An Execution-Semantic and Content-and-Context-Based Code-Clone Detection and Analysis

Toshihiro Kamiya
Department of Media Architecture, School of Systems Information Science, Future University Hakodate, Japan.
Email: kamiya@fun.ac.jp

Abstract—This paper presents a code-clone detection and its analysis method, based on an execution-semantic and arbitrary-granularity model of code fragments. The principal goal of introducing the proposed detection method is to provide a code-clone detection method suitable for programming languages, where software developers can define their own “control sentences” with such as lambda or lazy evaluation. Code clones detected with the proposed method are a kind of type-3 clone, where code fragments exist across boundaries of procedures or modules. The model also seems useful as clone metrics (for a clone triage) based on the contents and contexts of code fragments in a clone class and extensible to a unified method of code-clone detection and code search. This paper introduces an execution-semantic and content-and-context based code clone, describes its definition, a detection method, an analysis method, and a prototype implementation of a tool chain, which was applied to two open-source products as a preliminary empirical evaluation.

I. INTRODUCTION

Dynamically typed programming languages and functional languages (or functional features in traditional languages) have been widely adopted in software development projects. However, these technologies have difficulties in code-clone detection in software products developed in such ways.

In source code written in a dynamically typed programming language, types (of variables, parameters, return values, and so on) are not explicitly declared in code. As a result, such type information are not available in static analysis methods. Some dynamically typed languages (such as Python or Ruby) are object-oriented languages which include polymorphism and dynamic dispatching with types of objects; thus, call relations among procedures (methods) are not available in static analysis methods, either. Therefore, it can be difficult for such static clone detectors to detect a code clone of code fragments across boundaries of methods.

In source code written in a functional language, developers can develop their own “control statements” for their own demands for loops, branches, or parallel executions (e.g., Stream class of Java 8). (Remember that such control statements are special ones in traditional procedural programming languages and are defined only by language designers, not by application developers.) This means that using only predefined (by language) control statements is no longer sufficient to represent code patterns in code-clone detection methods, e.g., if (C) A; else B; can be equivalent to myIf(C, lambdaA, lambdaB);

To tackle the above emerging difficulties, this paper presents an arbitrary-granularity model of code fragments and a code-clone detection method and metrics for clone triage as applications of the model.

The contributions of this paper are:

- A clone-detection method designed for dynamically-typed programming languages and/or functional programming languages.
- A kind of type-3[17] clone detection. Each code fragment of a clone class can be across boundaries of procedures or modules (based on an arbitrary-granularity model).
- A prototype implementation and preliminary experiment to explore performance, scalability, and implementation issues of the proposed clone detection method.
- An analysis method of differences (gaps) of code fragments of a clone class, towards a kind of clone triage.

II. EXECUTION-SEMANTIC CLONE DETECTION

An execution-semantic code-clone detection has been introduced in [8], where two code fragments are defined to be equivalent to each other when their execution sequences include the same method invocations in the same order, in any possible executions of the target program. Such execution sequences were generated with a kind of static analysis, by constructing an “entire” call tree from branches and method calls in the target program. Dynamic dispatches of many Object-Oriented programming languages were resolved with callee object’s type information and argument objects’. In this study[8], the detection algorithm generates n-grams of an execution sequence and reports the same n-grams as clone classes. Code fragments of a clone class do not include gaps in terms of execution sequence, but include gaps on source code, because a branch or a procedure call in source code results in a kind of skip in some lines of code or a jump to another procedure. In this sense, such code clones are regarded as a kind of type-3 clone.

In this paper, the proposed clone detection method captures a clone class of code fragments including te gaps also in an execution sequence, with a frequent item-set mining algorithm; thus, the detected code clones are another kind of type-3 clones, near-miss clones in terms of both static (as source code) and dynamic (as an execution sequence) senses. A frequent item-set mining was also used in [20], where the algorithm is applied to an AST of source code, while this study applies the algorithm to a call tree generated from an execution sequence.
import os

def print_extensions_w_for_stmt(file_names):
    for fn in file_names:
        print os.path.splitext(fn)[1]

def get_extensions(file_names):
    return map(lambda fn: os.path.splitext(fn)[1], file_names)

def print_extensions_w_map_func(file_names):
    print ''.join(get_extensions(file_names))

def main():
    file_names = os.listdir(os.curdir)
    print_extensions_w_for_stmt(file_names)
    print_extensions_w_map_func(file_names)
    if '__main__' == '__main__': main()

Figure 1. A sample code with a loop with a for statement and a loop with a map function

Figure 2 shows a row (i.e. without a manual editing) output of a prototype implementation of the proposed clone-detection method, which was applied to an execution trace of the above source code. Here, a box represents a clone class having two code fragments of the above mentioned functions. These code fragments shared two items (contents), extracting an extension of a file name and printing it, as shown by the thick-framed ellipses. Note that a function pygoat.hook/Out/write represents the wrapping function of writing to a file, which was invoked internally by print statements of the above code. Also, posixpath//splitext was a body of os.path.splitext function.

An algorithm of the proposed clone detection in this paper essentially finds a kind of similar sub-graph in a DAG (directed acyclic graph) generated from a call graph. There are several fine-grained clone-detection methods[14][15][5][6] to find isomorphic sub-graphs in a PDG (program dependence graph) within a procedure, or a data-flow graph (a TR[3]). Also, a DAG from an AST was used in a code-clonedetection method[18]. As a kind of semantic approach, state of memory (prerepresentationing a computation, a kind of abstract interpretation) was used in [13].

A. An Example of a Code Clone

Figure 1 shows an example code, written in Python, including two kinds of “control statements”. Two functions print_extensions_w_for_stmt (at line 3) and print_extensions_w_map_func (at line 10) have the same functionality. The former function extracts extensions of given file names by a loop with a for statement (at line 4), while the latter does the same thing by a map function (at line 8) via sub-function get_extensions.

Figure 2 shows an example code, written in Python, including two kinds of “control statements”. Two functions print_extensions_w_for_stmt (at line 3) and print_extensions_w_map_func (at line 10) have the same functionality. The former function extracts extensions of given file names by a loop with a for statement (at line 4), while the latter does the same thing by a map function (at line 8) via sub-function get_extensions.
application to code written in functional programming languages or with such language features. Consider that a developer writes a control-statement like procedure myIf. Two call statements myIf(c, lambdaA, lambdaB) and myIf(d, lambdaE, lambdaF) will become the two nodes in a call tree, that have the same label (myIf) but completely distinct sets of child nodes. Contexts work as longer (more detailed) labels of these nodes to distinguish these calls.

B. Visualization for Clone Analysis

The prototype includes a kind of visualization of clone class. (Notice that the graphs in Figure 2 in Sec. II-A and Figure 6 shown later in this section were automatically generated with the prototype.) These graphs are essentially DAGs, which were generated by a dagification[18] of a call tree.

Nodes to be drawn in the visualization of a clone class are essentially determined as follows:

- Code fragments. For a given clone class c, nodes of code fragments, i.e. target nodes of string balloons.
- Contents. Nodes on the paths from each code fragment f to the common items of string balloons (nodes having any of the labels in a balloon of f).
- Contexts. Nodes on the paths from the root node of a call tree to each code fragment f.

However, such graphs often become large ones. Aiming for effective clone analysis, the prototype includes a kind of dominator analysis of contexts and contents. A dominator is defined as follows: for a given node n in a DAG of a call tree (thus the graph has a single root node), when a node d appears in all paths from the root to n, d is a “dominator” of n. Also, for a given set of nodes ni, when a node d appears in all paths from the root to each of ni, d is the dominator of a set of nodes ni. In the visualization, when there a dominator node of code fragments of a given clone class, nodes (and edges) between each of such a dominator node to the root node are
Fig. 6. A clone set detected from an open-source product removed from the graph. Likewise, if a content node $c$ has a dominator $d$ ($\neq c$) in content nodes, such node $c$ is removed from the graph (see Figure 5).

A content-and-context of a clone class may be applicable to a clone triage (or prioritizing) in a code-clone management. Size of a set of contents shows the complexity of such a clone removal and/or indicates various distinct ways to merge such code fragments of the clone class. Likewise, size of a context shows the area of influences of such a clone removal. E.g., when a context includes only one node, all code fragments of the clone class are called from the procedure of the node, so that it is relatively easy to check whether such a clone removal affects behaviors of the program or not.

Figure 6 shows a clone set with a somehow large set of contents, which was detected in a preliminary experiment (described later in the next section), a visualization framework for complex networks. The clone set (a) contains four code fragments (ellipses in the box). They had the same contents (b), codecs/* and posixpath//splitext, and had the distinct contents (c). If this were an analysis on a single level of a call tree (or on the surface of source code), these distinct contents would be regarded as differences (or gaps) among code fragments of a clone set. However, we can now see that these distinct contents (c) shared the same set of (sub)-contents (d), thus the nodes of (c) were regarded as another clone set. This broader view helps developers consider a plan (or an arrangement) for code-clone removal. In this case, if the distinct contents (c) were merged first, there would not be gaps of the clone set (a) anymore.

IV. A Preliminary Experiment

A. Prototype Tool

A prototype tool chain has been implemented. Figure 7 shows the architecture of the prototype. In the current implementation, a target program is a Python program. In addition to the detection method introduced in Sec. III-A, the following heuristics / optimizations were introduced.

- A parameter of the maximal depth of contents from a target node (default 5) and a parameter of the maximum number of content items of a candidate string balloon (“Max. # of contents of a candidate” for short; default 25). These parameters prevent from generation of item sets too large for a frequent item-set mining routine, because time/space complexities of a frequent item-set mining algorithm is highly sensitive.
TABLE I. EXPERIMENTAL SETUPS

<table>
<thead>
<tr>
<th>Target Product</th>
<th>Collection of Execution Sequence</th>
<th># function calls in execution sequence</th>
<th># unique labels of node in call tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>markdown2</td>
<td>Running 144 unit tests enclosed with its distribution.</td>
<td>277K</td>
<td>1128</td>
</tr>
<tr>
<td>wxPython</td>
<td>Invoking a sample program of GUI drawing application “pySketch”, then drawing a picture with it, saving and loading the picture.</td>
<td>483K</td>
<td>1058</td>
</tr>
</tbody>
</table>

TABLE II. STATISTICS OF CLONE DETECTION

<table>
<thead>
<tr>
<th>Product</th>
<th>Metric</th>
<th>Max. # of contents of a candidate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>markdown2</td>
<td># clone classes</td>
<td>58</td>
</tr>
<tr>
<td></td>
<td>Max. # nodes in a content</td>
<td>6</td>
</tr>
<tr>
<td>wxPython</td>
<td># clone classes</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Max. # nodes in a content</td>
<td>9</td>
</tr>
</tbody>
</table>

Fig. 8. Runtime performance of Clone Detection

(a) markdown2

(b) wxPython

The implementation reuses a scalable implementation\cite{4} of the Apriori algorithm; however, as a matter of course, this part is still the largest memory-intensive step of the prototype. The above heuristics and optimizations are vital for applying the algorithm to the actual size of a software product.

B. Application to Open-Source Products

To explore performance/scalability issues, the prototype implementation was applied to two open-source products written in Python, namely “markdown2”\cite{2} and “wxPython”\cite{3}. Markdown2 is a text-processing program, which converts a Markdown text into HTML format. The wxPython is a GUI framework, which wraps a C++ framework (“wxWidgets”\cite{4}).

The prototype was run on a PC with CPU Intel Xeon E5520 2.27GHz and a 32GiB RAM. Table I shows the experimental setups for these target products. In either case, the execution sequence invoked 1000+ distinct functions or methods, more than two million times in total. Table II shows the results of clone detection. The “Max. # of contents of a candidate” is a parameter of the detection method, described earlier in Sec. IV-A. By giving the larger value of this parameter, the closer nodes to the root node of a call tree (i.e. an entry point of the target program) will be regarded as candidates of clones.

Figure 8 shows elapsed times and peak memory usages of step 2 of the clone detection. Note that they have a logarithmic axis! These graphs shows that time and space complexity of the detection method are exponential to the “Max. # of contents of a candidate” parameter, as expected.

the large value of “Max. # of contents of a candidate” had a kind of sub-function in this experiment. Nodes near the root will always share some functionalities with the other nodes, so that they do not have to be regarded as candidates of code clones. For example, in the case of “markdown2”, giving the large value of “Max. # of contents of a candidate” had a kind of sub-function in this experiment. Nodes near the root will always share some functionalities with the other nodes, so that they do not have to be regarded as candidates of code clones. For example, in the case of “markdown2”, giving the

\cite{apriori 6.16 http://www.borgelt.net/apriori.html}

\cite{markdown2 2.2.2 https://github.com/trentm/python-markdown2 .}

\cite{wxPython 2.8.12.1 http://www.wxpython.org/ .}

\cite{wxWidgets http://wxwidgets.org/ .}
largest value (30) as this parameter, a clone class having 29 shared items in its contents was detected. Figure 9 shows a graph of this clone class, where some labels were dropped manually for readability and the limitation of space. Notice that a node at the left bottom of this graph has depth 5 (on one of the shortest paths from a code fragment node) or 12 (on the longest path). This node corresponds to the function of a regular expression library and such features seem too general and fine grained to be investigated, considering the top level nodes of this graph. As a result of including such fine grained and detailed information, this graph becomes rather difficult to understand.

V. Threats to Validity

A. Issues regarding Execution Traces

The proposed clone-detection method requires an execution trace of a target program. This method is not able to detect code clones in the code fragments that are not executed in a execution trace. In cases of a software development process with unit testing, we may possibly obtain a suit of unit tests with high coverage to a target program. This issue is much more difficult when trying to detect clone classes having the code fragments across (beyond) the boundary of procedures. To find such clones, the input execution sequence has to include such procedure calls, and thus, needs to have a high coverage in terms of branch coverage or condition coverage, not merely in statement coverage[16]. Some automatic test generation methods[2] might be a solution against this issue.

Another (possibly minor) issue of this topic is the configuration[10] of a software product. If a source code has a configuration for a portability of distinct environments (such as OSs or VMs), there is no single execution sequence which will cover entire source code. In the worse case, there will be no compatible execution sequences among such environments (because of differences in the opcodes of VM) to be analyzed.

B. Need of Refinements

The proposed detection method and its prototype are still at an early stage of study; thus, there are numbers of issues for a practical applications such as:

- Code block. A code fragment of a detected clone class is the entire body of a procedure. Because the proposed method is missing a representation of a code block or a sub-sequence of statements; this may be solved by introducing a fine-grained labeling of nodes in a call tree.

- Coverage of contents in code fragments. The current clone-detection method does not take into account a ratio (percentage) of contents in each code fragment in a clone class. As a result, when a small part of a procedure shares a code with another procedure, these procedures will be detected as a clone class. As a matter of course, this coverage issue is closely related to (depends on) the previous, “Code block” issue. The only available parameter is “minimum size of a set of contents”.

Such a content coverage may resolve a “sub-function” of a large value of parameter “Max. # of contents of a candidate”, as mentioned in Sec. IV-B. In Figure 9, contents of clone classes included the number of shared items among its code fragments; however, the contents also included a number of unshared items,
thus the content coverage of this clone class will be small.

- Recursion. The current definition of context does not take into account recursive calls among procedures. As a result, two contexts $f \rightarrow f$ and $\rightarrow f \rightarrow f$ were regarded as distinct contexts. A dominator analysis should be revised to account for such recursive calls.

- Unused information of an execution trace. The current definition and implementation merely uses the labels of procedures. However, an execution trace includes other kinds of information, e.g., an order of procedure calls, types, or values of variables, which were used in another study of code searches[9]. This information may have benefits in the definition of an equivalence of code fragments or as a more fine-grained label (or context) of nodes in a call tree, towards a more precise clone-detection method.

C. Performance/Scalability Issue

As described earlier in Sec. IV-B, the frequent item-set mining algorithm is highly sensitive to sizes of item sets in an input. Parameters “maximal depth of contents” and “Max. # of contents of a candidate”, I believe, partially solves (or relaxes) this issue. However, the preliminary experiment includes just two cases. This issue still needs further evaluations, by applying the proposed method to larger products or the products in different domains. Depending on the results of such experiments, other solutions will be required, such as further filtering of nodes, an automated parameter tuning embedded in a clone-detection method, or a new algorithm of frequent item-set mining which is tuned for this application domain of clone detection.

D. Dependence on a Programming Language

The current implementation is designed to a source code written in a Python programming language, which has a dynamically-typed nature and functional language entities. In addition, it has useful debugging/profiling APIs to extract an informative and fine-grained execution trace (of a call stack). However, there exist the programming languages that do not provide such APIs or do not use a call stack in an execution of a program because of a lazy evaluation and optimizations based on it, such as Haskell[5].

VI. CONCLUSION AND FUTURE WORK

This paper presents a presentation of a code fragment (named a “string balloon”) and a clone-detection method using it towards establishing an efficient type-3 code-cloning detection in dynamically typed programming languages and functional programming languages.

The performance of the detection method is still “peeky”, i.e. highly sensitive to the input and parameters, as expected and shown in the preliminary experiment. Apparently, further refinements are needed in its definition, algorithm and implementation, as preparations toward empirical evaluation of the proposed method.

Also, the detection method is based on an analysis of nodes of a call tree. This implies that the method seems relatively easily extended to or combined with a code search method[12]; thus, such a unified clone-detection and code-search method is planned.

Acknowledgment This work was supported by JSPS KAKENHI Grant Number 24650013.

REFERENCES
