

Interoperation Potential: Integration of Code-Clone Detection Methods with Other Analysis Methods

Toshihiro Kamiya

Optimized Design Research Team, Center for Service Research
National Institute of Advanced Industrial Science and Technology
Akihabara Dai Bldg. 1-18-13 Sotokanda, Chiyoda-ku, Tokyo, 101-0021, Japan
t-kamiya@aist.go.jp

Abstract

This (extended) position paper describes a future perspective of code-clone detection and analysis methods, in which code-clone detection and analysis methods are interoperated, or integrated with various kinds of analysis methods, in order to open new possibilities of research and applications.

1. Introduction

We know that code-clone detection and analysis have unique applications, such as the evaluation of products for refactoring, or detection of plagiarism. However, code-clone detection and analysis go far beyond that, once we regard the detection and analysis methods as fundamental techniques of reverse engineering. This perspective will open new possibilities of applications both in research and industrial fields, in which code-clone detection and analysis methods are used in an interoperated or integrated way with other analysis methods.

This paper introduces two types of ongoing research for this direction; the first one develops a new application by combination of code-clone detection and frequent itemset mining techniques. The second uses dependency analysis among modules as an enhancement of code-clone analysis.

2. Code-clone detection × frequent itemset mining

In this research, a code-clone detection method is combined with a frequent itemset mining technique [Agrawal93] in order to implement a tool to detect name changes between versions of a software product (see a paper [Kamiya08] for detail).

In the software maintenance process, names of functions, classes, methods, etc. are sometimes changed to improve the consistency of names and thus, the maintainability of the product, or, to improve API sets by replacing obsolete functions with newer ones. In some cases, these changes may not be fully documented. A tool for finding name changes from source code in an automatic way will be helpful especially for developers in distributed development processes, who routinely write, read, or review these documents.

Consider gapped clones between two versions of a source code of a product (Fig. 1). Here, a *gapped clone* is a pair of the code fragments that consist of almost the same code (called *matches*), but also have some differences (called *gaps*). When we extract gapped clone pairs between versions, a code fragment is taken from the older version (say, f^-), and the other code fragments from the newer version (say, f^+). The gaps will be the differences between the two versions. For a given name (of a function, a class, a method, etc.), if the name is changed, then the older name will be found

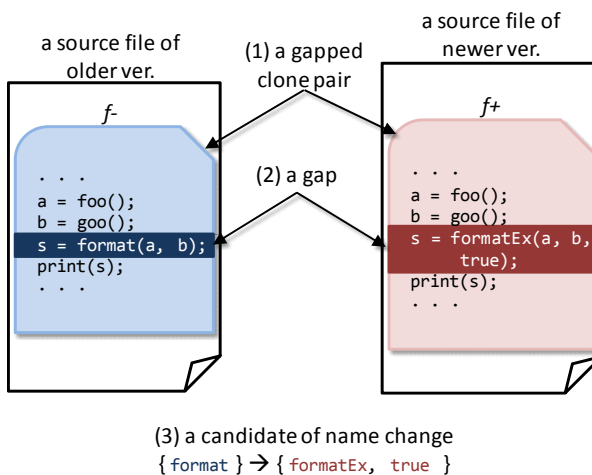


Figure 1. Gapped clone pair and name change

in a gap of f^- , and the newer name will be in a gap of f^+ , while unchanged names will be found in a match of both f^- and f^+ .

A tool named *Vaci* (<http://www.ccfinder.net/vaci/>) has been developed to test this assumption. First, this tool extracts gapped clones between two versions of source codes. Next, the tool finds the cases where a *name change* (that is, a pair of a set of the old names and a set of the newer names) is observed frequently. In other words, the name change is found in many gapped clones (the current development version of the tool uses a frequent itemset mining algorithm [Agrawal94] for this). Lastly, the tool scores each name change in terms of total length of adjacent matches by multiplying how many times the name change is observed by how long the matched texts appear before and after the gap, and prints out the name changes in order of their score.

In a tool demo [Kamiya08], we showed examples in which the early implementation of the tool was applied to some open-source products. The current development version can compare two versions of Linux kernels of 19k+ files each.

3. Code-clone analysis × dependency analysis between modules

This approach classifies code clones by dependencies between modules (not published yet).

Some kinds of libraries require ‘idiom’ (or ‘pattern’) code to call the functions of the libraries, and sometimes ‘boilerplate’ or ‘skeleton’ codes are provided to developers, to make it easy to write such

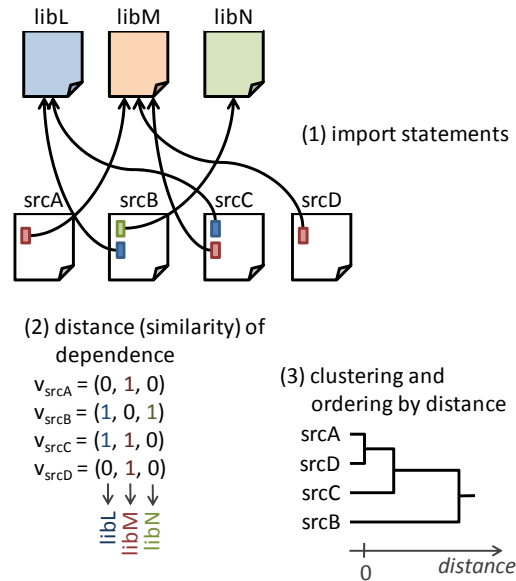


Figure 2. Imports, dependency by imports, and clustering by dependency

codes in a copy-paste-and-modify manner. (The example idioms are: a code for GUI to display a window or buttons on a screen, or a code for the database, to connect a database and to execute a query or to do a roll-back if needed.) As a result, these code fragments become code clones (called *idiom* code clones). If we can automatically label each of these code fragments as an “idiom for library x ”, such detailed classification will be helpful in the analysis. In some cases, idiom code clones will not be regarded as a target of analysis. In

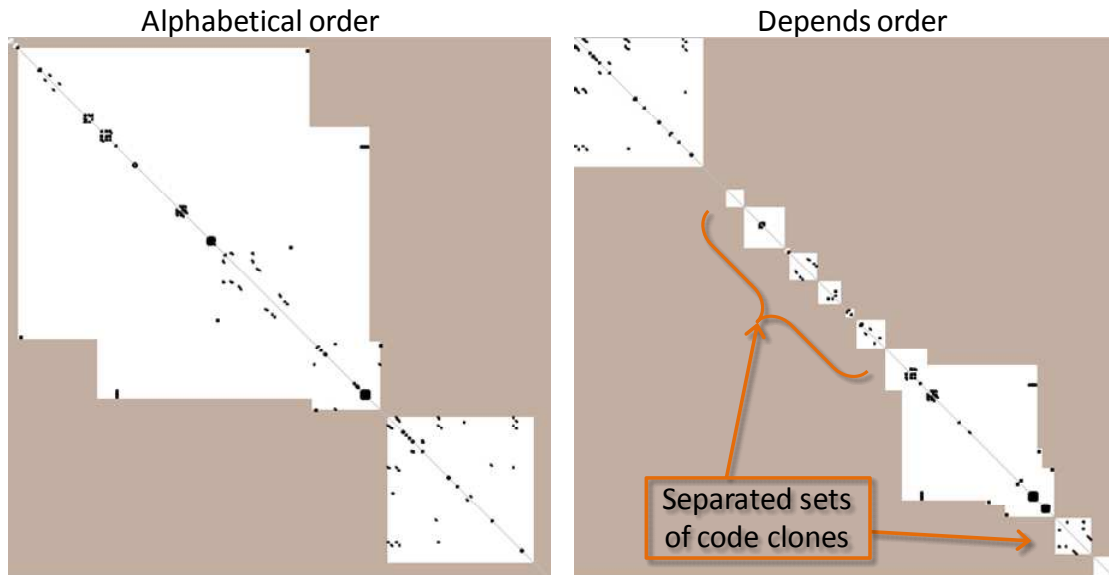


Figure 3. A scatter plot of alphabetical order vs. depends order

other cases, the idiom code clones are the primal target of analysis.

When a module m imports another module r (the importing command/statement varies in programming languages, “import”, “include”, “require”, etc., but they essentially mean reusing functionalities from other modules), let call m depends on r . If an idiom is used to call r 's functions, the calling codes will be code clones. If we can classify modules of a software product in terms of dependency, that is, classify each module m by a module r that is imported by m , then idiom code clones related to r will appear within code fragments of modules of the classification of r .

A tool named *sbyd* (sort source files by dependency) provides a view to explore this assumption. Fig. 2 illustrates how the tool works. First, the tool parses each of given source files to find import statements and makes a list of imported (depending) source files for the source file. Next, the tool performs a clustering of source files based on the similarity between the lists, and makes the result into a dendrogram (a kind of tree, in which each leaf represents a source file), and prints out names of the source files in the order of the leaves (call it *depends order*). In the depends order, the more similar import statements appear in two source files, the nearer the two files appears in the order. We can compare a scatter plot of code clones in which source files are aligned along alphabetical order and a scatter plot of depends order. Idiom code clones are expected to appear more closely to the main diagonal line in the latter scatter plot because the library related to the idiom collects the source files that include the idiom code clone.

In a preliminary experiment where the tool is applied to open-source products, some code clones were separated like islands in a scatter plot (Fig. 3).

4. Related work

Several research groups are working on code clones across versions [Adar07] [Chevalier07] [Lozano08]. Some people are working on analysis of relations between code clones and the architectural structure of software products [Kasper05] [MingJiang06]. Yoshida et al. [Yoshida05] attempt to classify code clones with class hierarchy in object-oriented programs. Ueda et al. [Ueda02] tried re-ordering source files in a scatter plot with code clones, which are included in the files. Manual classification of source code is used in some code-clone analysis research [Baxter98] [Monden02]. Examples of using frequent itemset mining as a clone detection algorithm are found in [Basit06] [Yoshida08].

The tool *Vaci* (introduced in Section 2) detects name changes by analyzing the use of names, that is, code fragments where the target names are used (referred). On the other hand, there is another approach to name changes, called *origin analysis*, which analyzes the definition of names, that is, code fragments where the target names are defined. Several research groups are actively working on origin analysis [Demeyer00] [Dig06] [Godfrey05] [Kim05] [Weiβgerber06]. Godfrey et al. [Godfrey02] points out that a kind of dependency analysis can be used together with origin analysis. Generally speaking, such refactoring can be detected not only by similarity in code, but also by difference of code. An approach by [VanRysseberghe06] called, *Code movement*, extracts refactoring by analyzing code movement.

References

- [Adar07] Eytan Adar and Miryung Kim, “SoftGUESS: Visualization and Exploration of Code Clones in Context”, Proc. the 29th IEEE Int’l Conference on Software Engineering (ICSE 2007), pp. 762-766 (2007).
- [Agrawal93] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami, “Mining associations between sets of items in massive databases”, Proc. ACM SIGMOD Int’l Conference on Management of Data, pp. 207-216 (1993).
- [Agrawal94] Rakesh Agrawal and Ramakrishnan Srikant. “Fast algorithms for mining association rules”, Proc. the 20th Int’l Conference on Very Large Databases (VLDB’94), pp. 487-499 (1994).
- [Basit07] Hamid Abdul Basit, Stan Jarzabek, “Detecting higher-level similarity patterns in programs”, Proc. 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC-FSE’05), pp. 156-165 (2005).
- [Baxter98] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant’Anna, and Lorraine Bier, “Clone Detection Using Abstract Syntax Trees”, Proc. the 14th IEEE Int’l Conference on Software Maintenance (ICSM’98), pp. 368-377 (1998).
- [Chevalier07] Fanny Chevalier, David Auber, and Alexandru Telea, “Structural Analysis and Visualization of C++ Code Evolution using Syntax Trees”, Proc. the 9th Int’l Workshop on Principles of Software Evolution (IWPSE 2007), pp 90-97 (2007).
- [Demeyer00] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz, “Finding refactorings via change metrics”, Proc. 15th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2000) pp. 166-177 (2000).

[Dig06] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson, "Automated Detection of Refactorings in Evolving Components", Proc. European Conference on Object-Oriented Programming (ECOOP'06), pp. 404-428 (2006).

[Godfrey05] Michael W. Godfrey and Lijie Zou, "Using Origin Analysis to Detect Merging and Splitting of Source Code Entities", IEEE Transactions on Software Engineering, vol. 31, no. 2, pp. 166-181 (2005).

[Godfrey02] Michael Godfrey and Qiang Tu, "Tracking structural evolution using origin analysis", Proc. IPSJ SIGSE/ACM SIGSOFT 5th International Workshop on Principles of Software Evolution (IWPSE 2002), pp. 117-119 (2002).

[Kamiya08] Toshihiro Kamiya, "Variation Analysis of Context-Sharing Identifiers with Code Clone", Proc. 24th IEEE Int'l Conference on Software Maintenance (ICSM 2008), pp. 464-465 (2008).

[Kapsner05] Cory Kapsner and Michael W. Godfrey, "Improved Tool Support for the Investigation of Duplication in Software", Proceedings of the 21st IEEE Int'l Conference on Software Maintenance (ICSM 2005), p.305-314 (2005).

[Kim05] Sunghun Kim, Kai Pan, and James E. Whitehead, "When Functions Change Their Names: Automatic Detection of Origin Relationships", Proc. 12th Working Conference on Reverse Engineering (WCRE 2005), pp. 143-152 (2005).

[Lozano08] Angela Lozano and Michel Wermelinger, "Assessing the effect of clones on changeability", Proc. 24th IEEE Int'l Conference on Software Maintenance (ICSM 2008), pp. 227-236 (2008).

[Monden02] Akito Monden, Daikai Nakae, Toshihiro Kamiya, Shin-ichi Sato, and Ken-ichi Matsumoto, "Software Quality Analysis by Code Clones in Industrial Legacy Software", Proc. the 8th IEEE Int'l Software Metrics Symposium (METRICS 2002), pp.87-94 (2002).

[MingJiang06] Zhen Ming Jiang, Ahmed E. Hassan, Richard C. Holt, Visualizing Clone Cohesion and Coupling, Proc. 13th Asia Pacific Software Engineering Conference (APSEC 06), pp. 467-476 (2006).

[Ueda02] Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue, "Gemini: Maintenance Support Environment Based on Code Clone Analysis", Proc. the 8th IEEE Symposium on Software Metrics (METRICS 2002), pp.67-76 (2002).

[VanRysselberghe06] Filip Van Rysselberghe, Matthias Rieger, and Serge Demeyer, "Detecting move operations in versioning information", Proc. 10th European Conference on Software Maintenance and Reengineering (CSMR 2006), pp. 271-278 (2006).

[Weißgerber06] Peter Weißgerber and Stephan Diehl, "Identifying Refactorings from Source-Code Changes", Proc. 21st IEEE/ACM International Conference on Automated Software Software Engineering (ASE 2006), pp. 231-240 (2006).

[Yoshida05] Norihiro Yoshida, Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue, "On Refactoring Support Based on Code Clone Dependency Relation", Proc. the 11th IEEE Int'l Software Metrics Symposium (METRICS 2005), p. 16 (2005).

[Yoshida08] Norihiro Yoshida, Takashi Ishio, Makoto Matsushita, and Katsuro Inoue, "Retrieving Similar Code Fragments based on Identifier Similarity for Defect Detection", Proc. Int'l Workshop on Defects in Large Software Systems (DEFECTS 2008), pp.41-42 (2008).