JSIMIL
A Java Bytecode Clone Detector

Luis Quesada, Fernando Berzal and Juan Carlos Cubero
Department of Computer Science and Artificial Intelligence, CITIC, University of Granada, Granada 18071, Spain

Keywords: Java bytecode, Clone detection, Metrics, Hierarchical matching.

Abstract: We present JSimil, a code clone detector that uses a novel algorithm to detect similarities in sets of Java programs at the bytecode level. The proposed technique emphasizes scalability and efficiency. It also supports customization through profiles that allow the user to specify matching rules, system behavior, pruning thresholds, and output details. Experimental results reveal that JSimil outperforms existing systems. It is even able to spot similarities when complex code obfuscation techniques have been applied.

1 INTRODUCTION

Code clone detection can expose many interesting features that are deeply embedded in the program source code: common clusters of sentences, perfect or almost perfect code matches, limits of the programming language and derived workarounds, coding style or lack thereof, and programming antipatterns.

Furthermore, tracking down duplicated code clusters has many commercial applications, as it can be used to prove plagiarism in legal disputes over intellectual property rights or software patents (Belkhouche et al., 2004), help in automatic refactorization or code maintenance (Tairas, 2008), and compare student assignments (Cosma and Joy, 2006).

Most traditional techniques include string matching, whose variants are implemented by YAP3 (Wise, 1996) and Baker and Mamber's (Baker and Manber, 1998); string hash matching, as in Moss (Schleimer et al., 2003); token-based matching, as implemented by JPlag (Prechelt et al., 2000), CP-Miner (Li et al., 2006), and CCFinder (Kamiya et al., 2002); or control-flow graph analysis and its variants, as the enhanced CFGs used by JDiff (Apiwattanapong et al., 2007).

More recent techniques compare program dependence graphs (Krinke, 2001), as in GPLAG (Liu et al., 2006); analyze simplifications of program behavior, either in the way of program slices (Weiser, 1981) or by considering them as black boxes, as implemented by Semantic Diff (Jackson and Ladd, 1994); or calculate and compare metrics (Dunsmore, 1984).

These code clone detectors are hard-coded and they are not customizable at all, hence the results they produce can be inaccurate beyond the specific situations they are designed for. In contrast, JSimil is customizable to fit any need.

2 JSIMIL

JSimil is a code clone detector that performs a heuristic matching of program hierarchies at the bytecode level.

JSimil results, which are comprised of nested elements matches, are generated by matching the classes, the methods, and the basic blocks within Java software. Several metrics, such as the number of different kinds of instruction, are calculated for every basic block.

JSimil input consists of a configuration file, which determines the paths to both compiler and disassembler and options about their usage; a profile, which determines the system behavior, thresholds, pruning rules, and output detail; and data input, which contains the source code (.java) files, bytecode (.jar or .class) files, or a mix of them, of the programs to compare. The output is a browsable hierarchy of matches across programs and their elements.

2.1 Hierarchical Matching

Matching is done at the block level, the method level, the class level, and the program level. Cus-

Quesada L., Berzal F. and Carlos Cubero J. (2010).
JSIMIL - A Java Bytecode Clone Detector.
In *Proceedings of the 5th International Conference on Software and Data Technologies*, pages 333-336
DOI: 10.5220/0003013403330336

tomized profiles determine how the system performs the matching.

2.1.1 Block Matching

The normalized similarity S_b for any two blocks (b_1 and b_2) is calculated as follows:

$$S_b(b_1,b_2) = 1 - \frac{\sum_{i=0}^{M} |b_1(i) - b_2(i)| * w(i)}{\sum_{i=0}^{M} \max\{b_1(i), b_2(i)\} * w(i)} \quad (1)$$

where M is the number of metrics computed for the blocks, $b_x(i)$ is the value of the ith measure for the x block, and $w(i)$ is the weight for the ith measure as defined in the profile.

It should be noted that $S_b(b_1,b_2) = S_b(b_2,b_1)$.

2.1.2 Method Matching

The normalized similarity S_m for any two methods is calculated by choosing one of them (m_1) according to profile parameters and method size, ordering its blocks by decreasing size, and matching them with the ones in the other method (m_2) by applying the following expression:

$$S_m(m_1,m_2) = \sum_{i=0}^{B_{m_1}} S_b(m_1(b_i), bm_b(m_2, m_1(b_i))) * s(m_1(b_i)) \quad (2)$$

where B_{m_x} is the number of blocks contained in m_x; $m_x(b_i)$ is the ith block in the method m_x, ordered by decreasing size; $bm_b(m_x, m_y(b_i))$ is the block contained in m_x that best matches the block b_i in m_y; and $s(m_x(b_i))$ is the number of instructions of the block b_i in m_x.

The profile parameters also determine which method matchings are tried and which blocks are effectively matched, according to similarity thresholds, elements sizes, and already matched elements, among other factors.

In general, $S_m(m_x,m_y) \neq S_m(m_y,m_x)$ because the best match for $m_x(b_i)$ might be $m_y(b_j)$ but the best match for $m_y(b_j)$ might not be $m_x(b_i)$.

2.1.3 Class Matching

The normalized similarity S_c for any two classes is calculated like the S_m method similarity, just by replacing blocks (b) with methods (m) and methods (m) with classes (c). Formally:

$$S_c(c_1,c_2) = \sum_{i=0}^{M_{c_1}} S_m(c_1(m_i), bm_m(c_2, c_1(m_i))) * s(c_1(m_i)) \quad (3)$$

Two classes will be matched only if their similarity is higher than an user-defined profile threshold. When any class is not matched, its methods ascend through the hierarchy and the program temporary becomes their parent, so they can be matched in the program matching step.

It should be noted that $S_c(c_1,c_2) \neq S_c(c_2,c_1)$, by the same reason that $S_m(m_1,m_2) \neq S_m(m_2,m_1)$.

2.1.4 Program Matching

The normalized similarity S_p for any two programs is calculated by choosing one of them (p_1) according to its size and the profile parameters, ordering its classes and still-unmatched methods by decreasing size, and matching them with the ones in the other program (p_2) by applying the following expression:

$$S_p(p_1,p_2) = \sum_{i=0}^{C_{p_1}} S_c(p_1(c_i), bm_c(p_2, p_1(c_i))) * s(p_1(c_i)) + \sum_{i=0}^{U_{p_1}} S_m(p_1(u_i), bm_u(p_2, p_1(u_i))) * s(p_1(u_i)) \quad (4)$$

where C_{p_x} is the number of classes contained in p_x; $p_x(c_i)$ is the ith class in the program p_x, ordered by decreasing size; $bm_c(p_x, p_y(c_y))$ is the class contained in p_x that best matches the class c_i of p_y; $s(p_x(c_i))$ is the number of instructions in class c_i in p_x; U_{p_x} is the number of still unmatched methods contained in p_x; $p_x(u_i)$ is the ith unmatched method in the program p_x, ordered by decreasing size; $bm_u(p_x, p_y(u_i))$ is the unmatched method contained in p_x that best matches the method u_i in p_y; and $s(p_x(u_i))$ is the number of instructions of the still-unmatched method u_i in p_x.

It should be noted that $S_p(p_1,p_2) \neq S_p(p_2,p_1)$, by the same reason that $S_m(m_1,m_2) \neq S_m(m_2,m_1)$ and $S_c(c_1,c_2) \neq S_c(c_2,c_1)$.

2.2 JSimil Profiles

Profiles are the main novelty of the proposed clone detector and they are, indeed, key to JSimil flexibility.

Profiles contain parameters that allow the user to fine-tune the behavior of the system by adjusting matching thresholds, and rules, output detail level, pruning thresholds and metrics weights.

JSimil distribution package includes JSimil Profile Manager, a tool for designing profiles that offers contextual help and shows several profile properties, both as normalized numeric values and as a Kiviat diagram called profile fingerprint (Figure 1):

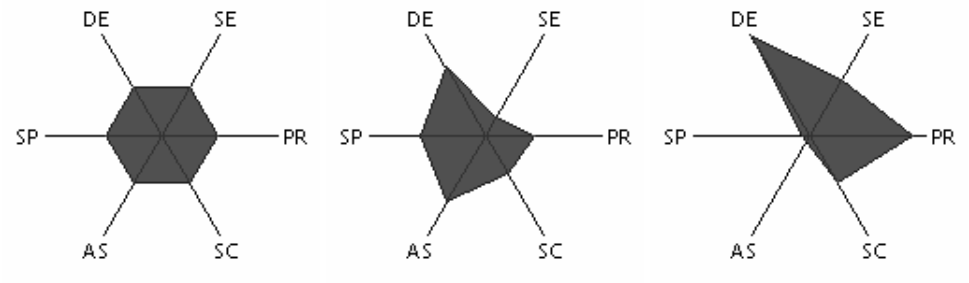

Figure 1: Fingerprints for default, student assignment plagiarism detection, and exhaustive profiles.

	JPlag	Moss	GPLAG	CP-Miner	CCFinder	Semantic Diff	JDiff	Baker and Manber's	JSimil
Applications									
Plagiarism	✓	✓	✓	✗	✗	✗	✗	✗	✓
Copy&Pasted code	✗	✗	✗	✓	✓	✗	✗	✓	✓
Diff	✗	✗	✗	✗	✗	✓	✓	✗	✓
Supported inputs									
Source code (java)	✓	✓	✓	✓	✓	✓	✓	✗	✓
Bytecode (jar/class)	✗	✗	✗	✗	✗	✓	✗	✓	✓
Mixed	✗	✗	✗	✗	✗	✓	✗	✗	✓
Supported outputs									
Plain text	✗	✓	✗	✓	✓	✓	✓	✓	✓
HTML	✓	✗	✓	✗	✗	✗	✗	✗	✓
XML	✗	✗	✗	✗	✗	✗	✗	✗	✓
Diff alike	✗	✗	✗	✗	✗	✓	✓	✓	✓
Robustness									
Textual changes	✓	✓	✓	✗	✓	✓	✓	✓	✓
Code reordering	✗	✗	✓	✗	✗	✓	✗	✗	✓
Code insertion	✗	✗	✓	✗	✗	✓	✗	✗	✓
Optimizations									
Parallelized	✗	✗	✗	✗	✗	✗	✗	✗	✓
Supports pruning	✗	✗	✓	✗	✗	✗	✗	✗	✓
Hierarchical matching	✗	✗	✗	✗	✗	✗	✗	✗	✓
Customization									
System behavior	✗	✗	✗	✗	✗	✗	✗	✗	✓
Matching rules	✗	✗	✗	✗	✗	✗	✗	✗	✓
Output detail	✗	✗	✗	✗	✗	✗	✗	✗	✓
Pruning thresholds	✗	✗	✓	✗	✗	✗	✗	✗	✓

Figure 2: Qualitative comparison of existing clone detectors and JSimil.

- Speed (SP) measures the number of used options that help reduce processing time.
- Detail (DE) measures the amount of data that will be generated.
- Sensibility (SE) measures how much partial results may influence global results.
- Precision (PR) measures how small the margin of error will be.
- Specialization (SC) measures the amount of options with non-standard values.
- Assimilation (AS) measures the amount of previous knowledge provided by the user to obtain better results.

These properties correspond to the visible characteristics of the system behavior the user will be most concerned about.

3 EXPERIMENTAL COMPARISON

A table summarizing the capabilities of similarity detection systems is shown in Figure 2.

JSimil was able to obtain more accurate results than other system in the comparison of 222 student

assignments that was performed in order to detect plagiarism. JSimil matched the 222 student assignments, of 200KB source code each on average, among themselves (a total of 24753 program-program matches) in 30 seconds using JSimil plagiarism detection profile, which corresponds to a millisecond per match on an mid-range Intel Quad Core personal computer.

4 CONCLUSIONS

We have described a Java code clone detector system that uses a novel hierarchical matching technique that solves issues that affect similar existing systems and offers advantages over them, such as: support for all Java constructs, the possibility of comparing programs when only bytecode is available, browsing results, a parallelized implementation, and pruning non-significant matches to reduce processing time.

The profiles allow users to adjust system behavior so unwanted features may be removed and algorithm adjustments can be made.

The proposed algorithm is not sensitive to common obfuscation and plagiarism concealing techniques that other systems are sensitive to. In fact, the experimental results revealed that JSimil outperforms existing systems as it is able to detect similarities they cannot.

Currently, JSimil matches hierarchies of Java bytecode whose leaf nodes are sets of metrics computed from basic blocks. JSimil can be extended by giving the system a description of the hierarchy to be used to match data from different sources using the same profiles. This will give researchers the possibility of developing general profiles (for example, for difference detection or similarity detection) that would define how to match any kind of hierarchical data.

REFERENCES

Apiwattanapong, T., Orso, A., and Harrold, M. J. (2007). JDiff: a differencing technique and tool for object-oriented programs. *Automated Software Engineering*, 14(1):3–36.

Baker, B. S. and Manber, U. (1998). Deducing similarities in java sources from bytecodes. In *Proc. of Usenix Annual Technical Conference*, pages 179–190.

Belkhouche, B., Nix, A., and Hassell, J. (2004). Plagiarism detection in software designs. In *Proc. of the 42nd Annual Southeast Regional Conference*, pages 207–211.

Cosma, G. and Joy, M. (2006). Source-code plagiarism: a UK academic perspective. Technical Report 422, University of Warwick.

Dunsmore, H. E. (1984). Software metrics: an overview of an evolving methodology. *Information Processing and Management*, 20(1-2):183–192.

Jackson, D. and Ladd, D. A. (1994.). Semantic Diff: a tool for summarizing the effects of modifications. In *Proc. of the International Conference on Software Maintenance*, pages 243–252.

Kamiya, T., Kusumoto, S., and Inoue, K. (2002). CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670.

Krinke, J. (2001). Identifying similar code with program dependence graphs. In *Proc. of the 8th Working Conference on Reverse Engineering*, pages 301–309.

Li, Z., Lu, S., Myagmar, S., and Zhou, Y. (2006). CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(2):176–192.

Liu, C., Chen, C., and Han, J. (2006). GPLAG: Detection of software plagiarism by program dependence graph analysis. In *Proc. of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 872–881.

Prechelt, L., Malpohl, G., and Philippsen, M. (2000). JPlag: Finding plagiarism among a set of programs. Technical Report 2000-1, University of Karlsruhe.

Schleimer, S., Wilkerson, D. S., and Aiken, A. (2003). Winnowing: Local algorithms for document fingerprinting. In *Proc. of the 22nd ACM SIGMOD International Conference on Management of Data*, pages 76–85.

Tairas, R. (2008). Clone maintenance through analysis and refactoring. In *Proc. of the 2008 Foundations of Software Engineering Doctoral Symposium*, pages 29–32.

Weiser, M. (1981). Program slicing. In *Proc. of the 5th International Conference on Software Engineering*, pages 439–449.

Wise, M. J. (1996). YAP3: Improved detection of similarities in computer program and other texts. In *Proc. of the 27th SIGCSE Technical Symposium on Computer Science Education*, pages 130–134.