ISSN 2277-3061

CLONING DETECTION ON THE BASE OF TEXTUAL SIMILARITY USING NEURAL AS A CLASSIFIER

KULBIR KAUR, RASBIR SINGH

M-TECH Scholar, RIMT, Mandi Gobingarh, Punjab, India
Asst. Prof. RIMT, Mandi Gobingarh, Punjab, India
erkulbirkaur@gmail.com, rasbir.rai@gmail.com

ABSTRACT

Cloning refer to copying the data from one place and using it on another place. This particular work has made the task of the developers easy as it is often that the developers copy the content of one work and paste it into another but this particular procedure violets the rule of copyrights. Copying data is feasible upto one extent but copying entire data set or data or code from one place to another is a serious issue .This paper focuses on the introduction of the code cloning and data cloning and the ways to prevent the cloning

KEYWORDS: code and data clone; methods of detection; copy paste forgery

Council for Innovative Research

Peer Review Research Publishing System

Journal: INTERNATIONAL JOURNAL OF COMPUTERS & TECHNOLOGY

Vol 13, No. 3

editor@cirworld.com

www.cirworld.com, www.ijctonline.com

ISSN 2277-3061

INTRODUCTION

Code cloning or the act of copying code fragments and making minor, non–functional alterations, is a well-known problem for evolving software systems leading to duplicated code fragments or code clones. Of course, the normal functioning of the system is not affected, but without countermeasures by the maintenance team, further development may become prohibitively expensive. Fortunately, the problem has been studied intensively and several techniques to both detect and remove duplicated code have been proposed in the literature. As far as removal of duplicated code is concerned, the state of the art proposes refactoring which is a technique to gradually improve the structure of (object-oriented) programs while preserving their external behaviour Extract Method which extracts portions of duplicated code in a separate method, is an example of a typical refactoring to remove duplicated code. However, quite often one must use a series of refactoring to actually remove duplicated code, as in Transform Conditionals into Polymorphism where duplicated conditional logic is refectories over the class hierarchy using polymorphism. With refactoring tools like the refactoring browser emerging from research laboratories into mainstream programming environments, refactoring is becoming a mature and widespread technique[1].

Concerning the detection of duplicated code, numerous techniques have been successfully applied on industrial systems. These techniques can be roughly classified into three categories.

1. String-based, i.e. the program is divided into a number of strings (typically lines) and these strings are compared against each other to find sequences of duplicated strings.

2. Token-based, i.e. a laxer tool divides the program into a stream of tokens and then searches for series of similar tokens.

3. parse–tree based, i.e., after building a complete parse-tree one performs pattern matching on the tree to search for similar sub–trees.[2]

On the first International Workshop on Detection of Software Clones, a number of research groups recently participated in a clone detection contest to compare the accuracy of different tools against a benchmark of programs containing known duplication. The results of this experiment are currently being analysed by the participants.

Despite all this progress, little is known about the most optimal application of a given clone detection technique during the maintenance process. For instance, which technique should one use in a problem assessment phase, when one suspects duplicated code but isn't sure how much and in which files? Or which technique works best in combination with a refactoring tool, which has to know the exact boundaries of the code segment to be refectories, including possible renaming of variables and parameters? To answer these questions, this paper compares three representative clone detection techniques —namely simple line matching, parameterized matching, and metric fingerprints— by means of five small to medium cases. The reported matches as well as the process are analysed with special interest in differences. Afterwards, our findings are interpreted in the context of a generic software maintenance process and some suggestions are made on the most optimal application of a given technique. The paper is structured as a comparative study, however due to the multiple aspects involved in the issue studied a more extensive experiment is necessary in the near future[3]. A brief overview of existing duplicated code detection techniques is given in section . The experimental set-up, including the questions and cases driving the experiment are discussed in section . The results of section are interpreted in section to evaluate where the given technique might fit into the software maintenance process. Finally[4], section summarises our findings in a conclusion.

Why Do Clones Occur?

Software clones appear for a variety of reasons:

- Code reuse by copying pre-existing idioms
- Coding styles
- Instantiations of definitional computations
- Failure to identify/use abstract data types
- Performance enhancement
- Accidents

State of the art software design has structured design processes, and formal reuse methods. Legacy code (and, alas, far too much of new code) is constructed by less structured means. In particular, a considerable amount of code[5] is or was produced by ad hoc reuse of existing code. Programmers intent on implementing new functionality find some code idiom that perform a computation nearly identical to the one desired, copy the idiom wholesale and then modify in place. Screen editors that universally have “copy” and “paste” functions hasten the ubiquity of this event.

In large systems, this method may even become a standard way to produce variant modules. When building device drivers for operating systems, much of the code is boilerplate, and only the part of the driver dealing with the device hardware needs to change. In such a context, it is commonplace for a device driver author to copy entirely an existing, well-known, trusted driver and simply modify it. While this is actually good reuse practice, it exacerbates the maintenance problem of fixing a bug found in the “trusted” driver by replicating its code (and reusing its bugs) over many new drivers. Sometimes a “style” for coding a regularly needed code fragment will arise, such as error reporting or user interface

displays. The fragment will purposely be copied to maintain the style. To the extent that the fragment consists only of parameters this is good practice. Often, however, the fragment unnecessarily contains considerably more knowledge of some program data structure, etc. It is also the case that many repeated computations (payroll tax, queue insertion, data structure access) are simple to the point of being definitional. As a consequence, even when copying is not used, a programmer may use a mental macro to write essentially the same code each time a definitional operation needs to be carried out. If the mental operation is frequent, he may even develop a regular style for coding it. Mental macros produce near-miss clones: the code is almost the same ignoring irrelevant order and variable names. Some clones are in fact complete duplicates of functions intended for use on another data structure of the same type; we have found many systems with poor copies of insertion sort on different arrays scattered around the code. Such clones are an indication that the data type operation should have been supported by reusing a library function rather than pasting a copy. Some clones exist for justifiable performance reasons. Systems with tight time constraints are often hand optimized

by replicating frequent computations, especially when a compiler does not offer in-lining of arbitrary expressions or computations.

Lastly, there are occasional code fragments that are just accidentally identical, but in fact are not clones. When investigated fully, such apparent clones just are not intended to carry out the same computation. Fortunately, as size goes up, the number of accidents of this type drops off dramatically.

Ignoring accidental clones, the presence of clones in code unnecessarily increases the mass of the code. This forces programmers to inspect more code than necessary, and consequently increases the cost of software maintenance. One could replace such clones by invocations of clone abstractions once the clones can be found, with potentially great savings.

Detection Techniques

The detection of code clones is a two phase process which consists of a transformation and a comparison phase. In the first phase, the source text is transformed into an internal format which allows the use of a more efficient comparison algorithm. During the succeeding comparison phase the actual matches are detected. Due to its central role, it is reasonable to classify detection techniques according to their internal format. This section gives an overview of the different techniques available for each category while selecting a representative for each category.

String Based

String based techniques use basic string transformation and comparison algorithms which makes them independent of programming languages.

Techniques in this category differ in underlying string comparison algorithm. Comparing calculated signatures per line, is one possibility to identify for matching substrings. Line matching, which comes in two variants, is an alternative which is selected as representative for this category because it uses general string manipulations.

Simple Line Matching

Simple Line Matching is the first variant of line matching in which both detection phases are straightforward. Only minor transformations using string manipulation operations, which can operate using no or very limited knowledge about possible language constructs, are applied. Typical transformations are the removal of empty lines and white spaces.

During comparison all lines are compared with each other using a string matching algorithm. This results in a large search space which is usually reduced using hashing buckets. Before comparing all the lines, they are hashed into one of n possible buckets. Afterwards all pairs in the same bucket are compared.

Duple is a Smalltalk tool which implements such a simple line matching technique however also a Java version is available

Parameterized Line Matching

It is another variant of line matching which detects both identical as well as similar code fragments. The idea is that since identifier–names and literals are likely to change when cloning a code fragment, they can be considered as changeable parameters. Therefore, similar fragments which differ only in the naming of these parameters, are allowed.

To enable such parameterization, the set of transformations is extended with an additional transformation that replaces all identifiers and literals with one, common identifier symbol like ”$”. Due to this additional substitution, the comparison becomes independent of the parameters. Therefore no additional changes are necessary to the comparison algorithm itself. Parameterized line matching is discussed in.

Token Based

Token based techniques use a more sophisticated transformation algorithm by constructing a token stream from the source code, hence require a laxer. The presence of such tokens makes it possible to use improved comparison algorithms.

Next is to parameterized matching with suffix trees, which acts as representative, we include in this category because it also transforms the source code in a token-structure which is afterwards matched. The latter tries to remove much more detail by summarising non interesting code fragments.

Parameterized Matching With Suffix Trees

It consists of three consecutive steps manipulating a suffix tree as internal representation. In the first step, a lexical analyser passes over the source text transforming identifiers and literals in parameter symbols, while the typographical structure of each line is encoded in a non-parameter symbol. One symbol always refers to the same identifier, literal or structure. The result of this first step is a parameterized string or p-string. Once the p-string is constructed, a criterion to decide whether two sequences in this p-string are a parameterized match or not is necessary. Two strings are a parameterized match if one can be transformed into the other by applying a one-to-one mapping renaming the parameter symbols[6]. An additional encoding prep(S) of the parameter symbols helps us verifying this criterion. In this encoding, each first occurrence of a parameter symbol is replaced by a 0. All later occurrences are replaced by the distance since the previous occurrence of the same symbol. Thus, when two sequences have the same encoding, they are the same except for a systematic renaming of the parameter symbols.

After the lexical analysis, a data structure called a parameterized suffix tree (p-suffix tree) is built for the string. A p-suffix tree is a generalisation of the suffix tree data structure which contains the prep()-encoding of every suffix of a P-string. Concatenating the labels of the arcs on the path from the root to the leaf yields the prep( )-encoding of one suffix. The use of a suffix tree allows a more efficient detection of maximal, parameterized matches.

All that is left for the last step, is to find maximal paths in the p-suffix tree that are longer than a predefined character length. Parameterized matching using suffix trees was introduced in with Dup as implementation example.

Parse tree Based

Parse tree based techniques use a heavyweight transformation algorithm, i.e. the construction of a parse tree. Because of the richness of this structure, it is possible to try various comparison algorithms as well.

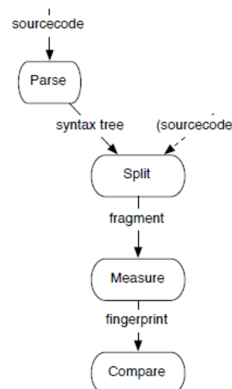

Figure 1. Detection steps for the metric fingerprint technique

The representing technique differs from in that the latter uses sub–tree matching on the syntax tree.

Metric Fingerprints

This builds on the idea that you can characterise a code fragment using a set of numbers. These numbers are measurements which identify the functional structure of the fragment and sometimes the layout. The metric fingerprint technique can be divided in five steps, each with a well-defined task. However the algorithm behind each task may differ between implementations. Figure 1f shows the basic steps in the detection process.

Before we can characterise the functional structure of a code fragment with numbers, it's wise to transform the source code into a representation that allows us to calculate such measurements efficiently[7]. This transformation job is done using a parser which builds the syntax tree of the source code. After parsing we end up with one large syntax tree. This tree is then split into interesting fragments. The choice of the type of fragments used is difficult because it affects the detection results. Most of the time, however, method and scope blocks are used as fragments since they are easily extracted from a syntax tree. Afterwards the fragments are characterised through a set of measurements by measuring the values for a set of metrics, chosen in advance. This set of metrics can differ between various implementations, but most of the time it specifies functional properties. However there are implementations in which layout metrics are used as well. Cyclamate complexity, function points, expression complexity (functional) and lines of code (layout) are examples of possible measures.

Finally, these sets of numbers are compared to each other. Depending on the implementation, algorithms with different levels of sophistication or power may be used. One possible approach calculates the Euclidean distance between each

pair of fingerprints, considering fragments within zero distance as clones. Describe a possible implementation of metric fingerprints.

REFERENCE

[1]. Filip Van Rysselberghe," Evaluating Clone Detection Techniques", Lab On Re-Engineering University Of Antwerp Middelheimlaan 1, B 2020 Antwerpen.

[2]. N. Davey," The Development of a Software Clone Detector", London Road, Harlow, Essex, UK, CM17 9NA P.C.Barson@bnr.co.uk

[3]. Rainer Koschke, " Clone Detection Using Abstract Syntax Suffix Trees" University of Bremen, Germany http://www.informatik.uni-bremen.de/st/

{koschke,rfalke,saint}@informatik.uni bremen.de

[4]. Yingnong Dang," Code Clone Detection Experience at Microsoft" IWSC'11, May 23, 2011, Waikiki, Honolulu, HI, USA Copyright 2011 ACM 978-1-4503-0588-4/11/05 ...$10.00.

[5]. Dr. Siobhán North," Retrieving Software Component using Clone Detection and Program Slicing", Memorandum CS-07-03 February 2007

[6]. Mitali, "An Analytical Review on the Techniques Opted For The Detection of Cloning In Spread Sheets", International Journal of Innovative Technology and Exploring Engineering (IJITEE) ISSN: 2278-3075, Volume-3, Issue-6, November 2013

[7]. Ripon K. Saha," Detection and Analysis of Near-Miss Clone Genealogies", FASE '11/ETAPS '11, pages 432446, Saarbr• ucken, Germany, 2011.