  $m-1$  or between  $m+k+1$  and  $n$ . The diagonals of this matrix are still describing the clone candidates in this case.

$$A_{n,n} = \begin{pmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ \mathbf{a}_{m,1} & \cdots & \mathbf{a}_{m,n} \\ \vdots & \ddots & \vdots \\ \mathbf{a}_{m+k,1} & \cdots & \mathbf{a}_{m+k,n} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{pmatrix} \Rightarrow \begin{pmatrix} a_{m,1} & \cdots & a_{m,n} \\ \vdots & \ddots & \vdots \\ a_{m+k,1} & \cdots & a_{m+k,n} \end{pmatrix} = S_{k,n}$$

The list of diagonals are given to the initial clone detector unit.

### 4.1.2 Determining initial clones

This initial clone detector is based on Algorithm 3.2, which is detailed in Section 3.1.3. The algorithm is a parallel algorithm with one coordinator process and many worker processes.

#### Coordinator process

First, the initial clone detector unit creates as many separate processes as many available logical processors can be used by the application. With this strategy, the administration and the timing of the processes are decreased as much as possible.

Then the detector divides the given list of diagonals into sublists of diagonals among the created processes. Each process is responsible for collecting clones from the given list of diagonals.

The coordinator process waits until all the worker processes have collected their results, then it forwards the result to the filtering unit.

#### Worker process

All workers can access and query the ETS table with indexes, thus avoiding the duplication of a huge amount of data.

The worker processes never try to build the matrix actually. Instead, they iterate through the given diagonals by accumulating the found clones. The calculations needed to detect clones, the initial clone detection and the trimming of the initial clones are done within a single iteration, as shown in Algorithm 3.2.

#### Variants

**D4.3. NIF.** *A NIF (Native Implemented Function) is a function which is implemented in a C-like language instead of Erlang. NIFs are transparent*

*from the caller side. To implement a NIF, an Erlang function and a corresponding C function are required and linked together. The C part of the function has to be defined in a shared library, which is loaded at runtime by the Erlang code of the encapsulated module.*

For determining whether a pair can form a clone, the similarity has to be calculated first. Several variants for determining the similarity were created to be able to choose the best one. The variants differ from each other mostly in the determination of the initial clones, because this part seems to be the most costly, from the computational requirements. Here, the variants are introduced. Every variant was measured, as detailed in Chapter 5.

**Type of the ETS table.** First of all, the type of the ETS table has to be chosen. The potential types were *set* and *ordered set*. Ordered set is unnecessary for the application because the content of the table is never sorted. Additionally, ordered set has a logarithmic access time, whilst set has a constant access time according to the official manual [1]. Thus, set is chosen for usage. This decision was absolutely right, the measurements have proofed it.

**String similarity metrics.** For determining the similarity, I use string similarity metrics. The normalised Levenshtein and Dice-Sørensen metrics were good candidates. However the way they express similarity is different. Levenshtein distance is the minimum number of edit operations to rewrite a word into another. The Dice-Sørensen metric is based on the doubled cardinality of the common unique bigrams of the two strings and the sum of the cardinalities of unique bigrams of each strings. An example is shown for both metrics in the following by calculating the similarity between **night** and **nacht**.

First, the calculation using Levenshtein metric is demonstrated. I use the following notions: characters that differ from each other are underlined, characters that had been edited are highlighted in bold, and arrows represent transitions between edit operations.

night, nacht  $\rightarrow$  night, night  $\rightarrow$  night, **n**ight

According to the example, two edit operations are needed to rewrite **nacht** into **night**, thus the Levenshtein distance of the two words is 2. The normalised Levenshtein metric value is computed as follows:

$$\frac{2}{\max(\text{length}(\text{night}), \text{length}(\text{nacht}))} = \frac{2}{\max(5, 5)} = 0.4$$

Next, the calculation using Dice-Sørensen metric is shown. By composing the bigrams of the strings, we get the following two sets:

$$\{ni, ig, gh, ht\}, \{na, ac, ch, ht\}$$

The set of the common unique bigrams can be determined based on the sets, it is  $\{ht\}$ . The metric value is computed as follows:

$$\frac{2|\{ht\}|}{|\{ni, ig, gh, ht\}| + |\{na, ac, ch, ht\}|} = \frac{2}{4 + 4} = 0.25$$

Based on my empirical results, it can be stated that the same clones can be found either by using Dice-Sørensen metric and defining the threshold to be 0.9 or by using Levenshtein metric and defining the threshold to be 0.8.

In the following statements  $m$  and  $n$  denote the length of the strings. Levenshtein metric uses  $\mathcal{O}(mn)$  space, while Dice-Sørensen metric uses  $\mathcal{O}(m + n)$  space, so the computation of Dice-Sørensen metric is cheaper, thus Dice-Sørensen is more suitable.

**Calculate only once.** I found a high duplication rate while I was studying the generalised units. If the algorithm can cache the calculated metric values, then the computational requirements of this variant becomes smaller. The general caching mechanism, described in Section 4.4.2, was used to implement it. This variant deeply depends on the available size of the memory, because if the available memory is not enough, the variant becomes much slower due to swapping between memory and disk.

**Pre-calculate everything.** Another variant consists in precalculating and storing the similarity for every two strings from the duplication-free set of generalised units. The worker process calculates nothing, and only queries the similarity of the examined pair. However, while the computational requirements of this variant is smaller, the space requirement is larger by orders of magnitude than the other variants. Indeed, for each pair of strings, their bigrams have to be constructed when the value of Dice-Sørensen metric is needed, hence the bigrams of each string are generated  $n - 1$  times.

**Sequence database.** SimString [8, 35] is a fast algorithm for approximate string matching and it seems to be promising for our purpose. SimString uses sequence database to store the base set of strings and it offers several string similarity metrics for matching, including Dice-Sørensen metric. The author of SimString stated the following [8]:

Fast algorithm for approximate string retrieval. For example, SimString can find strings in Google Web1T unigrams (13,588,391 strings) that have cosine similarity  $> 0.7$  in 1.10 [ms] per query (on Intel Xeon 5140 2.33 GHz CPU).

Thus I gave it a try, it is written in C++ and linking C-like application to Erlang is an established practice via NIF(D4.3), so I wrote a shared library which is suited for connecting the Erlang application to the SimString library. Using SimString from concurrent processes led to an abnormal termination, because the library is not prepared for concurrent usage - it is not thread-safe. Due to this deficiency, I had to solve the problem in a different way; I worked out two solutions which can be used in parallel environment.

The first solution consisted in placing an application server between the Erlang application and the shared library. The server is capable to synchronise concurrent requests by serialising them, so SimString can be used in a parallel environment. However the original problem was solved, my parallel application is sequenced along using SimString, thus it becomes much slower.

The second solution was multiplying sequence databases among processes. In other words, every worker process has its own local sequence database with

the same content and each local sequence database is accessed via the shared library. This works well, but the disc requirements are largely increased.

**To summarise the consequences:** the base variant with Dice-Sørensen metrics should be used on a personal computer which has a few amount of memory, but the variant with the general caching mechanism should be used within a server environment where a plenty of memory is available.

### 4.1.3 Filtering and trimming initial clones

The filtering and trimming unit is based on Algorithm 3.3, detailed in Section 3.1.5, is presented in this section. The algorithm is a parallel algorithm with one coordinator process and many worker processes. The creation of workers is analogous to the strategy, described in Section 4.1.2.

Due to the generalised unit and the similarity metrics, the initial clone detector returns a huge amount of initial clones, so the filtering unit has to deal with these clones by trying to decrease their volume and to filter out false positive clones.

There is one coordinator process which partitions the initial clones among the worker processes. The task of every worker is to construct the resulting clones from the given ones and to return the results to the coordinator. The coordinator waits for all the workers and collects all the results which become the result of the whole algorithm.

A lot of data is produced, which requires caching, described in Section 4.4.2. Almost all the information comes from some source code metric, so a *metric handler framework* deals with them. The framework is designed to be easily extendible, efficient and to work in a parallel environment.

The filtering unit, as described in Section 3.1.5, uses several *filters* which are handled by the metric handler framework. Each filter consist of a *calculator part* and a *decider part*. To evaluate a filter on a pair, the calculator part computes the corresponding metric values for both components of the pair, then the decision is made by the decider part of the filter based on the calculated metric values. A clone can be dropped or kept in the result set

based on the decision. Whenever the decision is to drop a clone, it is done so instantly because a conjunction exists within the filters. The workers evaluate the available filters via the metric handler framework. The evaluation has a lazy style: filters are calculated only when they are needed to make a decision on a pair.

## 4.2 Realising the metric based approach

In this section, I detail the adapted version of the metric based duplicated code detector. The original idea behind the algorithm comes from Mayrand et al. [34] and is discussed in Section 3.2.

The application of the metric based approach consists of two parts: the first is the production of the candidates, and the second is the determination of the cloning level. These parts are going to be discussed in great detail in the following sections.

### 4.2.1 Producing clone candidates

The metric based approach is implemented so as extendible as possible, thus units can be either function clauses or functions. The units are either all the function clauses or all the functions, and they can be queried from the Semantic Program Graph.

The units are given by the *General unit generator*, described in Section 4.4.1. After the generator has finished, an ETS table (D4.2) contains the requested entities with indexes, which are generated in the same way as described in Section 4.1.1. The strategy for producing candidates is that every entity is needed to be compared with every other entity, as mentioned in Section 3.2.1. These comparisons can be defined with the following list of index pairs, assuming that the cardinality of the units is  $n$ .

$$\{(i, j) \mid i \in [1 \dots n - 1] \subset \mathbb{Z}, j \in [i + 1 \dots n] \subset \mathbb{Z}\} \subset \mathbb{Z} \times \mathbb{Z}$$

These pairs are the input of the second component.

### 4.2.2 Determining the cloning level

The second part of the algorithm is based on the abstract algorithm shown in Algorithm 3.6 and described in Section 3.2.1.

It is a parallel algorithm with one coordinator process and many worker processes. The creation of workers follows the same policy as that, described in Section 4.1.2.

#### Coordinator process

The given pairs (candidates) are subdivided among the workers by the coordinator. The coordinator waits for all workers to complete their tasks, then collects every results from them. These are pairs of functions or function clauses with their determined cloning level.

#### Worker process

The task of every worker is to determine the cloning level of the given subset of pairs.

The values related to the points of comparison need to be computed to classify each pair. As described in Section 3.2.1, the values depend on the corresponding metrics and deltas.

All the metrics have the same properties: the definition of the metric itself and the given delta. Thus a general, extendible and efficient framework was developed, which can be simultaneously used by concurrent processes. This framework provides the available metrics grouped by the point of comparison. For each metric, it further enables access to the value of the metric for the given unit, and it makes the corresponding delta available for the workers.

In the worst case, all the metrics need to be calculated for both components of a pair to determine the cloning level of the pair. Therefore, all the metrics are calculated for every unit  $n - 1$  times with the assumption that the cardinality of the units is  $n$ . Although RefactorErl provides an efficient way to gather information from the Semantic Program Graph, caching the computed values is really necessary to achieve good performance. A General

caching mechanism was developed (see Section 4.4.2) for this purpose, thus every metric value of a unit is calculated only once.

Each worker iterates through the given pairs. At each iteration, the cloning level of a candidate is determined. For each point of comparison, the corresponding metrics with deltas are queried from the framework, and the value related to this point of comparison is determined in a lazy manner: a metric is only calculated when it is necessary. If the cloning level can exactly be assessed with the known values related to some of the points of comparison, then no computation is performed. For example, if the value of the Control flow view is “neq” for a candidate, then the candidate can be classified, therefore further computations are not performed.

## 4.3 Optimisations

One of my goals was to work out a solution which can be used in legacy code, with consideration for efficiency, computational and space cost-effectiveness and scalability. Whenever possible, the solution was designed and developed to be able to work in a parallel environment.

The parallelism, already mentioned in Section 4.1.2, improves the performance as much as possible with the run-time environment. More precisely, this means that if a computer has only 2 processors, then 2 worker processes are started, but if a computer has 32 processors, then 32 workers are started.

I worked out a completely generalised caching mechanism, detailed in Section 4.4.2, with which anything will be evaluated only once. To avoid unnecessary computations every value is calculated only if it is really needed.

**Optimisations related to the AST/metric based approach.** By introducing diagonals, the initial clone detector was improved greatly: the computation for determining whether a pair can form a clone, the extension of the clones and the trimming of the clones were fused into a single iteration. Additionally, further extensions of clones are not needed, because, with this algorithm, the longest clone is found.

The matrix is never constructed due to the diagonals, therefore a huge

amount of data do not need to be stored, so the space requirement is greatly reduced.

## 4.4 Utilities

In this section two interesting modules are detailed, which were used by both algorithms.

### 4.4.1 General unit generator

This module is capable of solving two tasks:

- collecting entities from the Semantic Program Graph along the given specification,
- transforming entities over a given alphabet.

#### Collecting entities

The module defines a hierarchy and introduces levels between different types of entities as follows.

1. *File* level is the starting level. Modules and header files are considered as File level entities.
2. *Form* level is under the File level. Modules and header files are built up from attribute forms and function definition forms. These forms are considered as Form level entities.
3. *Clause* level is under Form level. Function definition forms are built up from function clauses which are considered as Clause level entities.
4. *Expression* level is under Clause level. Top-level expressions and any sub-expression of a top-level expression are considered as Expression level entities.
5. *Token* level is under Expression level. Every syntactic structure is built up from tokens which are considered as Token level entities.

A mapping rule exists between each hierarchy level, defining the decomposition of entities into smaller ones. These rules can be redefined by passing new rules as parameters to the algorithm. There are default mapping rules, which are the following.

- *Selecting the starting nodes*: get all modules from the Semantic Program Graph.
- *From File to Form level*: get all function definition forms of the parent module.
- *From Form to Clause level*: get all function clauses of the parent form.
- *From Clause to Expression level*: get all top-level expressions of the parent clause.
- *From Expression to Token level*: get all tokens of the parent expression.

The decomposition can be stopped before Token level by defining the target level, which is Expression by default.

If the transformation of the entities is not requested, then the decomposed entities are simply returned or inserted into an ETS table with their parents depending on the actual parametrisation of the module.

## Transforming entities

The abstract algorithm to transform entities is shown in Algorithm 3.1.

The decomposed entities are usually not Token level entities, therefore the transformation to Token level needs to be done first. The basic mechanism of this transformation is: while preserving the layout, gather all the leaves of the syntax tree of the given entity. For example, the tokens of the top-level expression which is shown in Erlang source 14 are:

```
Result, =, case, f, (, ), of, ok, ->, 1, ;;, _, ->, -1, end
```

The decomposed entities can be transformed over an abstract alphabet, which is given by default. The alphabet assigns different words to different

```

Result = case f() of
    ok -> 1;
    _  -> -1
end

```

Erlang source 14: An example source for the abstract alphabet

token types, except the following types: *atoms*, *variables* and *numbers*. For example, Erlang source 14 is transformed to the following sentence represented as a string: "A=cA()pAzA;AzAe".

The transformed entities are simply returned or inserted into an ETS table with their parents depending on the actual parametrisation of the module.

#### 4.4.2 General caching mechanism

The caching mechanism is as general as possible and is capable of handling multiple cache tables simultaneously and it can be used by concurrent processes as well.

The rationale is that each computation should be done only once and only if the result of the computation is really needed.

Every row in a cache table contains computed values related to the same entity which is identified by a unique key, thus different properties of one entity can be grouped. In other words, every cache table has its own schema, which is defined by a record definition, as usual in Erlang.

Every property of an entity can be cached by answering the following questions:

- What is the name of the cache table?
- What is the key of the entity? In other words, what is the identifier of the entity?
- Which of the properties is needed?
- How to initialise a new row for the entity if the corresponding row of the entity has not been found?

- How to compute the value of the property?

The answers to the questions can be hidden behind a wrapper function or a lambda expression, therefore the caching becomes transparent to the callers.

The algorithm performs the followings steps. First, it tries to find the proper row for the given key in the referred cache table, which is an ETS table.

If the row is found, then the value of the given property is tested. If the value is different from the initial value, no further computation is needed, and the found value is returned. If the value has never been computed, then the value is computed and stored before being returned. If the row is not found, a new row is inserted into the cache table and the value is computed and stored before being returned.

# Chapter 5

## Measurements & evaluations

In this chapter, runtime measurements and memory usages are presented by focusing on the AST/metric based approach. Based on the measurements, I select two variants that are advisable to be used. Last, but not least I evaluate the results of the algorithms and I show some clones as examples.

### 5.1 Measurements

#### 5.1.1 Configurations

Measurements assume two different configurations, each distinguished by an identifier. The configurations are detailed in Appendix A, but the most important properties are shown in Figure 5.1.

| Identifier | Operating System   | Memory <sup>1</sup><br>(GB) | Num. of log.<br>processors <sup>2</sup> | Erlang/OTP<br>version |
|------------|--------------------|-----------------------------|-----------------------------------------|-----------------------|
| Config A   | Ubuntu 12.04.2 LTS | 20                          | 48                                      | R15B01                |
| Config B   | OS X 10.6.8        | 2                           | 8                                       | R14B03                |

Figure 5.1: Important properties of the used configurations

---

<sup>1</sup>can be used by the solutions

<sup>2</sup>as reported by the Erlang VM

### 5.1.2 Measurement methodology

Every measurement is performed at least with *Config A*. If a measurement is performed on more than one configuration and has resulted in an unexpected difference between the measurements, then the difference is discussed. Nobody should trust one measurement, so every measurement is performed at least three times, and the calculated average time is shown. The `timer:tc/3` function [3] is used to measure the real elapsed time in microseconds, which is converted to milliseconds or seconds.

Almost all the measurements are performed by giving the same source code as subject to the detectors, which is introduced in Section 5.1.3. If other source code is used then it is stated.

### 5.1.3 Mnesia - Subject of the detectors

Mnesia [2, 33] is a distributed, soft real-time database management system, which is part of Erlang/OTP. The source code of Mnesia 4.7 is used as a subject of duplicated code detectors. Some properties of the examined source code is listed below:

- 22 594 non-empty lines of code,
- 31 modules,
- 1 687 functions,
- 5 393 top-level expressions.

### 5.1.4 Back-ends of RefactorErl

RefactorErl provides 3 different storage back-ends which can be used to store the loaded source in it. These are based on:

- *Mnesia*: Mnesia supports parallel queries, but it writes data to disc very slowly.
- *NIF-graph*: This back-end has been developed by the RefactorErl team. The back-end is implemented in C++ and connected to Erlang by NIFs. It does not support parallel queries, but it writes data to disk faster

than Mnesia does. The NIF-graph tries to keep the whole Semantic Program Graph in memory, thus data accesses are faster than using Mnesia while the graph can be kept in memory.

- *Kyoto-Cabinet*: This back-end uses the Kyoto-Cabinet database [5]. It is implemented in C and connected to Erlang by NIFs. Although the back-end does not support parallel queries, it writes data to disk faster than Mnesia does. It has a lower memory footprint than NIF-graph has, and data accesses are also faster than using Mnesia.

To test the back-ends and to be able to choose the most suitable, the following test is set. The applications read the database in parallel while they are trying to detect clones, thus the test tries to simulate and assess this behaviour.

The base of the test is a sequence whose elements query from the Semantic Program Graph. The queries are equivalent to 8–10 steps in a graph, and they result in a huge set of results. This base becomes the worker process, run in parallel with its copies. The parallelism follows the same policy than that of the applications. The execution is measured 3 times.

The content of the databases represents one library of RefactorErl, called `referl_lib`. It has the following properties:

- 9 635 non-empty lines of code,
- 36 modules,
- 841 functions,
- 1 968 top-level expressions.

The test pointed out that the NIF-graph is the fastest on all configurations. The result of this test is interesting, because the orders among the back-ends vary depending on the used configuration. The results of *Config A* can be seen in Chart 1, the results of *Config B* can be seen in Chart 2.

### 5.1.5 Variants of the AST/metric based approach

In this section, the variants of the AST/metric based approach are discussed and compared with each other.

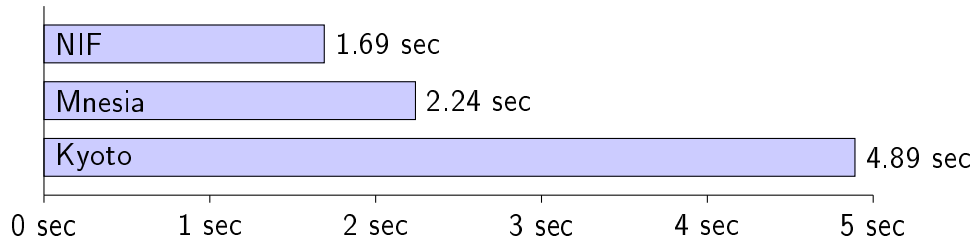


Chart 1: Performance of back-ends measured with *Config A*

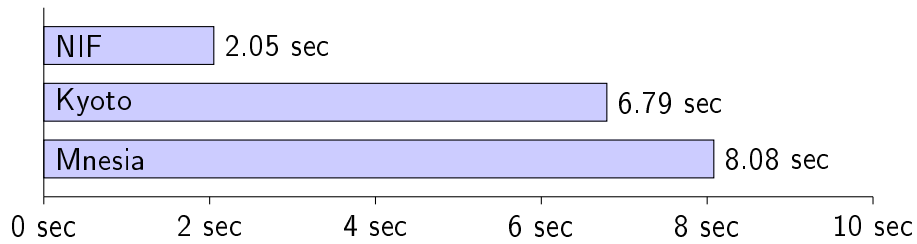


Chart 2: Performance of back-ends measured with *Config B*

### Dice-Sørensen vs. Levensthein

The difference between Dice-Sørensen and Levensthein is shown in Chart 3. In conclusion Dice-Sørensen metric is retained.

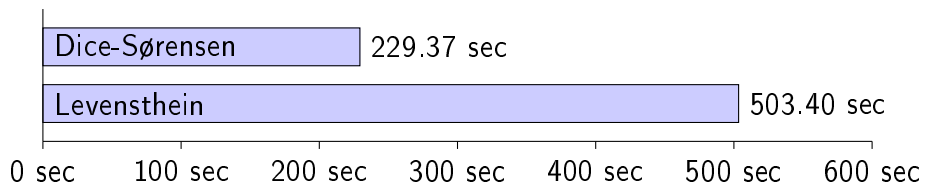


Chart 3: Dice-Sørensen vs. Levensthein measured with *Config A*

### Caching vs. no caching

The difference between the original solution and the solution which uses the General caching mechanism is shown in Chart 4.

As mentioned already, the efficiency of the variant using the General caching mechanism depends highly on the available memory. If the memory

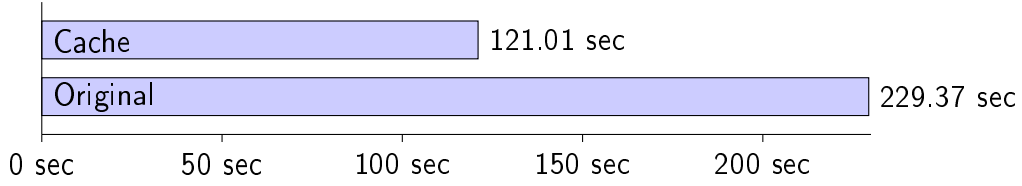


Chart 4: Caching vs. no caching measured with *Config A*

is not large enough, the runtime of the variant is greater than the original solution. If the memory is sufficient, the variant using the General caching mechanism is the most suitable.

#### Caching vs. using pre-calculated data

The difference between the solution using the General caching mechanism and the solution which pre-calculates metrics values for all different pairs, is shown in Chart 5.

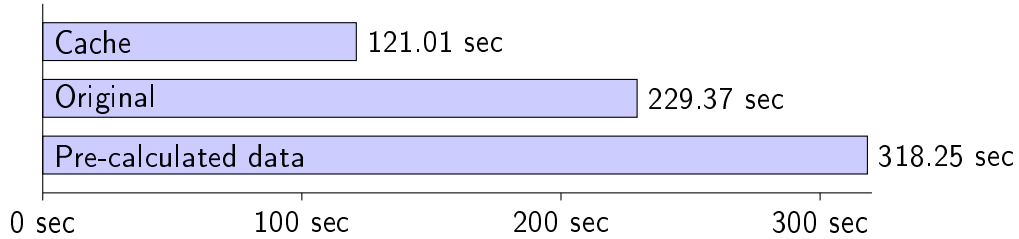


Chart 5: Caching vs. using pre-calculated data measured with *Config A*

If the memory is sufficient, the variant using the General caching mechanism is preferred, otherwise the original solution should be chosen, because the solution which pre-calculates data is the worst. The reason is that there is only a complex and costly method for pre-calculating all necessary data.

#### Calculating everything vs. central sequence database usage

The difference between the solution which uses a central sequence database and the original solution is shown in Chart 6. The measurement was per-

formed on a smaller database (`refer1_lib`) with *Config B*.

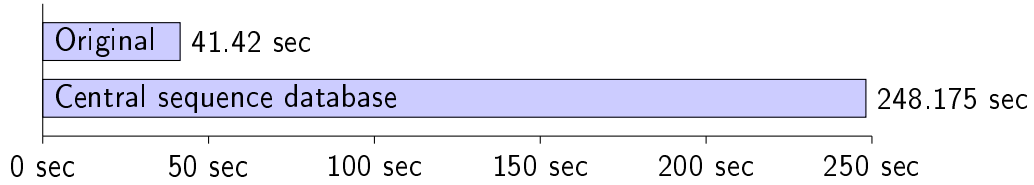


Chart 6: Calculating everything vs. using a central sequence database measured with *Config B*

As previously stated, the parallel application is sequenced along using SimString, so the original solution should be chosen.

### Memory usage

The maximum amount of allocated memory is shown for every variant in Chart 7. These amounts were allocated either by the Erlang VM or by the Erlang processes with *Config A*.

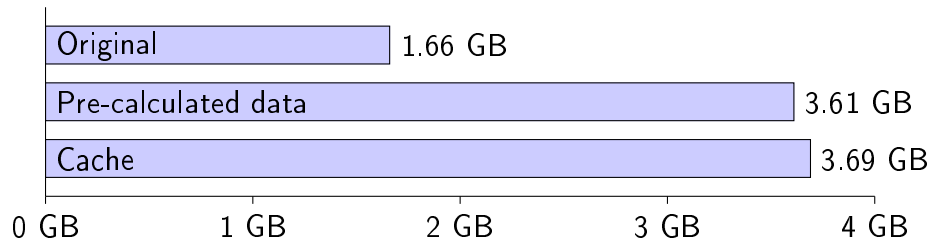


Chart 7: Size of allocated memory with *Config A*

### 5.1.6 Assessment of the measurements

Every variant of the AST/metric based approach was measured and the results were shown in the previous section.

The original solution using Dice-Sørensen metric is usable on both a developer computer and on a server. The corresponding runtime can be seen in Chart 8.

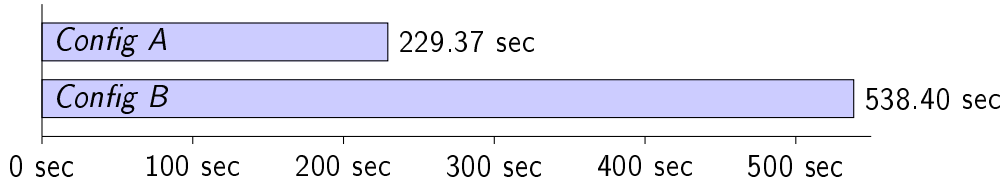


Chart 8: Original solution measured with different configurations

To summarise: if the memory does not matter, then the solution which uses the general caching mechanism is better, otherwise the original solution should be preferred.

## 5.2 Evaluations

The implementation of the algorithms, which are extremely specialised on Erlang, were successfully executed on Mnesia [33], and resulted in clones. Unfortunately, I had to analyse the detected clones manually, because the well-known Bellon’s benchmark [12] deals with only imperative languages and does not offer any test projects written in Erlang, and I have not found any alternatives specialised for Erlang.

### 5.2.1 AST/metric based approach

The AST/metric based approach was successfully evaluated on Mnesia, resulting in 801 clone pairs. Thanks to my filtering system, neither irrelevant nor false positive clones were found.

The main types of clones are duplicated configurations, handler branchings, debugging sequences, constructions of validator functions and message processing. One of the interesting results is shown in this section, and further noteworthy clones are presented in Appendix B.

The body of `mnesia:val/1` has 156 cloned occurrences, that slightly differ from each other. One of these clones can be seen as the example clone in Erlang source 15, where the extent of generalisation differ from each other. The instances of the clone provide the same service, therefore they can be

```

% First instance (found in mnesia_loader):
case ?catch_val(send_compressed) of
    {'EXIT', _} ->
        mnesia_lib:set(send_compressed, NoCompression
            (cont.)),
        NoCompression;
    Val -> Val
end

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

```

Erlang source 15: A frequent clone found in Mnesia by the AST/metric based approach

eliminated.

### 5.2.2 Metric based approach

The metric based approach was evaluated with *Config A*. Two different cases were examined, which differ from each other in the chosen unit. Function clauses and functions were the candidates. Using functions as units results in a smaller result set, whose elements are more usable clones.

As stated earlier, the set of initial clones contains several false positive or irrelevant clones. This statement is especially true in the event of this approach, because the initial clone detection is not syntax-driven, thus clones whose structures are different from each other appear in the set of initial clones.

#### Function clauses as units

The execution time of the detector was around 448 seconds, which is not bad if the cardinality of comparisons are taken into consideration. There are 4 212 253 candidates in the clone candidates set, so the algorithm performs

4 212 253 comparisons. The maximum amount of allocated memory was 2.27 GB, including both the needs of the Erlang VM and Erlang processes. The algorithm found 52 514 clone pairs whose cloning level was less than or equal to 4. Unfortunately, several irrelevant clones exist in the result of the algorithm, thus the result is not comprehensive. A false positive clone and an irrelevant clone detected by the algorithm are shown in Erlang source 16 and in Erlang source 17, respectively.

```
% First instance (found in mnesia):
deloid(_Oid, []) ->
    [];

% Second instance (found in mnesia_tm):
sync_send_dirty(_Tid, [], _Tab, WaitFor) ->
    {WaitFor, {'EXIT', {aborted, {node_not_running, WaitFor}}}
    (cont.)}};
```

Erlang source 16: A false positive clone, whose cloning level is 3, found in Mnesia by the metric based approach

```
% First instance (found in mnesia_lib):
search_delete(Obj, [Obj|Tail], Ack, _Res) ->
    search_delete(Obj, Tail, Ack, Obj);

% Second instance (found in mnesia_tm):
sublist([H|R], N, Acc) when N > 0 ->
    sublist(R, N-1, [H| Acc]);
```

Erlang source 17: An irrelevant clone, whose cloning level is 3, found in Mnesia by the metric based approach

## Functions as units

The runtime of the detector was 84 seconds, which is extremely fast. The algorithm dealt with 1 687 functions, which means 1 422 141 comparisons were performed. The maximum amount of allocated memory was 0.37 GB, which is a very good result. Moreover, 4 860 function pairs were found by the algorithm, whose cloning level was less than or equal to 4. Although this

result is much better than the one is shown in the previous subsection, more than 70 percent of the clones are still useless, so the result of the algorithm is still hard to study. A useless clone and also a false positive clone detected by the algorithm are shown in Erlang source 18 and in Erlang source 19, respectively.

```
% First instance (found in mnesia):
is_dollar_digits(Var) ->
    case atom_to_list(Var) of
        [$$ | Digs] ->
            is_digits(Digs);
        _ ->
            false
    end.

% Second instance (found in mnesia_recover):
cast(Msg) ->
    case whereis(?MODULE) of
        undefined -> ignore;
        Pid -> gen_server:cast(Pid, Msg)
    end.
```

Erlang source 18: An irrelevant clone, whose cloning level is 3, found in Mnesia by the metric based approach

```
% First instance (found in mnesia_controller):
default_merge(F) ->
    F([]).

% Second instance (found in mnesia_frag_hash):
power2(Y) ->
    1 bsl Y. % trunc(math:pow(2, Y)).
```

Erlang source 19: A false positive clone, whose cloning level is 2, found in Mnesia by the metric based approach

Besides the false clones reported by the algorithm, real clones were found in the result set, too. The cloning level of the clone, shown in Erlang source 20, is 2, because the following pair only differs in their identifiers. Obviously, the generalisation of the clone can be easily done.

```

% First instance (found in mnesia_tm):
commit_delete([], _, _, _, _, _) -> ok;
commit_delete([{{checkpoints, CpList}}|R], Tid, Tab, K, Obj, _)
  (cont.) ->
    Old = mnesia_checkpoint:tm_retain(Tid, Tab, K, delete
      (cont.), CpList),
    commit_delete(R, Tid, Tab, K, Obj, Old);
commit_delete([H|R], Tid, Tab, K, Obj, Old) when element(1, H
  (cont.)) == subscribers ->
    mnesia_subscr:report_table_event(H, Tab, Tid, Obj,
      (cont.)delete, Old),
    commit_delete(R, Tid, Tab, K, Obj, Old);
commit_delete([H|R], Tid, Tab, K, Obj, Old) when element(1, H
  (cont.)) == index ->
    mnesia_index:delete_index(H, Tab, K),
    commit_delete(R, Tid, Tab, K, Obj, Old).

% Second instance (found in mnesia_tm):
commit_del_object([], _, _, _, _, _) -> ok;
commit_del_object([{{checkpoints, CpList}}|R], Tid, Tab, K, Obj
  (cont.), _) ->
    Old = mnesia_checkpoint:tm_retain(Tid, Tab, K,
      (cont.)delete_object, CpList),
    commit_del_object(R, Tid, Tab, K, Obj, Old);
commit_del_object([H|R], Tid, Tab, K, Obj, Old) when element
  (cont.)(1, H) == subscribers ->
    mnesia_subscr:report_table_event(H, Tab, Tid, Obj,
      (cont.)delete_object, Old),
    commit_del_object(R, Tid, Tab, K, Obj, Old);
commit_del_object([H|R], Tid, Tab, K, Obj, Old) when element
  (cont.)(1, H) == index ->
    mnesia_index:del_object_index(H, Tab, K, Obj, Old),
    commit_del_object(R, Tid, Tab, K, Obj, Old).

```

Erlang source 20: A relevant clone found in Mnesia by the metric based approach

## Chapter 6

# Comparison with Wrangler

In this chapter, I compare my AST/metric based approach with another duplicated code detector software included in Wrangler. I briefly introduce this detector in Section 6.1, then the method of comparison is described in Section 6.2. Finally, the results of the comparison are assessed in Section 6.3.

### 6.1 The clone detector of Wrangler

Wrangler [30] is an open source refactoring tool for Erlang, maintained at the University of Kent (United Kingdom). Wrangler is also written in Erlang and provides user interfaces in Emacs and in Eclipse. The duplicated code detector of Wrangler is a

token and AST based hybrid approach

as stated in the paper by [29].

The approach tokenises the source code, then transforms tokens over an abstract alphabet, then builds the abstract syntax tree and a suffix tree from them. The initial clones are gathered from the suffix tree, then further analysis are performed using semantic binding structure information, and location information to detect only such clones which form a syntactic unit and can be removed by consistent renaming of variables and constants. The clones are grouped along their anti-unifications (see D1.6).

The algorithm is implemented in Erlang, except the suffix tree algorithm which is written in C. The algorithm contains some parallelism, for example, the suffix tree and the abstract syntax tree are built in parallel, but it does not seem to be designed to be parallel. This can be said, because the structure of the algorithm necessitates sequential execution and the parallel parts are implemented using the built-in primitives provided by Erlang to manage both processes and inter-process communications. Using these primitives without any strategy related to parallelism can cause a performance bottleneck and can make the whole application become a non-scalable application.

The detector has the following parameters:

- *Minimum length of a clone.* Clones whose length are less than the given value are not detected.
- *Minimum number of tokens.* Clones which have less tokens than the given value are not taken into consideration.
- *Minimum frequency of a clone occurrences.* Groups which have less clones than the given value are dropped.
- *Maximum count of consistent renaming.* Clones which can be refactored into each other by more renamings than the given value, are not reported.
- *Similarity score.* Clones whose similarity is less than the given value are not included in the result.

The first three parameters are the most important. These parameters greatly influence the cardinality of the initial clones, because only such clones are gathered from the suffix tree which satisfy the constraints derived from the parameters.

## 6.2 Method of comparison

My AST/metric based approach uses top-level expressions as units which have to be matched with tokens used by the detector of Wrangler.

My algorithm deals with a single top-level expression which is equivalent to a token in the worst case. However, a suffix tree can deal with such searches,

but these are not efficient in this case. Choosing 1 for the minimum number of tokens would be unfair with the algorithm, because the measurement of this execution points out only the worst profile of the algorithm, so 10 becomes the value of the parameter. One-character-long variable names are allowed in the Erlang grammar, thus 1 is chosen for the minimum length of a clone. My algorithm returns clone pairs as result, therefore the minimum frequency of a clone occurrences should be 2. The maximum count of consistent renaming is 20 to avoid dropping potential clones, because my algorithm does not take into account this parameter. The similarity score is 0.6, because the metric, which is used to calculate it, was not defined exactly either in the implementation of Wrangler’s algorithm or in the official documentation. By setting the discussed values to these parameters, the detector nearly handles the same amount and type of candidates as my algorithm does.

However, there is no dedicated way to use Wrangler from an Erlang shell, I managed to use it from the console finally by studying the source code. The measurements were performed with *Config A* using Mnesia (see Section 5.1.3) as the subject of the detectors.

## 6.3 Assessment of the results

Both of the algorithms were successfully executed on Mnesia, and resulted in several clones. I could not find a clone which was reported by Wrangler and was not found by my algorithm, except for trivial clones.

```
% First instance
Ops2 = [{op, rec, unknown, {NewOid, [Rec], write}}

% Second instance
{op, rec, unknown, {OldOid, [OldOid], delete}} | Ops]
```

Erlang source 21: An instance of a trivial clone, reported by Wrangler

Indeed, thanks to my filters, my results were more usable as trivial clones are not flooded the user. For example, the useless expression, shown in Erlang source 21, was reported by Wrangler, but not by my algorithm.

The runtime measured for the two algorithms is shown in Chart 9.

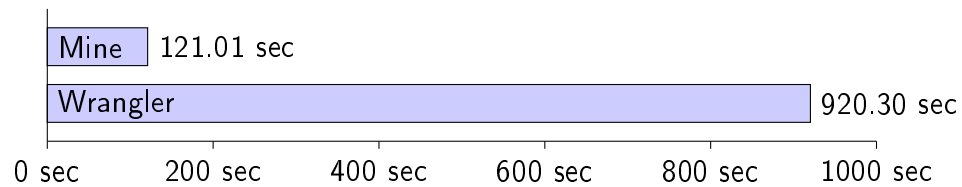


Chart 9: My algorithm vs. Wrangler's algorithm measured with *Config A*

# Chapter 7

## Enhancement possibilities

In this chapter, some enhancements for the AST/metric based approach are discussed. The common base of the improvements are reviewed, then possible solutions are discussed.

**Common bases** All enhancements deal with the production of the initial clones, because this part of the algorithm has an intensive computational requirement. For example, in Mnesia, there are 5 393 top-level expressions. Examining diagonals is efficient, but the number of comparisons remains huge: more than 14 millions of them are needed to produce the initial clones, if Mnesia is the subject. The result of the initial clone detector is also very large, thus the filtering and trimming unit of the algorithm has to compute a lot to narrow down the set of initial clones, and this is another area of improvement.

### 7.1 Decreasing the cardinality of initial clones

**Introducing a new constraint** The trivial solution to decrease the computational needs is to introduce a new constraint, namely, the minimum length of a clone. This constraint indicates that candidates whose lengths are less than the minimum length are not taken into account, thus the problem space is largely decreased. However, I have decided not to limit the algorithm

by this constraint, because it is likely to happen that small clones are also valuable.

**Elaborating a heuristic** The efficiency of the whole algorithm can be increased if the cardinality of the initial clones can be decreased somehow, for example by introducing a heuristic. The heuristic should be strict enough to slightly narrow down the set, but not too strict to avoid dropping real clones. Moreover, it should focus primarily on the one unit long clones, because this kind of clone is often useless. The useless clones should be gathered and classified in order to infer rules for the heuristic.

## 7.2 Replacing the technique of the initial clones detector

Another possibility is to use a suffix tree to produce initial clones. The longest common substring can be gathered from a suffix tree in linear time, which is equivalent to a clone.

In this case, new trimming rules need to be defined to split overlapping clones. Working with suffix tree has a benefit: the clones are grouped together due to the nature of the algorithm, but it also causes some extra work for the algorithm. Namely, the new trimming rules should also maintain the whole group from which the trimmed clone instances are originated.

Despite the time linearity of the new technique, the behaviour of the initial clone detector has to be rethought, and it is likely to happen that new methods have to be added, too. With these new methods it cannot be guaranteed that the computational space of the whole component will become smaller, because the complexity of the redesigned initial clone detector might increase too much.

Last but not least, using this new technique implies that pointing out clones that contains some invalid sequence becomes impossible.

### 7.3 Self-organising general caching mechanism

In Section 5.1.6, I pointed out that the caching variant uses twice as much memory as the non-caching variant in the event of Mnesia is being analysed. The memory requirement of the caching variant depends on the analysed source code, so it cannot be predicted in advance. If the available memory is not enough, then the caching variant becomes slower than the non-caching variant because of swapping. If the general caching mechanism is reimplemented as a self-organising solution, the caching variant becomes usable on any computer. The new caching mechanism needs to be re-organised as follows.

During the initialisation of caching, the algorithm measures the average time of gathering already cached information. These times stand for reference times during the usage of caching. While the cache is being used, each information gathering is measured. If the difference between the current measured time and the maximum of the references times is greater than a threshold, then the algorithm has consumed all the available memory. In this event, the caching mechanism can continue in two separate ways.

The simplest way is to make the cache empty by dropping all cached information and then if a new query arrives, the caching mechanism calculates and stores the result of the query again. The another way is to drop only a small amount of cached information. After that, if a new query arrives, the caching mechanism tries to serve the information from the cache first. If the information cannot be found in the cache, then the caching mechanism calculates the result of the query and returns it without storing the new information into the cache.

### 7.4 Exploiting GPU instead of CPU

The whole initial clone detector can be rewritten in C++ AMP, which is a new C++ language feature extending the language to be able to easily write parallel applications evaluated on a GPU [44]. The computational capacity of a GPU is an order of magnitudes larger than the capacity of a CPU. While

a CPU usually has multiple cores, a GPU has thousands of cores.

The weakest point of this solution is the inter-application communication as usual. If interactions were necessary, then communications would be expensive in this case, because they may be long-lasting that can corrupt the scheduling of the whole Erlang VM and can cause memory leaks. Additionally, interaction would be a special inter-application communication that necessitates copying huge amount of data between CPU and GPU, which is considered to be very inefficient. Fortunately, the initial clone detector of the algorithm uses no information from the Semantic Program Graph, therefore costly communications between Erlang and C++ do not matter. In [41], a parallel algorithm implemented in C++ AMP is 650 times faster than the unoptimised algorithm.

The main problem is that only one implementation of C++ AMP exists, maintained by Microsoft, who provides only Windows-specific support. Until no implementation exists on other platforms, the replacement can not be done, because the platform-independence of the solution is essential.

# Conclusion

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

In my thesis, I described issues with clones and the related standard principles of software development. I brought to the fore possible scenarios to deal with them.

I constructed an abstract, general algorithm that fits every concept known to me, which was also formalised. The algorithm is presented with the hope that it will foster a better understanding of the field of inquiry.

I gave a detailed review at the methods published in the last decades. Every theory was detailed by linking it to the abstract algorithm and a real implementation was also shown in which the described technique was used.

I elaborated a new, efficient AST/metric based approach which is designed to be parallel, and also capable of handling industrial source code. The algorithm is presented abstractly to highlight the ideas behind it. In my thesis, I also presented the case studies which helped me to define filters, which can be considered heuristics, for the algorithm. As to my current knowledge, none of the published papers proposed a pluggable filtering system to provide more accurate results, thus mine is a novelty. I implemented different variants of the algorithm to choose the best. The selection followed a defined methodology and was based on documented measurements.

My approach was compared with other clone detector for Erlang and my solution performed well in the comparison. Additionally, I described further enhancement possibilities. This duplicated code detector was successfully

evaluated on an open-source project, which has more than 22 000 non-empty lines of code. The software will be the part of the next official RefactorErl release. A publication about this algorithm has been submitted to the *13th Symposium on Programming Languages and Software Tools*.

Furthermore, I also studied other techniques. Following an atypical, metric based algorithm, I pointed out how differently can an imperative language be characterised in contrast with a declarative language, using the originally proposed software metrics. I adapted this algorithm to Erlang by trying to preserve what the author wanted to express by the original metrics. Last but not least, I presented my ideas about how to use control-flow analysis to detect code clones and plagiarism.

To summarise my main results in this thesis: I reviewed the topic related to duplicated code and I gave an abstract, general algorithm of the duplicated code detectors. Besides studying and analysing already published techniques, I described and successfully evaluated my own algorithm on open-source projects.

# Appendix A

## Used configurations

### A.1 Config A

| Property name        | Value                                |
|----------------------|--------------------------------------|
| Operating System     | Ubuntu 12.04.2 LTS 64bit             |
| Available Memory     | 32 GB                                |
| Free memory          | 20 GB                                |
| Total Swap           | 33 GB                                |
| Free Swap            | 33 GB                                |
| Num. of processors   | 48                                   |
| Model of a processor | Intel(R) Xeon(R) CPU E7530 @ 1.87GHz |
| Cache / processor    | 12 288 KB                            |
| Num. of cores/proc   | 6                                    |

Figure A.1: Properties of *Config A*

## A.2 Config B

| Property name        | Value                   |
|----------------------|-------------------------|
| Operating System     | Mac OS X 10.6.8 64bit   |
| Available Memory     | 4 GB                    |
| Free memory          | 2 GB                    |
| Total Swap           | 4 GB                    |
| Free Swap            | 4 GB                    |
| Num. of processors   | 1                       |
| Model of a processor | Intel Core i7 @ 2.8 GHz |
| Cache / processor    | 8 MB                    |
| Num. of cores/proc   | 8                       |

Figure A.2: Properties of *Config B*

# Appendix B

## Some interesting detected clones

These clones were found in Mnesia by the AST/metric based approach.

```
% First instance (found in mnesia_frag):
case ?catch_val(Var) of
    {'EXIT', Reason} -> mnesia_lib:other_val(Var,
        (cont.)Reason);
    Value -> Value
end

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

```
% First instance (found in mnesia_bup):
unlink(Master),
exit(Reason)

% Second instance (found in mnesia_monitor):
unlink(Mon),
exit(normal)
```

```
% First instance (found in mnesia_recover):
error("~p got unexpected info: ~p~n", [?MODULE, Msg]),
{noreply, State}

% Second instance (found in mnesia_monitor):
error("~p got unexpected call: ~p~n", [?MODULE, Msg]),
{noreply, State}
```

```

% First instance (found in mnesia_controller):
case whereis(?SERVER_NAME) of
    undefined ->
        {error, {node_not_running, node()}};
    Pid ->
        link(Pid),
        Res = gen_server:call(Pid, Msg, infinity),
        unlink(Pid),

        %% We get an exit signal if server dies
        receive
            {'EXIT', Pid, _Reason} ->
                {error, {node_not_running,
                    (cont.)node()}}
            after 0 ->
                Res
        end
    end
end

% Second instance (found in mnesia_monitor):
case whereis(?MODULE) of
    undefined ->
        {error, {node_not_running, node()}};
    Pid ->
        link(Pid),
        Res = gen_server:call(Pid, Msg, infinity),
        unlink(Pid),

        %% We get an exit signal if server dies
        receive
            {'EXIT', Pid, _Reason} ->
                {error, {node_not_running,
                    (cont.)node()}}
            after 0 ->
                Res
        end
    end
end
end

```

```

% First instance (found in mnesia_recover):
mnesia_lib:dbg_out("~p was ~p~n",[?MODULE, R]),
{stop, shutdown, State}

% Second instance (found in mnesia_monitor):
dbg_out("~p was ~p by supervisor~n",[?MODULE, R]),
{stop, R, State}

```

```

% First instance (found in mnesia_dumper):
Cs = mnesia_schema:list2cs(TabDef),
case InitBy of
    startup -> ignore;
    _ -> mnesia_controller:change_table_majority(Cs)
end,
insert_cstruct(Tid, Cs, true, InPlace, InitBy)

% Second instance (found in mnesia_dumper):
Cs = mnesia_schema:list2cs(TabDef),
case mnesia_lib:cs_to_storage_type(node(), Cs) of
    disc_copies ->
        open_dcl(Cs#cstruct.name);
    _ ->
        ignore
end,
insert_cstruct(Tid, Cs, true, InPlace, InitBy)

```

```

% First instance (found in mnesia_snmp_sup):
Flags = {simple_one_for_one, 0, timer:hours(24)},
MFA = {mnesia_snmp_hook, start, []},
Modules = [?MODULE, mnesia_snmp_hook, supervisor],
KillAfter = mnesia_kernel_sup:supervisor_timeout(timer:
    (cont.)seconds(3)),
Workers = [{?MODULE, MFA, transient, KillAfter, worker,
    (cont.)Modules}],
{ok, {Flags, Workers}}

% Second instance (found in mnesia_checkpoint_sup):
Flags = {simple_one_for_one, 0, timer:hours(24)},
MFA = {mnesia_checkpoint, start, []},
Modules = [?MODULE, mnesia_checkpoint, supervisor],
KillAfter = mnesia_kernel_sup:supervisor_timeout(timer:
    (cont.)seconds(3)),
Workers = [{?MODULE, MFA, transient, KillAfter, worker,
    (cont.)Modules}],
{ok, {Flags, Workers}}

```

```

% First instance (found in mnesia_frag):
InitFun = fun(FixedSpec) -> mnesia:dirty_sel_init(Node, Tab,
    (cont.)FixedSpec, Limit, Type) end

% Second instance (found in mnesia):
InitFun = fun(FixedSpec) -> dirty_sel_init(Where, Tab,
    (cont.)FixedSpec, NObjects, Type) end

```

```

% First instance (found in mnesia_controller):
case val({Tab, where_to_read}) of
  N -> mnesia_lib:set_remote_where_to_read(Tab);
  _ -> ignore
end

% Second instance (found in mnesia_controller):
case catch val({Tab , where_to_read}) of
  To ->
    mnesia_lib:set_remote_where_to_read(Tab);
  _ ->
    ignore
end

```

```

% First instance (found in mnesia_event):
Fun = fun(I) when I < 10 -> ["_0",I];
      (I) -> ["_",I]
end,
List = lists:append([Fun(I) || I <- Integers])

% Second instance (found in mnesia_lib):
Fun = fun(I) when I < 10 -> ["_0", I];
      (I) -> ["_", I]
end,
List = lists:append([Fun(I) || I <- Integers])

```

```

% First instance (found in mnesia_schema):
case lists:keysearch(Tab, 1, R#r.table_options) of
  {value, {Tab, OP}} ->
    OP;
  false ->
    R#r.default_op
end

% Second instance (found in mnesia_tm):
case lists:keysearch(Key, 1, Info) of
  {value, {_, Val}} ->
    Val;
  _ ->
    0
end

```

# List of Algorithms

|     |                                                                 |    |
|-----|-----------------------------------------------------------------|----|
| 0.1 | Algorithm Example . . . . .                                     | 6  |
| 1.1 | General duplicated code detector . . . . .                      | 10 |
| 3.1 | Algorithm of the alphabet . . . . .                             | 28 |
| 3.2 | Parallel algorithm of the initial clones detector . . . . .     | 37 |
| 3.3 | Filtering and trimming unit of the algorithm . . . . .          | 42 |
| 3.4 | Filtering and trimming unit of the one unit long clones . . . . | 42 |
| 3.5 | Filtering and trimming unit of the multi-unit long clones . . . | 43 |
| 3.6 | Determining the cloning level . . . . .                         | 52 |

# Bibliography

- [1] Erlang - ETS. <http://www.erlang.org/doc/man/ets.html>.
- [2] Erlang - Mnesia. <http://www.erlang.org/doc/man/mnesia.html>.
- [3] Erlang - timer module. <http://www.erlang.org/doc/man/timer.html>.
- [4] Erlang Programming Language. <http://erlang.org>.
- [5] Kyoto Cabinet: a straightforward implementation of DBM. <http://fallabs.com/kyotocabinet/>.
- [6] Moss Homepage. <http://theory.stanford.edu/~aiken/moss/>.
- [7] RefactorErl Homepage. <http://refactorerl.com>.
- [8] SimString - A fast and simple algorithm for approximate string matching/retrieval. <http://www.chokkan.org/software/simstring/index.html.en>.
- [9] Joe Armstrong. Programming Erlang: Software for a Concurrent World, 2007.
- [10] Brenda S. Baker. A program for identifying duplicated code. In *Computer Science and Statistics: Proc. Symp. on the Interface*, pages 49–57, March 1992.
- [11] 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.
- [12] 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.

- [13] 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.
- [14] 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.
- [15] 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.
- [16] Lee Raymond Dice. Measures of the Amount of Ecologic Association Between Species. *Ecology*, 26(3):297–302, July 1945.
- [17] David Eppstein. Subgraph isomorphism in planar graphs and related problems. In *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, SODA '95, pages 632–640, Philadelphia, PA, USA, 1995. Society for Industrial and Applied Mathematics.
- [18] Viktória Fördős and Melinda Tóth. *Identifying Code Clones with RefactorErl*. Submitted to the 13th Symposium on Programming Languages and Software Tools, 2013.
- [19] A.A. Fraenkel and Y.B. Hillel. *Foundations of Set Theory*. Foundations of mathematics. North-Holland Publishing Company, 1958.
- [20] Nils Göde and Rainer Koschke. Incremental Clone Detection. In Andreas Winter, Rudolf Ferenc, and Jens Knodel, editors, *CSMR*, pages 219–228. IEEE, 2009.
- [21] Dan Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, New York, NY, USA, 1997.
- [22] Andrew Hunt and David Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

- [23] Elmar Juergens and Nils Göde. 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.
- [24] 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.
- [25] 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, Komárno, Slovakia, July 2010.
- [26] Raghavan Komondoor and Susan Horwitz. Using Slicing to Identify Duplication in Source Code. In *Proceedings of the 8th International Symposium on Static Analysis*, SAS '01, pages 40–56, London, UK, UK, 2001. Springer-Verlag.
- [27] Jens Krinke. Identifying Similar Code with Program Dependence Graphs. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, WCRE '01, pages 301–310, Washington, DC, USA, 2001. IEEE Computer Society.
- [28] VI Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966.
- [29] 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.
- [30] Huiqing Li, Simon Thompson, György Orosz, and Melinda Tóth. Refactoring with wrangler, updated: data and process refactorings, and integration with eclipse. In *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, ERLANG '08, pages 61–72, New York, NY, USA, 2008. ACM.
- [31] 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 Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 20–20, Berkeley, CA, USA, 2004. USENIX Association.

- [32] Fowler Martin, Beck Kent, Brant John, Opdyke William, and Don Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [33] 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.
- [34] 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.
- [35] Naoaki Okazaki and Jun'ichi Tsujii. Simple and Efficient Algorithm for Approximate Dictionary Matching. In *Proceedings of the 23rd International Conference on Computational Linguistics (Coling 2010)*, pages 851–859, Beijing, China, August 2010.
- [36] Susan H. Randy Smith. Detecting and Measuring Similarity in Code Clones. *IWSC*, 2009.
- [37] 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.
- [38] Filip Van Rysselberghe and Serge Demeyer. Evaluating Clone Detection Techniques from a Refactoring Perspective. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering, ASE '04*, pages 336–339, Washington, DC, USA, 2004. IEEE Computer Society.
- [39] 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.
- [40] T. Sørensen. 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.
- [41] K.C. Thyholdt. *Lattice Boltzmann Simulations on a GPU: An optimization approach using C++ AMP*. PhD thesis, Norwegian University of Science and Technology, 2012.

- [42] M. Tóth, I. Bozó, and Z. Horváth. Applying the query language to support program comprehension. In *Proceeding of International Scientific Conference on Computer Science and Engineering, ISBN 978-80-8086-164-3*, pages 52–59, Stara Lubovna, Slovakia, September 2010.
- [43] 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.
- [44] I. Šimeček. CPU or GPU: that is the question. In *Proceedings of Workshop 2008*, volume 1, Prague, 2008. CTU.