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

Toshihiro Kamiya

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

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[1][2], and consequently, a type-3 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[3]. 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[4][5][6], data flow[7][8][9][10], PDG(program dependence graph)[11][12], AST(abstract syntax tree)[13][14][15], and sequences of such as token[16][17], line[18], or byte code[19]. 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 developers’ caring about which source files are at which stage.
Figure 2: Model of development stages and clone searching
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

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

*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


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


