SeByte: A Semantic Clone Detection Tool for Intermediate Languages

Iman Keivanloo[±], Chanchal K. Roy[*], Juergen Rilling[±]

[±]Department of Computer Science
Concordia University
Montreal, Canada
{i_keiv, rilling@cse.concordia.ca}

[*]Department of Computer Science
University of Saskatchewan
Saskatoon, Canada
croy@cs.usask.ca

***Abstract*—SeByte is a semantic clone detection tool which accepts Java bytecode (binary) as input. SeByte provides a complementary approach to traditional pattern-based source code level clone detection. It is capable of detecting clones missed by existing clone detection tools since it exploits both pattern and content similarity at binary level.**

***Index Terms*— clone detection, Semantic Web, Java bytecode.**

I. Introduction

In this paper, we introduce SeByte, a clone detection tool that is capable of detecting semantic clone classes such as the one in Fig. 1. In our earlier research [1] we introduced the core of SeByte, including (1) relaxation on code fingerprinting, (2) multi-dimensional comparison, and (3) metric-based clone detection using both set theory and pattern matching.

Contrary to type-1, 2, and 3 clones, there is no agreement on the definition of a *semantic clone*. Although, one might consider type-4 clones to be semantic clones, there are cases when semantic clones go beyond the scope of type-4 clones (based on the definition of Roy et al. [2]). Figure 1 illustrates such a case, which has been detected in the EIRC dataset [3] using our SeByte clone detection tool. Block A and C are semantically related since both blocks, based on their code pattern and variable names, manipulate the color of GUI elements. Therefore, from a maintenance perspective, if one of the blocks requires a bug fix or an update also similar cloned blocks should be inspected as part of the maintenance task.

In this example, method block A and B constitute an easy detectable type-3 clone-pair (group). However, detecting that blocks A and C are cloned is not as straight forward. Clone detection tools such as CCFinder [4] or NiCad [5] when configured with less restrictive settings (e.g., NiCad with blind detection enabled) can detect these two blocks as type-2 clones, since in this example only method names were re-named.

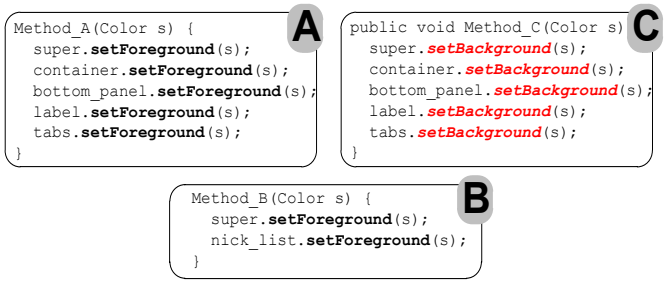

Fig. 1. Three methods with similar pattern. Are they type-3 or semantic clones? From maintenance point of view the best answer is semantic clone.

However, using these less restrictive settings would also create a high false positive rate, significantly affecting the usability of the results. Rather than reducing the precision of existing tools, a more specialized detection approach for identifying functionality similarities between code blocks is required. In fact, such a semantic clone detection approach (i.e., SeByte) should classify in our example all three blocks in one class (i.e., group) as semantic clones with high confidence.

II. SeByte Architecture Overview

SeByte is implemented in Java and currently only supports clone detection on Java bytecode. The tool is available online for download at http://aseg.cs.concordia.ca/sebyte. SeByte was one of the (1) few binary-level detection tools and (2) the first binary level tool using an inference engine-based approach. To improve the ease of use, we emphasized an open and flexible tool design to help both end-users and the research community to apply the tool in various application contexts. Thus, within the download package, there are four executable standalone Java files which are tagged based on their execution order. Figure 2 provides a schematic overview for each process, including its (1) main task, (2) inputs, and (3) outputs. Each process must be executed using the command line such as "`java -jar step1_prepare.jar`". Note that after completing the first process, the subject system's bytecode files (i.e., .class) must be copied to the corresponding folder. Figure 3 and 4 show a sample output of *step2_extract* process. The output generated in the steps is plain text. In Fig. 3, we use a directed labeled graph notation to illustrate the content of the populated ontology. A generated query (Fig. 4) then takes advantage of the embedded Semantic Web inference engine to detect all similar patterns to the one shown in Fig. 3, even for cases with gap [1].

III. SeByte Configuration

SeByte uses five major, stable configuration parameters which we describe below. The non-stable input parameters were omitted due to space limitations. The parameters are read from the `config` file (Fig. 2), with the first parameter (last one in the config file) being the root address, which is used by SeByte as destination for its (1) temporary directories and files[1], (2) ontologies, (3) SPARQL queries, and (4) all the CSV and textual reports to the *root address* parameter.

[1] Since the idea behind SeByte development has been to support full openness, it reports all of its internal data step by step in plain-text format.

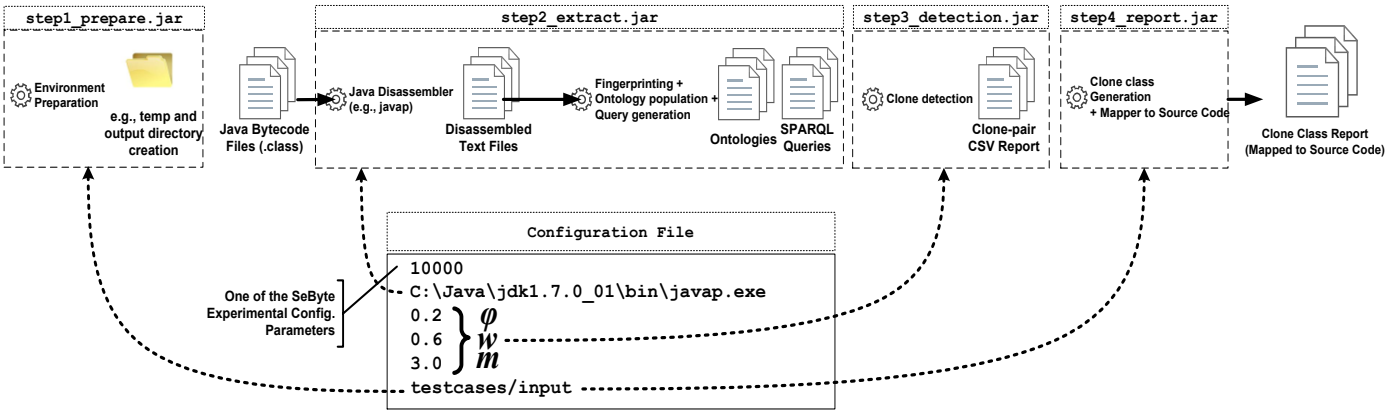

Fig. 2. SeByte processes and the configuration file description - The execution order (left to right) and their dependency on the configuration parameters

The address to the *Java disassembler* (e.g., javap.exe) is the second major parameter. The last three parameters are type, method, and minimum size values denoted by φ, ω, and m respectively. The acceptable ranges for these parameters are: $0 \leq \varphi \leq 1$, $0 \leq \omega \leq 1$, and $0 \leq m \leq \infty$. φ and ω are the major SeByte thresholds [1] for its *Jaccard function*. m specifies the minimum size of clone pairs in terms of fingerprinted tokens in the Java intermediate language level.

A. SeByte Parameter Calibration

One of the challenges when using SeByte is the need to select appropriate values for the thresholds. As noted earlier [1], we do believe SeByte should be configured based on its application context. In other words, there is no single configuration for its two thresholds (i.e., φ and ω). Nevertheless, we have observed [1] based on our manually created dataset that some combination (e.g., $\{\omega = 5.3, \varphi = 1.9\}$) seems to outperform other configurations. The default value for m is 3.

B. SeByte Extensibility

We have designed SeByte to support openness, ease of use and extensibility. We identified four major tasks and implemented them as separated and independent executable processes (Fig. 2). All implemented steps can be replaced by further customized instances. For example, the current implementation of step 2 is based on our specialized fingerprinting rules [1], which can easily be replaced by user customized implementations. Note that, step 2 is responsible for fact extraction which disassembles the bytecode content and creates the ontologies and SPARQL queries.

IV. Interesting Examples

SeByte has been designed to detect semantic clone-pairs, which are not detectable by the source code level clone detection tools [1]. Figure 5 shows one of the interesting examples of such a clone. Pair #1 and #2, which constitute the clone-pair, are textually highly dissimilar. However, SeByte can detect them as a clone-pair, due to the use of (1) binary content and (2) relaxation on fingerprinting in its detection approach. Figure 5 illustrate the advantage of using binary content that provides like in this case *method inlining*. The method inlining in pair #1 (indicated through the arrow) exemplifies how SeByte can recognize these two fragments as a semantic clone-pair with high confidence.

V. Demonstration

As part of the tool demonstration we execute step by step the four major processes of SeByte and describe in detail (1) the necessity of each process step, (2) its external, and (3) internal serialized data. Describing the internal data will be of interest to the audience (1) to provide additional insights on how SeByte takes advantage of Semantic Web openness and reasoning and (2) how this internal data can be used for further processing and extensions. We will also demonstrate the SeByte Semantic Web-based search process using a graphical querying and inference engine to highlight the intuitiveness of the SeByte pattern search process. Finally, the SeByte clone reports in CSV, intermediate language, and source code-level will be discussed and examples illustrating the applicability of SeByte in detecting semantic clones will be shown.

VI. Conclusion

In this paper, we introduce SeByte which is the first implementation of our semantic clone detection approach [1]. SeByte is able to detect semantic clone by (1) exploiting valuable facts available at bytecode level and (2) taking advantage of our heterogeneous clone detection algorithm [1]. We have designed and implemented SeByte to make it easy to understand and extend by supporting openness as much as possible.

References

[1] I. Keivanloo, C. K. Roy and J. Rilling, "Java Bytecode Clone Detection via Relaxation on Code Fingerprint and Semantic Web Reasoning," IWSC, 2012, 7 pp.

[2] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach", Science of Com. Prog., vol. 74, no. 7, May 2009, pp. 470-495.

[3] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools", Tran. Soft. Eng. vol. 33, no. 9, 2007, pp. 577–591.

[4] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," Tran. Soft. Eng., vol. 28, no. 7, 2002, pp. 654–670.

[5] C. K. Roy and J. R. Cordy, "NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization", Proc. ICPC, 2008, pp. 172-181.

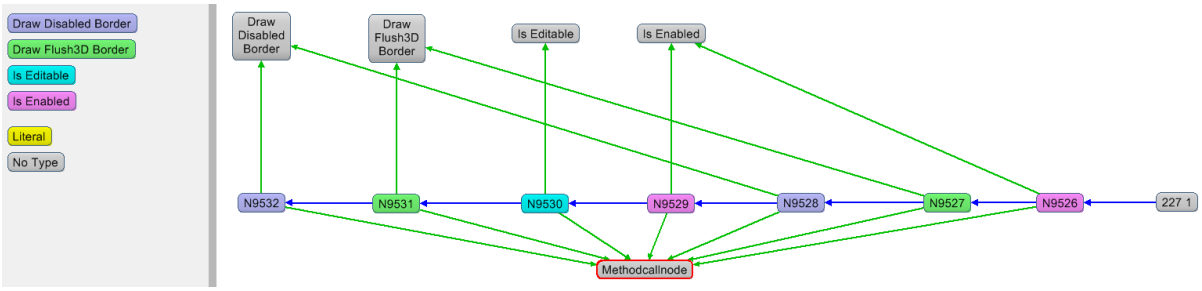

Fig. 3. Part of the populated ontology for method fingerprints which is shown using directed labeled graph notation.

```
select distinct ?link where { ?n0 ?link  ?n1 .
   ?n1 rdf:type method:isEnabled .
   ?n1 ?link  ?n2 .
   ?n2 rdf:type method:drawFlush3DBorder .
   ?n2 ?link  ?n3 .
   ?n3 rdf:type method:drawDisabledBorder .
   ?n3 ?link  ?n4 .
   ?n4 rdf:type method:isEnabled .
   ?n4 ?link  ?n5 .
   ?n5 rdf:type method:isEditable .
   ?n5 ?link  ?n6 .
   ?n6 rdf:type method:drawFlush3DBorder .
   ?n6 ?link  ?n7 .
   ?n7 rdf:type method:drawDisabledBorder .}
```

Formulates the expected method tokens to be matched including their order

Fig. 4. Sample SPARQL query which detects all similar code fragments with and without transitivity (i.e., gap) similar to the modeled ontology in Fig. 3

Pair #1

```
public static String attrEncode(String src) {
StringBuffer buf = new StringBuffer();
char [] ref = src.toCharArray();

for (int i = 0; i < ref.length; i++) {
    buf.append(attrEncode(ref[i]));
}

return buf.toString();
}
```

```
public static String attrEncode(char c) {
char subst;

switch (c) {
case BOLD:
    subst = 'B';
    break;
case COLOR:
    subst = 'C';
    break;
case BELL:
    subst = 'A';
    break;
case RESET:
    subst = 'O';
    break;
case ITALIC:
    subst = 'I';
    break;
case REVERSE:
    subst = 'R';
    break;
case UNDERLINE:
    subst = 'U';
    break;
default:
```

Pair #2

```
public static String attrDecode(String src) {
StringBuffer buf = new StringBuffer();
char [] ref = src.toCharArray();

for (int i = 0; i < ref.length; i++) {
    // If it's not an escape character or it's the last character,
    if ('%' != ref[i] || i == ref.length - 1) {
    buf.append(ref[i]);
    continue;
    }

    char subst;

    i++; // Advance to the next character and try to decode it.

    switch (ref[i]) {
    case 'B':
    subst = BOLD;
    break;
    case 'C':
    subst = COLOR;
    break;
    case 'A':
    subst = BELL;
    break;
    case 'O':
    subst = RESET;
    break;
    case 'I':
    subst = ITALIC;
    break;
    case 'R':
    subst = REVERSE;
    break;
    case 'U':
```

Fig. 5. An interesting semantic clone class detected by SeByte which takes advantage of method inlining within binary content