

Searching for Configurations in Clone Evaluation A Replication Study

Chaoyong Ragkhitwetsagul¹, Matheus Paixao¹, Manal Adham¹
Saheed Busari¹, Jens Krinke¹ and John H. Drake²

¹University College London, ²Queen Mary University of London

Abstract. Clone detection is the process of finding duplicated code within a software code base in an automated manner. It is useful in several areas of software development such as code quality analysis, bug detection, and program understanding. We replicate a study of a genetic-algorithm based framework that optimises parameters for clone agreement (EvaClone). We apply the framework to 14 releases of Mockito, a Java mocking framework. We observe that the optimised parameters outperform the tools' default parameters in term of clone agreement by 19.91% to 66.43%. However, the framework gives undesirable results in term of clone quality. EvaClone either maximises or minimises a number of clones in order to achieve the highest agreement resulting in more false positives or false negatives introduced consequently.

1 Introduction

Code cloning is a common activity in software development. Clones can be created by reuse of well-written code or adaptation of functionality from existing code, and may lead to software maintenance issues. Numerous tools exist to detect clones in a given software system [4, 8, 10]. Not only do these tools differ in their detection approach, but they also come with a number of parameters to choose from which greatly affect their sensitivity [7]. The oracle problem in clone detection is the absence of the possibility to establish a ground truth, i.e. knowing if code is actually cloned. Therefore, multiple clone detectors are often used on the assumption that it is more likely that code is actually cloned when multiple clone detectors agree.

We perform a replication study of EvaClone [11] which uses a Genetic Algorithm to optimise clone detection tools parameters to maximise clone agreement, but in a different settings. We select four tools for this study: CCFinder [6], Deckard [5], NiCad [9], and Simian [2] and apply the framework to only a single subject, Mockito [1] (a mocking framework for unit testing within Java), over its 14 major releases. This experimental settings have not been explored yet in the previous study.

2 Optimising Parameters of Clone Detectors

Previous work by Wang et al. [11] has shown that a Genetic Algorithm (GA) is able to find a set of parameter values that maximise agreement between an ensemble of clone detection tools. They show that the derived optimised parameters

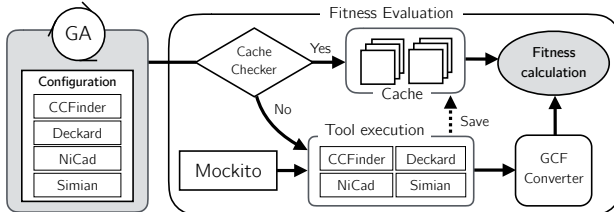


Fig. 1. A framework for optimising parameters of clone detectors using a GA

provide better agreement among tools compared to using the tools’ default settings, which are often used in empirical investigations in the literature. In this study, we adopt their EvaClone framework to search for configurations which maximise the level of agreement between the four clone detection tools.

Fig. 1 presents a high-level overview of the system. Given predefined configuration settings X , each tool generates a clone report containing either clone pairs or clone clusters in its own specific format. These output files are then converted into a General Clone Format (GCF) [11] so that they can be analysed in the same way. This is followed by fitness calculation of a given configuration X based on number of agreed clone lines. The fitness function computes the level of agreement between n different tools applied to detect clones in a subject system. $AgreedLines[i]$ is the number of lines where exactly i tools agree that they are part of a clone:

$$F(X) = \frac{\sum_i^n (i \times AgreedLines[i])}{n \times \sum_i^n AgreedLines[i]}$$

To search the space of configurations, we program the GA to initially generate a population of 100 feasible solutions (99 random individuals and one individual as the default configuration). Each individual solution encodes values for the 25 parameters of the four tools. These solutions are evolved using selection, crossover and mutation to create better quality solutions guided by the fitness value in each iteration. The crossover and mutation rate are the same as in [11], set at 0.8 and 0.1 respectively. We choose an elitism rate at 0.25.

The clone detectors selected for this study are representatives of (1) commonly used clone detection tools in research, and (2) different clone detection techniques, including string-based (Simian), parser-based (NiCad), token-based (CCFinder) and tree-based (Deckard). We reuse the default configurations given in [11] for CCFinder, NiCad, and Simian. Deckard has no default configuration so we choose the default parameters used in a recent study [10].

3 Experimental Study

We collected 14 major releases of Mockito from Google Code and GitHub repositories as subjects for this study. A manual investigation of the source code and release notes shows that 2 Java class files from Apache Commons have been included in the system since release 1.0 (*EqualsBuilder* and *ReflectionEquals*). The files are constantly modified over releases so we treat them as a part of Mockito. However, there are 2 complete libraries (*cglib* and *asm*) embedded in

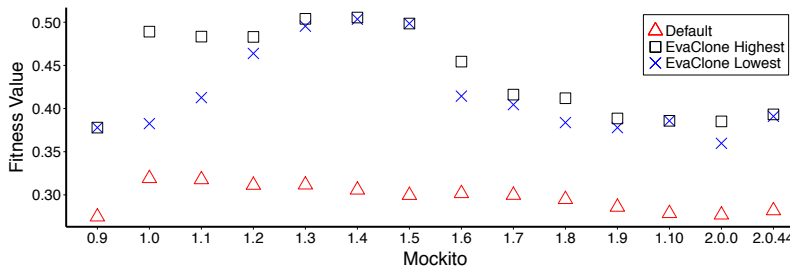
Table 1. Mockito releases, their size (SLOC), size increment (%Inc) and churn rates

Release	0.9	1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	1.10	2.0.0	2.0.44
SLOC	5500	6669	6784	6824	7239	7566	8364	8944	10143	12426	17876	22796	23555	25321
%Inc	N/A	21%	2%	1%	6%	5%	11%	7%	13%	23%	44%	28%	3%	8%
Insertions	N/A	1786	318	199	632	661	1494	1536	1445	5446	7151	7667	1452	13969
Deletions	N/A	618	204	157	218	335	656	989	245	3170	1765	2789	1577	11370

Mockito from release 1.5 to 1.9 which are used without modification. They make Mockito releases 1.5 to 1.9 grow three times bigger than release 1.4 and would introduce a strong bias to our results. Hence, we removed these two libraries out of the five releases. The size of the 14 releases (SLOC) after removal of the two libraries, and churn rates (inserted and deleted lines) are presented in Table 1.

We are interested in three research questions, which will be individually presented and discussed.

RQ1 (optimised agreement): *how do the default parameters perform in terms of clone agreement on each Mockito release compared to the optimised ones?* This is to measure how good the default configuration is for each release compared to its optimised counterpart. If we can find a better configuration than the default, it should be used for finding clones in each particular release.

**Fig. 2.** Comparison of optimised tools agreement (the highest and the lowest in 20 runs) to the default agreement over 14 Mockito releases

The experimental findings show that one can use EvaClone to find parameters that outperform the default parameters for all 14 releases. As depicted in Fig. 2, the optimised parameters always provide a higher level of tools agreement than the default ones. The lowest clone agreement obtained from EvaClone among 20 runs (represented using × symbol) is still higher than using the default configuration. We calculated percentage of agreement improvement and found that the optimised one always outperform the default configuration ranging from 19.91% up to 64.43%. These findings support the results of Wang et al. [11] that the default parameters offer a poor level of clone agreement and one should optimise the tools' configurations for every subject system or for every release of a system to maximise agreement.

RQ2 (stability of optimised parameters): *are there noticeable differences in the values of optimised parameters over releases?* Since each release of Mockito contains several changes made to its code base, we are interested to see what the impact of these modifications is to the optimised parameters. If the optimised parameters are stable over releases, it means that we can use the same optimised

Table 2. Clone detection tools with their default configurations (DF) and optimised configurations per release. Bold parameters are dominant in each release (i.e. no variation found among 20 runs)

Tool	Parameter	DF	Optimised														
			0.9	1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	1.10	2.0.0	2.0.44	
CCFinder	MinToken	50	10	70	70	70	80	80	80	80	10	10	10	10	10	10	
	TKS	12	10	16	18	19	18	18	19	20	14	17	10	10	10	10	
Deckard	MinToken	30	30	50	50	50	50	50	50	50	50	50	50	50	50	50	
	Stride	5	inf	8	8	8	5	8	8	8	16	5	inf	inf	inf	inf	
	Similarity	0.9	0.9	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.95	1.0	0.9	0.9	0.9	0.9	
NiCad	MinLine	6	5	7	7	7	6	6	6	7	6	5	5	5	5	5	
	MaxLine	1000	200	100	100	400	400	200	200	200	200	100	100	100	200	200	
	UPI	0.3	0.3	0.0	0.1	0.0	0.0	0.1	0.1	0.0	0.3	0.1	0.3	0.3	0.3	0.3	
	Blind	0	1	0	0	0	0	0	0	1	1	1	1	1	1	1	
	Abstract	0	4	6	6	6	6	5	5	6	6	2	4	4	4	4	
Simian	ignoreCurlyBraces	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	
	ignoreIdentifiers	0	1	0	0	0	0	0	0	0	1	1	1	1	1	1	
	ignoreIdentifierCase	0	*	*	*	*	*	*	*	*	*	*	*	*	*	*	
	ignoreStrings	0	1	0	0	0	0	0	0	0	1	0	*	*	*	*	
	ignoreStringCase	1	*	1	1	0	0	0	0	0	0	*	*	*	*	*	
	ignoreNumbers	0	1	0	1	0	1	1	0	1	1	0	*	*	*	*	
	ignoreCharacters	0	0	0	1	0	0	0	1	0	0	1	*	*	*	*	
	ignoreCharacterCase	1	0	0	*	1	1	0	*	1	1	*	*	*	*	*	
	ignoreLiterals	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
	ignoreSubtypeNames	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	ignoreModifiers	1	1	1	0	1	0	0	0	0	0	0	1	1	1	1	1
	ignoreVariableNames	0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	1
	balanceParentheses	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0
	balanceSquareBrackets	0	1	0	0	0	1	1	0	1	1	1	1	1	1	1	0
MinLine	6	5	6	6	6	6	6	6	6	6	7	7	5	5	5	5	

parameters to detect clones in any Mockito release. If not, it means that one may need to optimise the parameters for each individual release.

With 20 GA runs for each release, we found several sets of distinct parameter settings that can achieve the same highest clone agreement level. Among sets of these equally-performing optimised parameters, we select one that has minimum amount of change from the optimised parameters chosen in the previous release¹ (using Euclidean distance). This method maximises the stability of optimised parameters over all releases. The optimised parameters over 14 Mockito releases are reported in Table 2 and can be used as a guideline for setting the parameters of these tools in further studies of clones or clone evolution in Mockito. We can see that none of the optimised parameters is stable over all releases. However, if we inspect each tool’s settings individually, we notice some stability of specific parameters spanning over a number of releases. The parameters shown in bold (e.g. **50**, **inf**, **0.9**) represent parameters that are “dominant” in each specific release. Dominant parameters are those that have only a single value across all 20 runs. We can see that there are some parameters that are both dominant and stable over a number of releases. In addition, we observe that changing some parameters of Simian does not affect the tool’s behaviour at all since they are subsumed by another parameter. For example, the `ignoreNumbers`, `ignoreCharacters`, and `ignoreStrings` flags are subsumed by a more general `ignoreLiterals` flag. These parameters can be changed without any effect if the `ignoreLiterals` flag is enabled. Their values are represented using `*` meaning they can be freely set to any value. We also found that changing the value of

¹ The full set of optimised parameters are at cragkhit.github.io/ssbsechallenge2016.

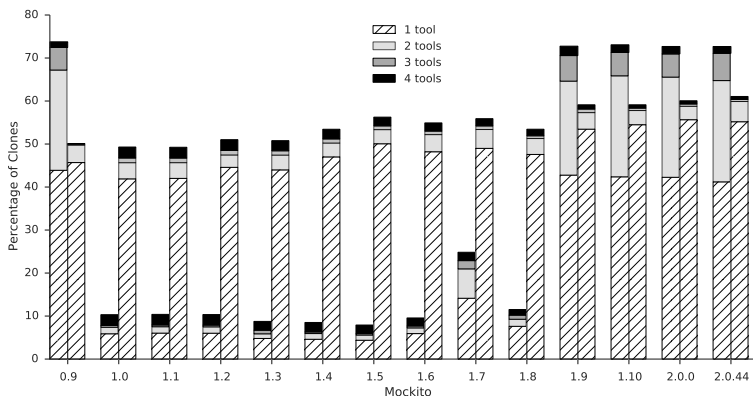


Fig. 3. Comparison of number of agreed clone lines (SLOC) for 1, 2, 3, and 4 tools reported by optimised parameters (left) and default parameters (right) in each release

`ignoreIdentifierCase` does not have any effect at all. In summary, for Mockito, the optimised parameters are observed to be varied over 14 releases with some stability in a specific region of releases. There is no single set of optimised parameters that work well across all releases.

RQ3 (clones over releases): *how many clones in Mockito are reported with the highest agreement over releases?* We would like to observe the number of clone lines (LOC) reported by the tools using optimised parameters in each release. This insight can support Mockito developers’ decision to perform code refactoring in future releases and future research studying clone detection.

The number of agreed clone lines detected by EvaClone using optimised parameters agreed by exactly 1, 2, 3, and 4 tools over 14 releases are presented in Fig. 3. We can clearly see that there are spikes in the number of agreed clone lines in release 0.9 and from release 1.9 onwards compared to releases 1.0–1.8. In releases 1.0–1.8, the highest agreement has been achieved by drastically *decreasing* the overall number of cloned lines, while for the other releases it has been achieved by *increasing* the overall number of cloned lines. Moreover, Fig. 3 shows that a large percentage (40%-50%) of the code is identified as cloned by only one tool. A manual investigation of the clone reports from the four tools revealed that the cloned lines reported by only one tool in every release are 80.8% generated by Deckard, 9.8% by Simian, 4.8% by CCFinder and 4.6% by NiCad for the optimised configurations and 87.9%, 10.9%, 0.7%, and 0.5% respectively for the default configurations. These fluctuations in the number of agreed clone lines reveal a weakness in the fitness function used by Wang et al. [11]: It increases agreement by significantly increasing or decreasing the number of cloned lines. The evaluation of the original study showed that EvaClone favours recall over precision [11], however, the drastic decrease in reported lines for releases 1.0–1.8 will reduce recall. Moreover, the large percentage of cloned lines in the default configuration suggests a low precision of at least one tool and the optimised configurations of release 0.9 and 1.9 onward decreases the precision even further.

This phenomenon is not a desirable result in terms of clone quality since there will be either too many false positives or false negatives. Since the fitness evaluation function is also a component of the framework, one should find a better fitness function in order to overcome this problem. For example, the fitness function must not only rely on the number of cloned lines, but also include other aspects like how often a line is found to be cloned to other places.

Our replication study produced more evidence that designing a general fitness function that works well in all situations is a difficult task. Approaches to solve this problem of designing proper fitness functions are emerging [3]. Because of the large fluctuations in the number of clones reported by the framework, we decided not to draw any conclusion about clones in Mockito from these results.

4 Conclusion

We performed a replication study by applying EvaClone, a framework for optimising clone detection tool's configurations using a Genetic Algorithm, with four tools to 14 Mockito releases in order to study the optimised parameters and how variations in the analysed data impact the results of the Genetic Algorithm.

The results show that the optimised parameters given by the framework achieve a higher clone agreement among the tools over all releases of Mockito. Some of the optimised parameters are observed to be dominant in a single release or over some releases but there is no parameter set that consistently superior over all releases. We also discover a weakness in the fitness evaluation function, as it increases agreement by significantly increasing or decreasing the number of cloned lines, producing more false positives or false negatives respectively.

References

1. Mockito. Online - <http://mockito.org>, accessed: 07.04.2016
2. Simian. Online - <http://www.harukizaemon.com/simian>, accessed: 07.04.2016
3. Amal, B., Kessentini, M., Bechikh, S., Dea, J., Said, L.B.: On the use of machine learning and search-based software engineering for ill-defined fitness function: A case study on software refactoring. In: SBSE '14 (2014)
4. Bellon, S., Koschke, R., Antoniol, G., Krinke, J., Merlo, E.: Comparison and evaluation of clone detection tools. TSE 33(9) (2007)
5. Jiang, L., Misherghi, G., Su, Z., Glondu, S.: DECKARD: Scalable and accurate tree-based detection of code clones. In: ICSE (2007)
6. Kamiya, T., Kusumoto, S., Inoue, K.: CCFinder: a multilinguistic token-based code clone detection system for large scale source code. TSE 28 (2002)
7. Mondal, M., Roy, C.K., Rahman, M.S., Saha, R.K., Krinke, J., Schneider, K.A.: Comparative stability of cloned and non-cloned code. In: SAC '12 (2012)
8. Roy, C.K., Cordy, J.R., Koschke, R.: Comparison and evaluation of code clone detection techniques and tools. Science of Computer Programming 74(7) (2009)
9. Roy, C., Cordy, J.: NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In: ICPC '08 (2008)
10. Svajlenko, J., Roy, C.K.: Evaluating modern clone detection tools. In: ICSME (2014)
11. Wang, T., Harman, M., Jia, Y., Krinke, J.: Searching for better configurations: a rigorous approach to clone evaluation. In: FSE '13 (2013)