Semantic Code Clone Detection for Enterprise Applications

Jan Svacina
Computer Science, Baylor University
Waco, Texas
jan_svacina2@baylor.edu

Jonathan Simmons
Computer Science, Baylor University
Waco, Texas
john_simmons2@baylor.edu

Tomas Cerny
Computer Science, Baylor University
Waco, Texas
tomas_cerny@baylor.edu

ABSTRACT

Enterprise systems are widely adopted across industries as methods of solving complex problems. As software complexity increases, the software's codebase becomes harder to manage and maintenance costs raise significantly. One such source of cost-raising complexity and code bloat is that of code clones. We proposed an approach to identify semantic code clones in enterprise frameworks by using control flow graphs (CFGs) and applying various proprietary similarity functions to compare enterprise targeted metadata for each pair of CFGs. This approach enables us to detect semantic code clones with high accuracy within a time complexity of $O(n^2)$ where n is equal to the number of CFGs composed in the enterprise application (usually around hundreds). We demonstrated our solution on a blind study utilizing a production enterprise application.

CCS CONCEPTS

• **Software and its engineering** → **Reusability**; **Software verification and validation**; *Software maintenance tools*; • **Theory of computation** → Graph algorithms analysis;

KEYWORDS

Source Code Analysis, Code Clone Detection, Semantic Clone, Enterprise Software, Software Engineering

ACM Reference Format:

Jan Svacina, Jonathan Simmons, and Tomas Cerny. 2020. Semantic Code Clone Detection for Enterprise Applications. In *The 35th ACM/SIGAPP Symposium on Applied Computing (SAC '20), March 30-April 3, 2020, Brno, Czech Republic.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3341105.3374117

1 INTRODUCTION

Enterprise applications are ubiquitous and essential for the modern world as they address numerous complex problems. These comprehensive problems require complex solutions using enterprise frameworks [1, 8], typically involving service-oriented architectures [8]. Due to such complexity, corporations pay up to 25% of the total costs for maintenance [7, 13]. Reducing the complexity of a codebase by lowering the saturation of code duplication incidents would speed up the bug fixing process.

We pose that semantic code clone detection can provide meaningful insights and results in the engineering of large enterprise applications and their respective costs. We propose a method in which we represent an enterprise system as a set of control flow graphs (CFG), where nodes are method statements and edges are calls between methods. Each CFG is associated with its semantic meaning in the system. These CFGs are compared with one another by a global similarity function, which runs in $O(n)$. Hence, comparing each CFG pair results in $O(n^2)$ combinations. This method keeps both the time complexity and the number of CFGs n low. Lastly, we conducted a blind study on a production enterprise application and verified the method's usability.

SAC '20, March 30-April 3, 2020, Brno, Czech Republic

ACM ISBN 978-1-4503-6866-7/20/03.
https://doi.org/10.1145/3341105.3374117

2 RELATED WORK

Code clones are blocks of code that have been copied from some source and pasted elsewhere, not necessarily always from the parent system [17]. There are 4 code clone types: type 1 - exact clones, type 2 - parameterized clones, type 3 - near-miss clones, and type 4 - semantic code clones [5, 12, 16, 17]. Our focus is on type 4 with the same behavior but different structure or method of approach.

The research into developing tools focused on enterprise systems with regards to code clone detection is underdeveloped and well needed. There are many semantic code clone detection tools [10, 14, 19]; however, they are scoped for single methods and cannot provide program-wide analysis nor provide enterprise metadata. These tools also are purely academic, making them unusable in real-world applications. Machine Learning tools exist [6, 18, 20, 21], but due to the need to train, the models are made useless in the real-world with radically changing enterprise application and fluid business logic needs. Due to the need to train the model before running them, run times exponentially increase on large codebases.

Some top of the line applications such as [11] works great for types 1 through 3; However, they fail to capture type 4 clones and lack in the ability to parse inter-method flows - when a method calls another method in the same program. Similarly, it fails to produce enterprise-specific context and information such as ours.

Program Representation: We aim to show that to effectively identify semantic code clones of an enterprise system, the optimal choice is to use Control-Flow Graphs (CFG) to represent the code segments in question. Our CFG represents the methods of a system where the edges are calls to other methods within the system. Our preference toward this method of program representation over others [2–4, 9] is to capture more meta-information regarding the context of the code clones or methods with regards to enterprise architecture.

3 PROPOSED METHOD

We examine the properties of each graph by applying a global similarity function in the Definition 3.1. Properties of the CFG bring higher value to identifying code clones because programs

Method Type	Similarity name	Weight	Properties
Controller	ctr	3	arguments, return type, HTTP method, security
Services	fc	1	arguments, return type
Message calls	rfc	2	IP, port, http type, arguments, return entity
Repository	rp	2	Arguments, Return type, Database operation

Table 1: Classification of properties

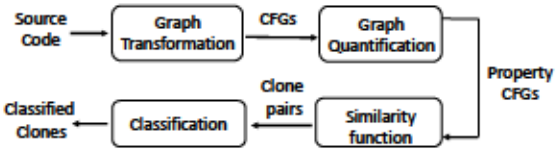

Figure 1: Schema of the algorithm

in enterprise systems tend to be repetitive in their structure but differ in meaning of the data in the input and output of the program Our approach consists of four stages: graph transformation, graph quantification, similarity comparison, and classification (Figure 1).

Definition 3.1 (Global similarity).

$$G(A,B) = \sum_{i=1}^{k} w_i \times sim_i(a_i, b_i) / \sum_{i=1}^{k} w_i$$

Where k is the number of attributes, w_i is the weight of importance of an attribute i, $sim_i(a_i, b_i)$ is a local similarity function taking attributes i of cases A and B, and w is a weight coefficient corresponding to the importance of the method type in the system.

In the first phase, we transform Java source code into a CFG. We scan the code for declared methods and internal method calls using Java Reflection and Javassist. Through the depth-first search we construct a graph for each method without a parent method call. Such a method is an entry-point to the enterprise application. Consider Listing 1 with an example system, where an endpoint method *create* in the *PosController* that calls *savePos* method in the service *PosService*. Next, *PosService* makes two procedure calls, first to a third-party API, and the second to a repository.

Next, we quantify each declared method by associating a set of properties P. The set of properties varies by the type of the method, as shown in the Table 1. Each method type a has set of associated properties that correspond to its role in the system. All types share argument types, return types, and metadata associated with the methods. The metadata is used to describe the purpose of the method in the system and associated properties.

In the next phase, we apply a global similarity function on properties P of two arbitrary CFGs, as shown in the Definition 3.1. The global similarity function G applies the similarity function for each Method type and multiplies the result by the weight coefficient that corresponds to the importance of a particular method type in the system, as shown in Table 1. Method type *Controller* has the highest significance because it denotes the input, output, and type of operation of a particular entry point. These properties tend to be unique in the system and therefore have a weight of 3. Method type *Repository* persists data to stable storage, and *Message calls* triggers actions usually via HTTP calls on some third party. Both of them

Classification Type	Global similarity	Characteristics
A	1.0 - 0.91	Same or differs in one function
B	0.9 - 0.81	Differs in 1-2 functions
C	0.8 - 0.71	Differs in 2-3 functions

Table 2: Classification of code clones

```
1  @Controller
2  public class PosController{
3      @PreAuthorize("hasRole('USER')")
4      @RequestMapping(value = "/pos", method =
           RequestMethod.POST)
5      public POS create (@RequestBody Pos pos) {
6          return service.save(pos);
7      }
8  }
```

Listing 1: Source code example

have a significant impact on the system and thus have a weight of 2. Method type *Services* have the least importance; service methods are usually reused within the system and thus weigh around 1.

We also provide definitions for various other local similarity functions as shown in Definition 3.2. The function takes as input properties of each method and evaluates each attribute. The similarity function for a controller method evaluates its arguments, return type, and HTTP method.

The similarity function *ctr* targets controller methods, which is a pattern for the method that handles input from the user. These methods are usually in REST formats and thus accept HTTP requests, hence the HTTP method type. Endpoints typically restrict access based on the roles of individual users within an enterprise system. We include the definition of permission roles into the metadata and thus include it in the similarity function.

The similarity function rfc compares all method calls from one system to another system, e.g., HTTP calls from CFG A to B are retrieved and proportioned into a ratio. When comparing two function calls, similarity function rfc takes into account IP, port, HTTP type, argument type, and return type. The similarity function fc does the same as rfc, but for method calls in the system.

Lastly, the similarity function rp compares methods working with databases by evaluating database operations (as in Table 1).

Definition 3.2.

$$sim(a_i, b_i) = ctr(a_i, b_i) + fc(a_i, b_i) + rfc(a_i, b_i) + rp(a_i, b_i)$$

The last stage is applying classification of the similarity between CFGs in the system. We classify graphs based on their global similarity into three categories as shown in Table 2. In the first category are the pairs of CFGs that are similar in all local similarity groups or differ in only one; these correspond to an interval of global similarity within [1.0, 0.91]. Category B has a larger tolerance, thus it encompasses code clones with one or two different local functions. Finally, category C has global similarity within the range [0.8, 0.71] and refers to pairs that differ in 2 to 3 functions.

4 CASE STUDY

We used the Central Texas Computational Thinking, Coding and Tinkering Enterprise Application (EA) for managing and evaluating test questions. This multilayered EA uses microservice architecture and Spring Boot [15]. It has a set of API methods using standard technology, controllers, services, repositories, and role-based access control. We analyzed the application in a blind study to detect

	CFG_A	CFG_B
Controller - *ctr*		
Arguments	ExamDTO	ExamDTO
Return type	Exam	Exam
HTTP method	POST	POST
Security	Admin	User
Service methods - *fc*	3	3
Rest methods - *rfc*	2	2
Repository - *rp*		
Database Operation	create	create
Arguments	Exam	Exam
Return type	Exam	Exam

Table 3: Example of properties of CFG_A and CFG_B

	Similarity	Weight	Weighed SIM
ctr	0.75	3	2.25
fc	1	1	1
rfc	1	2	2
rp	1	2	2
total	**3.75**		**7.25**

Table 4: Example of results of similarity function

Totals	Count
number of pairs	6
number of CFG	20
number of combinations	190

Table 5: Results summary

Category	Count
A	1
B	3
C	2

Table 6: Found clones

semantic code clones across the application. We present an example of derived properties from two CFGs, CFG_A and CFG_B derived from the EA in the Table 3. CFG_A and CFG_B have the same input (object *ExamDTO*), output (an object *Exam*), use the same HTTP method *POST*, use a function with the same inputs and outputs, and persist the same object type via *create* database operation.

Properties of both graphs CFG_A and CFG_B from the Table 3 were evaluated by local similarity functions as described in the Table 4. Similarity functions fc, rfc, rp give a full match result, whereas the similarity function ctr shows a lower match value due to the different signatures of RBAC security.

The Table 4 shows total similarity 3.75 and weight 7.25. The graphs CFG_A and CFG_B have a similarity 0.908. This value falls into category A on our scale from Table 2, thus, this is an example of two strongly semantically similar CFGs. We weighed all of the similarities in order to reflect their importance in the system as described in the proposed method.

Table 5 shows that we derived 20 CFGs from this EA, which comes out to 190 combinations in total. After applying similarity functions on each pair, 6 pairs had a similarity index above 0.71. They thus fell into respective categories as shown in Table 6, which shows that one pair of CFGs was strongly similar and 5 were fairly similar - which account for 3% of all combinations.

A low percentage is caused by having a high sensitivity or weight based on input/output types. Types of arguments and returns are important as the same constructs intended for other data types will tend to have the same behavior; both are necessary and cannot be removed from the application due to semantic similarity alone. This sensitivity with the weights avoids including structurally identical but semantically different CFGs as semantic clones in our results.

Threats to Validity: Experiments were not executed under various settings to optimize our experimental weights. Semantic clones require a low threshold to detect, thus the classification classes were set within the first third of our scale.

5 CONCLUSION

Our method for semantic code clone detection targets enterprise applications; an area of this field that has barely been studied. We utilized a blind study on a production enterprise application to confirm our method in finding the clones.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1854049.

REFERENCES

[1] 2019. Java Platform, Enterprise Edition (Java EE) | Oracle Technology Network | Oracle. https://www.oracle.com/java/technologies/java-ee-glance.html
[2] A. Agapitos, M. O'Neill, and Anthony Brabazon. 2011. Stateful program representations for evolving technical trading rules. 199–200.
[3] Hamid Abdul Basit, Simon J. Puglisi, William F. Smyth, Andrew Turpin, and Stan Jarzabek. 2007. Efficient Token Based Clone Detection with Flexible Tokenization. In *The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers (ESEC-FSE companion '07)*. ACM, New York, NY, USA, 513–516.
[4] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. 368–377.
[5] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. 2007. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering* 33, 9 (Sept. 2007), 577–591.
[6] L. Buch and A. Andrzejak. 2019. Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 95–104.
[7] Chris Doig. 2015. Calculating the total cost of ownership for enterprise software. https://www.cio.com/article/3005705/calculating-the-total-cost-of-ownership-for-enterprise-software.html
[8] Wu He and Li Da Xu. 2014. Integration of Distributed Enterprise Applications: A Survey. *IEEE Transactions on Industrial Informatics* 10, 1 (Feb. 2014), 35–42.
[9] Y. Higo and S. Kusumoto. 2009. Enhancing Quality of Code Clone Detection with Program Dependency Graph. In *2009 16th Working Conference on Reverse Engineering*. 315–316.
[10] T. Kamiya. 2013. Agec: An execution-semantic clone detection tool. In *2013 21st International Conference on Program Comprehension (ICPC)*. 227–229.
[11] T. Kamiya, S. Kusumoto, and K. Inoue. 2001. A Token based Code Clone Detection Technique and Its Evaluation. (Jan. 2001).
[12] R. Koschke, I. Baxter, M. Conradt, and J. Cordy. 2012. Software Clone Management Towards Industrial Application (Dagstuhl Seminar 12071). *Dagstuhl Reports* 2, 2 (2012), 21–57. http://drops.dagstuhl.de/opus/volltexte/2012/3477
[13] Michael Krigsman. 2019. Danger zone: Enterprise maintenance and support. https://www.zdnet.com/article/danger-zone-enterprise-maintenance-and-support/
[14] H. Nasirloo and F. Azimzadeh. 2018. Semantic code clone detection using abstract memory states and program dependency graphs. In *2018 4th International Conference on Web Research (ICWR)*. 19–27.
[15] Pivotal. 2019. Spring Framework. Retrieved Dec., 2019 from https://spring.io/
[16] C. Kumar Roy and James R Cordy. 2007. A Survey on Software Clone Detection Research. *School of Computing TR 2007-541, Queen's University* 115 (2007).
[17] N. Saini, S. Singh, and Suman. 2018. Code Clones: Detection and Management. *Procedia Computer Science* 132 (Jan. 2018), 718–727. http://www.sciencedirect.com/science/article/pii/S1877050918308123
[18] A. Sheneamer and J. Kalita. 2016. Semantic Clone Detection Using Machine Learning. In *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*. 1024–1028.
[19] Rajkumar Tekchandani, Rajesh Bhatia, and Maninder Singh. 2018. Semantic Code Clone Detection for Internet of Things Applications Using Reaching Definition and Liveness Analysis. *J. Supercomput.* 74, 9 (Sept. 2018), 4199–4226.
[20] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. 2016. Deep Learning Code Fragments for Code Clone Detection. In *The 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, New York, NY, USA, 87–98.
[21] H. Yu, W. Lam, L. Chen, G. Li, T. Xie, and Q. Wang. 2019. Neural Detection of Semantic Code Clones Via Tree-Based Convolution. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. 70–80.