Agec: An Execution-Semantic Clone Detection Tool

Toshihiro Kamiya
Department of Media Architecture, School of Systems Information Science, Future University Hakodate
116-2 Kamedanakano-cho, Hakodate, Hokkaido, Japan 041-8655. Email: kamiya@fun.ac.jp

***Abstract*—*Agec* is a semantic code-clone detection tool from Java bytecode, which (1) applies a kind of abstract interpretation to bytecode as a static analysis, in order to generate n-grams of possible execution traces, (2) detects the same n-grams from distinct places of the bytecode, and (3) then reports these n-grams as code clones. The strengths of the tool are: static analysis (no need for test cases), detection of clones of deeply nested invocations, and Map-Reduce ready detection algorithms for scalability.**

I. Introduction

Refactoring is an important application of code-clone detection. A number of code-clone detection tools have been designed to detect the code fragments that need to be refactored. Moreover, such clone detection tools should be able to detect a code clone which includes both not-yet-refactored code fragments and the refactored code fragments equivalent to them.

This paper presents a code-clone detection tool named **Agec**, based on an execution model of multi-level invocation, which detects code fragments equivalent in terms of method invocation, but not equivalent in terms of code structure (thus, *semantic* clones).

Figure 1 shows a motivationg example as an example code, which includes both refactored code `(a)` and not-yet-refactored code `(b)`. These codes apparently have distinct code structures; nonetheless, the methods invoked during their executions are the same. Agec reports these code fragments as a code clone.

Contributions of the proposed method/tool are:

1) a method for a kind of abstract interpretation, to track multi-level invocations and generate possible execution sequences
2) a definition of equivalency for such arbitrary granularity execution sequences
3) a working prototype clone-detection tool for empirical evaluation

```
import static java.lang.Integer.parseInt;
import java.util.Calendar;

public class ShowWeekdayR {
  static String[] weekDays = new String[] {
    "Sun", "Mon", "Tues", "Wednes", "Thurs", "Fri",
        "Satur" };
  static void setCalendarDate(Calendar cal,
      String date, String sep) {
    String[] fs = date.split(sep);
    cal.set(parseInt(fs[0]), parseInt(fs[1]) - 1,
        parseInt(fs[2]));
  }
  public static void main(String[] args) {
    Calendar cal = Calendar.getInstance();
    if (args.length >= 1) {
      if (args[0].indexOf("-") >= 0)
        setCalendarDate(cal, args[0], "-");
      else if (args[0].indexOf("/") >= 0)
        setCalendarDate(cal, args[0], "/");
      else if (args[0].indexOf(".") >= 0) {
        String[] fs = args[0].split(".");
        cal.set(parseInt(fs[0]), parseInt(fs[1]) - 1,
            parseInt(fs[2]));
      }
    }
    int wd = cal.get(Calendar.DAY_OF_WEEK);
    System.out.printf("%sday\n", weekDays[wd - 1]);
  }
}
```

Fig. 1. An Example Code Undergoing Refactoring

II. Arbitrary-Granularity Execution Sequence

To determine which two code fragments are equivalent, algorithms need to compare code fragments as a sequence (or some kind of data structure such as a sub-graph) of unit entities of software products. However, which entities can become such "unit" entities is a tough question to answer. The existing clone-detection methods/tools consider this point in their own assumptions (or approximations): some use characters or lines [15][16], some use syntactic tokens[1][3][8][10], some use the nodes of a AST[4][7], some use the nodes of a PDG[9][11][14] or design diagrams [2][6][17], and some use bytecode instructions [13].

The proposed detection method generates possible method invocation sequences with a kind of abstract interpretation in a static way. The detection method compares code fragments as a sequence of method invocations; however, the "unit" entities are not limited to the methods directly invoked, but all methods at all nesting levels of method invocations are regarded as such unit entities.

This multi-level (thus arbitrary-granularity) equivalency enables a semantic clone detection. When the same method invocation sequence is "folded" into two distinct code structures, such as a code fragment in a method and the two code fragments of two methods that are executed sequentially in a program execution, such two structures are the same by definition.

III. Steps of the Detection Method

The proposed clone-detection method consists of the following steps:

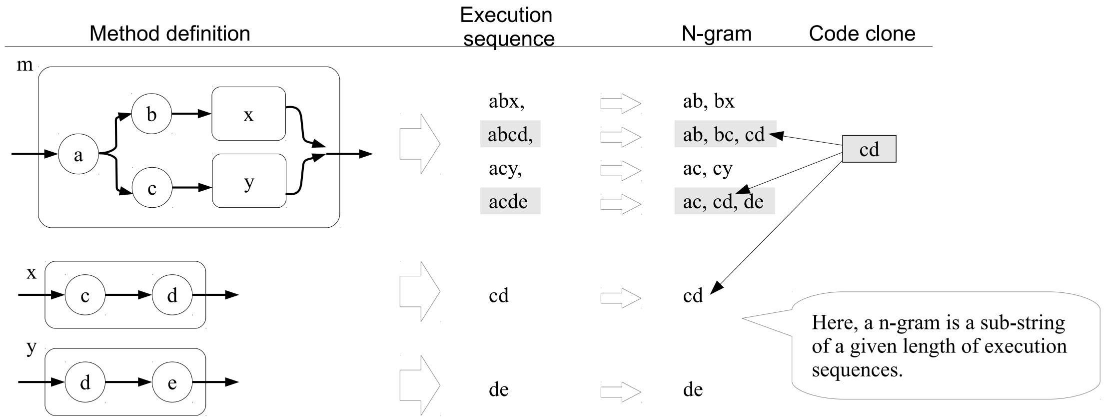

Fig. 2. Illustration of Control Dependencies among Nested Methods

1) Generating arbitrary-granularity execution sequences
2) Extracting n-grams from execution sequences
3) Detecting the same n-grams from distinct locations

Figure 2 shows an example to explain how each step works. First, in step 1, execution traces in each method are generated. At each method invocation, one execution trace is converted into two traces, one includes the method as itself and the other tracks inside of the called method definition. In the Figure, `abx` and `abcd` are one of such two traces, and here the sub-sequence `cd` within the latter is invoked inside of `x`. Next, in step 2, n-grams (2-grams, here) are extracted from all positions of all these traces. Finally, in step 3, the same n-grams that are generated from the distinct positions are detected as code clones. The n-gram `cd` appears at three locations in the execution traces: `abcd` (within method `m`), `acde` (also within method `m`), and `cd` (within method `x`). The second n-gram location includes an invocation of `c` directly from `m`'s definition and an invocation of `d` via method `y`, which is invoked from `m`.

If we did not consider an arbitrary-granularity execution sequence, but collect execution traces from just the surface of each method definition, the execution sequences shown in the gray background would not generated and thus the code clone of n-gram `cd` would not be detected in this case.

IV. Prototype Implementation

A prototype CLI tool has been implemented in about 1500 lines of Python code. Figure 3 shows a screen capture of a terminal where the tool is applied to a Java code in Figure 1. Here, the output (a file "clone-linenums.txt") includes locations of code fragments of each code clone and its method invocation sequence to help understanding functions of such code fragments.

A. Performance

The proposed detection method implies generating a large number of n-grams, and this could be a performance flaw. As an early empirical evaluation, the prototype implementation was applied to an open-source product, namely ArgoUML 0.28.1[1]. The detection result is shown in Table I, where the given n-gram size is 6. The performance data are shown in Table II. The tool was run on a PC with a CPU Intel Xeon E5520 2.27GHz and memory 32GiB. No multi-threading was used in the implementation, so that the elapsed time is almost the same as the CPU time.

TABLE I
Product Size and Code-Clone Detection Result from ArgoUML

Metric	Value
Classes having method definitions	1,700
Method definitions	8,888
Locations where n-grams were generated	1,232,292
Distinct n-grams	282,753
n-grams of code clones	4,634

TABLE II
Performance Data of Code-Clone Detection from ArgoUML

Step	Elapsed time (sec.)	Peak memory use (MiB)
Disassemble of bytecode	1,351	n/a
Step 1 and Step 2	387	564
Step 3	38	500
Format code fragments	3	57

V. Discussions/Related Work

The original motivation of this study was to find out a code clone, which includes code fragments scattered in methods, as a result of a refactoring (code modification) presented in [12].

The proposed approach employs a kind of abstract interpretation of Java bytecode. Another study of an abstract interpretation for code-clone detection is found in [5].

The prototype implementation is hosted in a GitHub page[2]. The tool is not matured, and still needs optimizations such as memorization in n-gram extraction or a Map-Reduce style

[1] http://argouml.tigris.org/
[2] http://github.com/tos-kamiya/agec

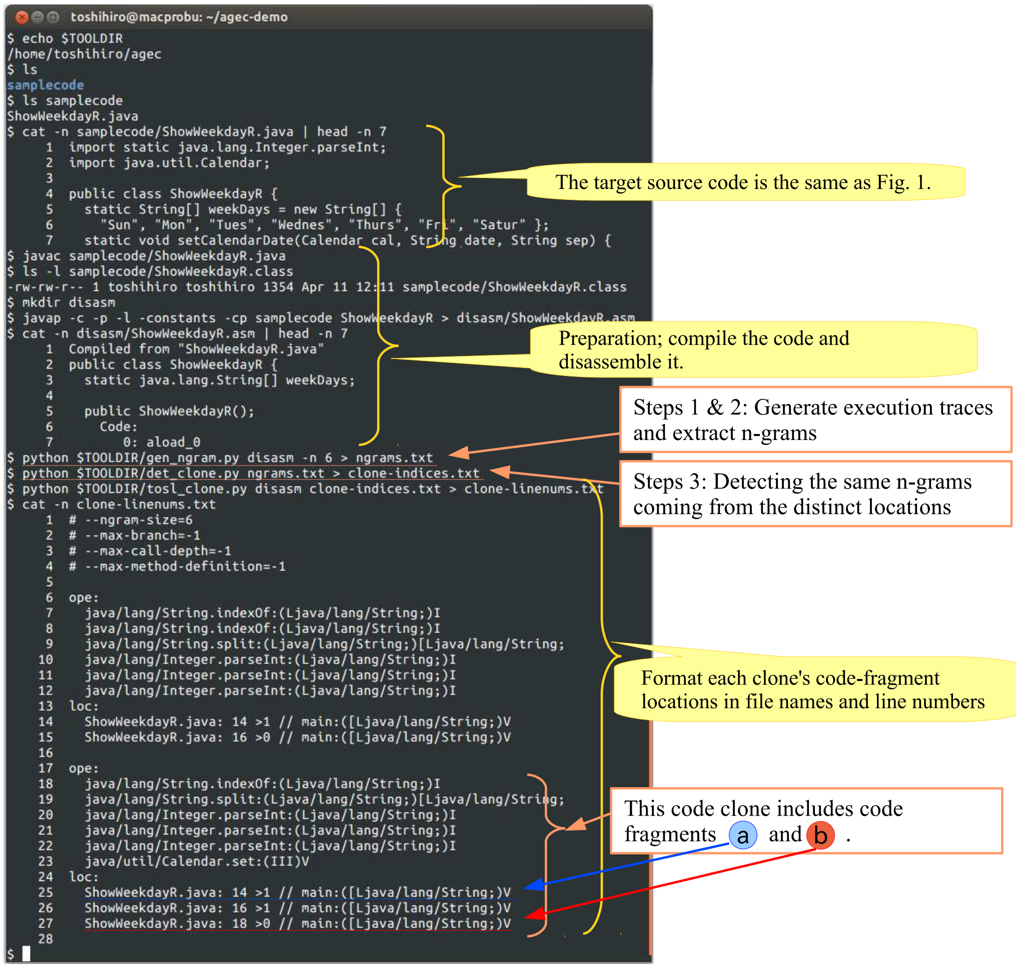

Fig. 3. Screen Capture of Application of Tool to the Example

detection algorithm. The detection algorithm also needs refinement to permit more code modifications by refactoring tasks.

Acknowledgments

This work was supported by JSPS KAKENHI Grant Number 24650013.

References

[1] H. Abdul Basit and S. Jarzabek, “A Data Mining Approach for Detecting Higher-Level Clones in Software,” *IEEE TSE*, vol. 35, no. 4, pp. 497-514, 2009.

[2] M. Alalfi, J. Cordy, T. Dean, M. Stephan, and A. Stevenson, “Near-Miss Model Clone Detection for Simulink Models,” 6th Int’l Workshop on Software Clone (IWSC’12), pp. 78-79, 2012.

[3] B. Baker, “Finding Clones with Dup: Analysis of an Experiment,” *IEEE TSE*, vol. 33, no. 9, pp. 608-621, 2007

[4] I.D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier, “Clone Detection Using Abstract Syntax Trees,” Int’l Conf. Software Maintenance (ICSM’98), 1998.

[5] W. Evans, C. Fraser, Fei Ma, “Clone Detection via Structural Abstraction,” 14th Working Conf. Rev. Eng. (WCRE’07), pp. 150-159, 2007.

[6] B. Hummel, E. Juergens, and D. Steidl, “Index-Based Model Clone Detection,” 5th Int’l Workshop Softw. Clone (IWSC’11), pp. 21-27, 2011.

[7] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, “Deckard: Scalable and Accurate Tree-Based Detection of Code Clones,” IEEE ICSE ’07, pp. 96-105, 2007.

[8] T. Kamiya, S. Kusumoto, and K. Inoue, “CCFinder: A Multi- Linguistic Token-Based Code Clone Detection System for Large Scale Source Code,” *IEEE TSE*, vol. 28, no. 7, pp. 654-670, 2002.

[9] R. Komondoor and S. Horwitz, “Using Slicing to Identify Duplication in Source Code,” 8th Int’l Sympo. Static Analysis (SAS’01), p. 40-56, 2001.

[10] L. Prechelt, G. Malphol, and M. Philippsen, “Finding Plagiarisms among a Set of Programs with JPlag,” *Journal of Universal Computer Sci.*, vol. 8, no. 11, pp. 1016-1038, 2002.

[11] Y. Higo, Y. Ueda, M. Nishino, and S. Kusumoto, “Incremental Code Clone Detection: a Pdg-Based Approach,” 18th Working Conf. Rev. Eng. (WCRE’11) pp.3-12, 2011.

[12] T. Kamiya, “Conte∗t clones or re-thinking clone on a call graph,” 6th Int’l Workshop Softw. Clone (IWSC’12), pp. 74-75, 2012.

[13] I. Keivanloo, Chanchal K. Roy, J. Rilling, “Java Bytecode Clone Detection via Relaxation on Code Fingerprint and Semantic Web reasoning,” 6th Int’l Workshop Softw. Clone (IWSC’12), pp. 36-42, 2012.

[14] J. Krinke, “Identifying Similar Code with Program Dependence Graphs,” 8th Working Conf. Rev. Eng. (WCRE’01), pp. 301-309, 2001.

[15] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, “ CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code,” *IEEE TSE*, vol. 32, no. 3, pp. 289-302, 2004.

[16] C.K. Roy and J.R. Cordy, “NiCad: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization,” IEEE ICPC’08, pp. 172-181, 2008.

[17] U. Tekin, U. Erdemir, and F. Buzluca, “Mining Object-Oriented Design Models for Detecting Identical Design Structures,” 6th Int’l Workshop Softw. Clone (IWSC’12), pp. 43-49, 2012.