Electronic Communications of the EASST
Volume 63 (2014)

# Proceedings of the
# Eighth International Workshop on
# Software Clones
# (IWSC 2014)

## Toward a Code-Clone Search through the Entire Lifecycle of a Software Product

— Position Paper —

Toshihiro Kamiya

7 pages

Guest Editors: Nils Göde, Yoshiki Higo, Rainer Koschke
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

# Toward a Code-Clone Search through the Entire Lifecycle of a Software Product

**Toshihiro Kamiya**[1]

[1] kamiya@fun.ac.jp
Department of Media Architecture,
School of Systems Information Science,
Future University Hakodate
116-2 Kamedanakano-cho, Hakodate, Hokkaido, Japan 041-8655

**Abstract:** This paper presents a clone-detection method/tool currently under development. This tool is useful as a code-clone search through the entire lifecycle of a software product; The tool searches code examples and analyzes of code clones in both preventive and postmortem ways[LRHK10]. The approach is based on a sequence equivalence on execution paths[Kam13] and extends the equivalence to include gaps, thus type-3[BKA+07] clone detection. Each of the detected clones is a sub-sequence of an execution path of a given program, in other words, a set of code fragments of multiple procedures (methods) which can be executed in a run of the program. The approach is relaxed in terms of adaptability to incomplete (not-yet-finished) code, but also makes use of concrete information such as types (including hierarchy) and dynamic dispatch when such information is available.

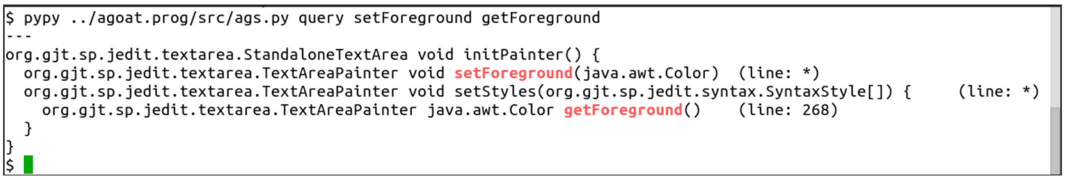
**Keywords:** Code Clone, Code Search, Postmortem Code-Clone Detection, Preventive Code-Clone Detection

## 1 Introduction

This paper presents a clone-detection method/tool under development. The tool will support searching a similar code at each stage of the entire lifecycle of a software product, that is, (1) a code example search will be executed even when no (or a very small volume of) code of a product has been written, (2) preventively (or instant)[LRHK10], i.e., in an automatic code-clone search to encourage code reuse by searching code fragments similar to the code written by a developer, and (3) postmortem[LRHK10]; i.e., in the code-clone search for a refactoring[BYM+98][HKKI04][ZR11] or a consistent code modification[Kri07]. A model of these stages is explained in Sec. 2.

The proposed method/tool is a kind of searching where both an input (query) and an output (search results) are code fragments. As for such kind of tools, CodeBroker[YFR00] and Sniff[CJS09] search code fragments with a similarity of identifiers and text in comments. A tool Strathcona[HM06] searches code fragments with a similarity of types (including the types that are related with the query code in terms of inheritance), or with methods directly called in the query code fragment.

The essence of the search algorithm first identifies candidate code fragments with a similarity of *names/values* of types, literals, and direct- or indirect-call of procedures, and then filters out



```
$ pypy ../agoat.prog/src/ags.py query setForeground getForeground
---
org.gjt.sp.jedit.textarea.StandaloneTextArea void initPainter() {
  org.gjt.sp.jedit.textarea.TextAreaPainter void setForeground(java.awt.Color)  (line: *)
  org.gjt.sp.jedit.textarea.TextAreaPainter void setStyles(org.gjt.sp.jedit.syntax.SyntaxStyle[]) {     (line: *)
    org.gjt.sp.jedit.textarea.TextAreaPainter java.awt.Color getForeground()    (line: 268)
  }
}
$ 
```

Figure 1: Searching keywords in source code of jEdit

(or sorts) the candidate code fragments with a similarity as a sequence on an execution path; thus, a kind of *structure*. Here an execution path represents a set of the code fragments that appear in distinct procedures, but are connected with procedure calls[Kam13], and consequently, a type-3[BKA$^+$07] (including arbitrary-size gaps) clone.

A picture of the proposed search method is explained in Sec. 3. ( The implementation is built on top of the code search method/tool[Kam14]. Figure 1 shows a sample run of the code search tool, where two keywords `setForeground` and `getForeground` were searched. Here the former was called directly in a method body of `initPainter` and the latter called indirectly, via a method `setStyles`. In this latter case, the tool found such an example code of code fragments from distinct source files but connected by an execution path. )

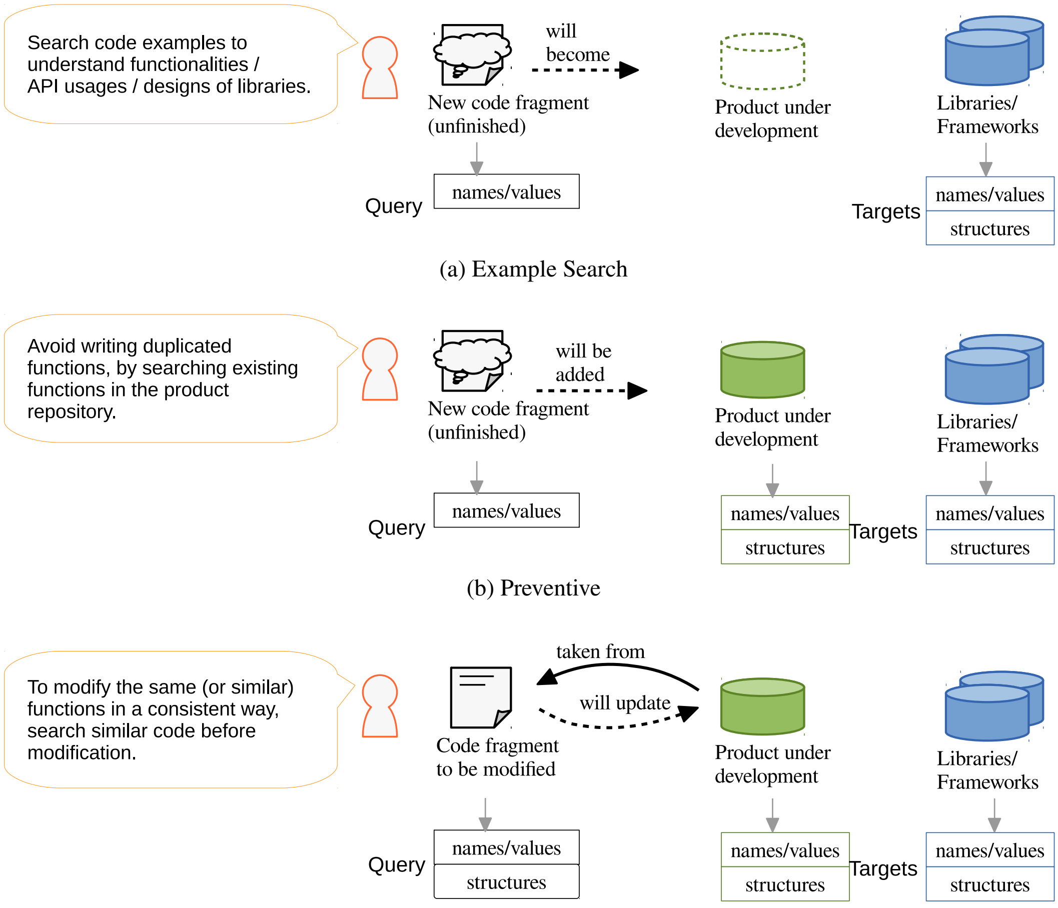
Note that this approach is opposite to approaches that find similarity in structures in terms of software architectures[BJ05][TEB12], data flow[DHJ$^+$08][PNN$^+$09][ACD$^+$12], PDG(program dependence graph)[Kri01][GJS08], AST(abstract syntax tree)[BYM$^+$98][KFF06][JMSG07] [TH12], and sequences of such as token[LHMI07][GK09], line [Bak12][DRD99], or byte code[DG10]. These approaches regard names/values as parameters of a structure, while the proposed approach finds similarity of names/values before structures.

## 2 Development Stages and Available Information

Figure 2 shows a model of the development stages. In an early stage of software development, a code search tool perhaps uses (as a query) only names/values of a code edited by a developer, because structures of such unfinished code will be incorrect or unstable. In the later stages, if a code being edited is stable enough and modifications on it are relatively few and small ones, its structures will be stable and a code search tool will use the structures as a query in addition to the names/values of the code.

As for search targets, in an early stage of development (or when a product introduces a totally new feature and the product's code base does not include any reference code in a practical sense), the tool is not able to use a product's code base as a target. In such a case, the tool searches in code bodies of libraries (or frameworks) to be reused. In the later stage, the tool also searches in a code body of the product under development.

An actual product would be a mix of these stages, especially in an incremental development process; some source files are matured and the others are immature or not-yet finished. In such a case, the proposed method/tool will be applicable in a seamless way, without develpers' caring about which source files are at which stage.


Search code examples to understand functionalities / API usages / designs of libraries.
will become
New code fragment (unfinished)
Product under development
Libraries/ Frameworks
Query
names/values
Targets
names/values
structures
Avoid writing duplicated functions, by searching existing functions in the product repository.
will be added
New code fragment (unfinished)
Product under development
Libraries/ Frameworks
Query
names/values
names/values
structures
Targets
names/values
structures
To modify the same (or similar) functions in a consistent way, search similar code before modification.
taken from
will update
Code fragment to be modified
Product under development
Libraries/ Frameworks
Query
names/values
structures
names/values
structures
Targets
names/values
structures

(a) Example Search

(b) Preventive

(c) Postmortem

Figure 2: Model of development stages and clone searching

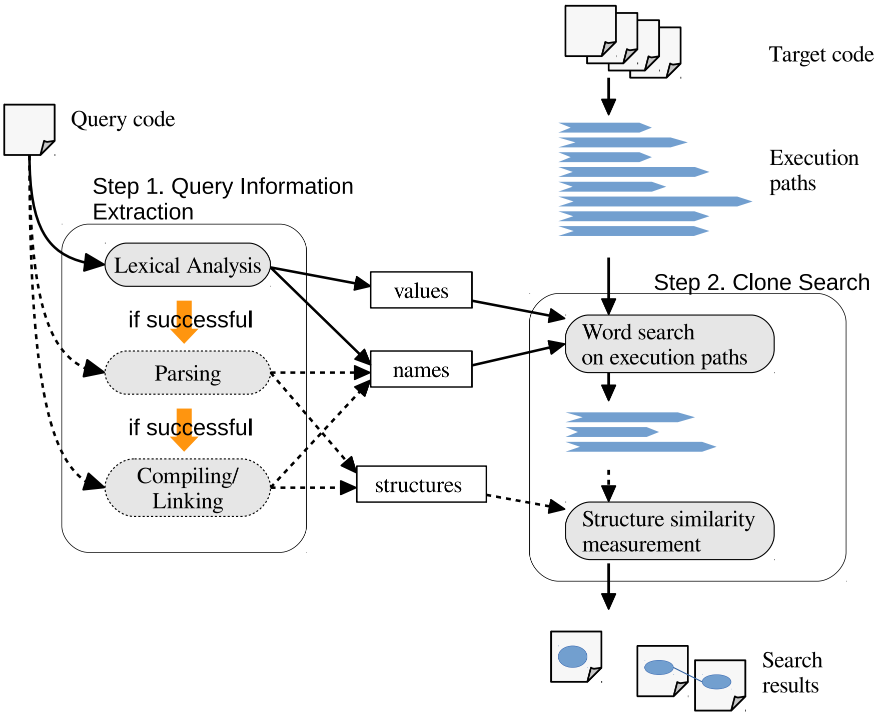


Figure 3: Steps of clone-search method

# 3 Steps of the Searching Method

The proposed clone-code search consists of two steps. Figure 3 shows a overall architecture of Searching Methods including these steps.

## 3.1 Step 1. Query Information Extraction

For better precision of search results, more detailed information extraction is required; the tool has to extract names/values and structures (if possible) from a query code. At the same time, for applicability to a product at any stage, robustness is required; the tool has to be applicable to a query code incomplete in terms of semantic and syntactic structures. To balance the precision and the robustness, the tool at first tries to compile (and link) the code, then (if not successful) tries parsing the code, and tries lexical analysis at the end.

Table 1 shows an information extracted in each of these trials. If the code is successfully compiled and linked, as shown Compilation/Linking column in the table, extract names/values and structures including not only types, literals, and directly-called procedures the developer explicitly described in the code, but also types of "intermediate" values in expressions (such as a return value of a method call, which is passed to another method call as an argument without being assigned to a variable) or indirectly-called procedures. On the other hand, if the code is not finished yet and incorrect in terms of grammar of a programming language (e.g., unbalanced { }), the tool can extract only names (of something) or literals from such code fragment, as shown in

Table 1: Information extracted from a query code

| | Lexical Analysis | Parsing | Compilation/Linking |
|---|---|---|---|
| Values | • Literals (of string, integer, boolean, etc.) | ← | ← |
| Names | • Identifiers which may be either variables, functions/methods, or types | • Variable names<br>• Function/method names<br>• Type names appearing in declarations | ←<br>• Type names of intermediate values |
| Structures | | • Branches<br>• Direct calls (detected with a inner-method analysis) | ←<br>• Indirect calls with a dynamic dispatch (inter-method analysis) |

*The "←" represents "the same items to the left cell".*

the Lexical Analysis column in the table.

### 3.2 Step 2. Clone Search

In [Kam14], a data structure named an And/Or/Call graph was introduced to represent (both inner- and inter-method) execution paths of a given program in a compact form. The code-search algorithm finds the sub-graphs that include all keywords (of types, string literals, method signatures) and convert the sub-graphs into execution paths. The planned code-clone search algorithm will be an extension of the above algorithm in the following ways: (1) finding execution paths including many keywords (not all keywords are necessary) and (2) filtering or ranking execution paths with similarity of structures, in terms of order, distance, and frequency of appearances of these names/values on each execution path.

## 4 Summary

The paper has presented a code-clone search tool/method usable through the lifecycle of a software product. In such a development process, available information differs between development stages and the tool design should maximize applicability and precision at these stages.

**Acknowledgements:** This work was supported by JSPS KAKENHI Grant Number 24650013.

## Bibliography

[ACD+12] M. Alalfi, J. Cordy, T. Dean, M. Stephan, A. Stevenson. Near-Miss Model Clone Detection for Simulink Models. In *Proc. IWSC 2012*. Pp. 79–79. 2012.

[Bak12] B. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *Proc. WCRE 1995*. Pp. 86–95. 2012.

[BJ05] H. Basit, S. Jarzabek. Detecting Higher-level Similarity Patterns in Programs. In *Proc. FSE 2005*. Pp. 156–165. 2005.

[BKA$^+$07] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE TSE* 33(9):577–591, 2007.

[BYM$^+$98] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proc. ICSM 1998*. Pp. 368–377. 1998.

[CJS09] S. Chatterjee, S. Juvekar, K. Sen. Sniff: A Search Engine for Java Using Free-Form Queries. In *Proc. FASE 2009, LNCS 5503*. Pp. 385–400. 2009.

[DG10] I. Davis, M. Godfrey. From Whence It Came: Detecting Source Code Clones by Analyzing Assembler. In *Proc. WCRE 2010*. Pp. 242–246. 2010.

[DHJ$^+$08] F. Deißenböck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J.-F. Girard, S. Teuchert. Clone Detection in Automotive Model-Based Development. In *Proc. ICSE 2008*. Pp. 603–612. 2008.

[DRD99] S. Ducasse, M. Rieger, S. Demeyer. A Language Independent Approach for Detecting Duplicated Code. In *Proc. ICSE 1999*. Pp. 109–118. 1999.

[GJS08] M. Gabel, L. Jiang, Z. Su. Scalable Detection of Semantic Clones. In *Proc. ICSE 2008*. Pp. 321–330. 2008.

[GK09] N. Göde, R. Koschke. Incremental Clone Detection. In *Proc. CSMR 2009*. Pp. 219–228. 2009.

[HKKI04] Y. Higo, T. Kamiya, S. Kusumoto, K. Inoue. Refactoring Support Based on Code Clone Analysis. In *Proc. PROFES 2004, LNCS 3009*. Pp. 220–233. 2004.

[HM06] R. Holmes, G. Murphy. Approximate Structural Context Matching: An Approach to Recommend Relevant Examples. *IEEE TSE* 32(12):952–970, 2006.

[JMSG07] L. Jiang, G. Misherghi, Z. Su, S. Glondu. Deckard: Scalable and Accurate Tree-Based Detection of Code Clones. In *Proc. ICSE 2007*. Pp. 96–105. 2007.

[Kam13] T. Kamiya. Agec: An Execution-Semantic Clone Detection Tool. In *Proc. ICPC 2013*. Pp. 227–229. 2013.

[Kam14] T. Kamiya. An Algorithm for Keyword Search on an Execution Path. In *Proc. CSMR-WCRE 2014 (to appear)*. 2014.

[KFF06] R. Koschke, R. Falke, P. Frenzel. Clone Detection Using Abstract Syntax Suffix Trees. In *Proc. WCRE 2006*. Pp. 253–262. 2006.

[Kri01] J. Krinke. Identifying Similar Code with Program Dependence Graphs. In *Proc. WCRE 2001*. Pp. 301–309. 2001.

[Kri07] J. Krinke. A Study of Consistent and Inconsistent Changes to Code Clones. In *Proc. WCRE 2007*. Pp. 170–178. 2007.

[LHMI07] S. Livieri, Y. Higo, M. Matsushita, K. Inoue. Very-large Scale Code Clone Analysis and Visualization of Open Source Programs Using Distributed CCFinder: D-CCFinder. In *Proc. ICSE 2007*. Pp. 106–115. 2007.

[LRHK10] M.-W. Lee, J.-W. Roh, S. w. Hwang, S. Kim. A Instant Code Clone Search. In *Proc. FSE 2010*. Pp. 167–176. 2010.

[PNN$^+$09] N. Pham, H. Nguyen, T. Nguyen, J. Al-Kofahi, T. Nguyen. Complete and Accurate Clone Detection in Graph-Based Models. In *Proc. ICSE 2009*. Pp. 276–286. 2009.

[TEB12] U. Tekin, U. Erdemir, F. Buzluca. Mining Object-Oriented Design Models for Detecting Identical Design Structures. In *Proc. IWSC 2012*. Pp. 43–49. 2012.

[TH12] M. Thomsen, F. Henglein. Clone Detection Using Rolling Hashing, Suffix Trees and Dagification: A Case Study. In *Proc. IWSC 2012*. Pp. 22–28. 2012.

[YFR00] Y. Ye, G. Fischer, B. Reeves. Integrating Active Information Delivery and Reuse Repository Systems. In *Proc. FSE 2000*. Pp. 60–68. 2000.

[ZR11] M. Zibran, C. Roy. A Constraint Programming Approach to Conflict-Aware Optimal Scheduling of Prioritized Code Clone Refactoring. In *Proc. SCAM 2011*. Pp. 105–114. 2011.