Semantic Code Clone Detection Method for Distributed Enterprise Systems

Jan Svacina[a], Vincent Bushong[b], Dipta Das[c] and Tomas Cerny[d]
Computer Science, Baylor University, One Bear Pl, Waco, TX, U.S.A.

Keywords: Source Code Analysis, Code Clone Detection, Semantic Clone, Enterprise Applications, Code Smells.

Abstract: Conventional approaches to code clone detection consider systems from elementary construct perspectives, making it difficult to detect semantic clones. This paper argues that semantic clone detection could be improved for enterprise systems since they typically use well-established architectures and standards. Semantic clone detection is crucial for enterprises where software's codebase grows and evolves and maintenance costs rise significantly. While researchers have proposed many code clone detection techniques, there is a lack of solutions targeted explicitly toward enterprise systems and even fewer solutions dedicated to semantic clones. Semantic clones exhibit the same behavior between clone pairs but differ in the syntactic structure. This paper proposes a novel approach to detect semantic clones for enterprise frameworks. The driving idea is to transform a particular enterprise application into a control-flow graph representation. Next, various proprietary similarity functions are applied to compare targeted enterprise metadata for each pair of the control-flow graph fragment. As a result, we achieve to detect semantic clones with high accuracy and reasonable time complexity.

1 INTRODUCTION

Enterprise applications support many vital systems in our modern society (Fowler, 2002). As the addressed tasks grow in complexity, so too must the applications themselves. An array of enterprise frameworks provide a degree of standardization and stability to the development of these large applications (Oracle, 2020; He and Xu, 2014; Jin, 2014), but due to evolving requirements, changing technologies, and a constant backlog of legacy code, increases in complexity are inevitable. Complexity in software applications is ultimately felt in increased maintenance costs (Banker et al., 1993). Maintenance costs demand between 20% and 25% of corporations' total costs and fees, so the need for reducing complexity is clearly seen as companies fight to keep costs down (Krigsman, 2015; Doig, 2015).

Code clones are one source of complexity in software, which comprise between 10% to 23% of large codebases (Kapser and Godfrey, 2003; Roy et al., 2009). Code clones are duplicated or redundant snippets of code and they add complexity to the system. Code clones make updates to the codebase more time-consuming since the changes have to be made in multiple places. The effect is multiplied if there are bugs in the cloned code. The possibility of incomplete bug fixes increases, and developers need to scour potentially thousands or hundreds of thousands of code lines for the duplicate usages (Saini et al., 2018). When code clones are embedded in legacy systems, the problem further compounds; training developers on large legacy codebases is expensive and time-consuming, and it drains resources away from new development. Furthermore, attempting to fix code clone-filled legacy systems by assigning more developers comes with its own problems. Suppose the clones are not systematically identified and cataloged. In that case, new developers may waste hours discovering and rediscovering the same bugs and poor coding practices, resulting in no extra progress being made. Reducing code duplication in a codebase would save extra maintenance costs and prevent unneeded refactorings.

Developers can keep codebases more manageable with better practices enforced with the help of code clone detection (Saini et al., 2018). Of the varieties of

[a] https://orcid.org/0000-0002-6958-6455
[b] https://orcid.org/0000-0003-0475-4232
[c] https://orcid.org/0000-0001-8366-2453
[d] https://orcid.org/0000-0002-5882-5502

Svacina, J., Bushong, V., Das, D. and Cerny, T.
Semantic Code Clone Detection Method for Distributed Enterprise Systems.
DOI: 10.5220/0011032200003200
In *Proceedings of the 12th International Conference on Cloud Computing and Services Science (CLOSER 2022)*, pages 27-37
ISBN: 978-989-758-570-8; ISSN: 2184-5042

code clones, semantic code clones are of the most interest in large enterprise systems, since enterprise developers often employ superficially similar code snippets that are quite different in their effects and end results; for example, the process of fetching, processing, and storing data in different places appears structurally similar each time it is used, but semantically, it is different in different contexts. On the flip side, it is possible that different developers (or the same developer at different times) created semantically similar code in different places with different structural elements. These two facts lead us to conclude that semantic code clones are far more interesting and impactful to enterprise applications. Therefore, since code clones cause complexity and raise costs for enterprise applications, we pose that semantic code clone detection can provide meaningful results to the fields of development, quality assurance, and maintenance with respect to the software engineering of large enterprise applications.

In previous work (Svacina et al., 2020), we proposed a method to represent enterprise systems as a set of Control-Flow Graphs (CFG) where nodes are represented by method statements and edges by calls between methods. In this work, we increase the number of properties to CFGs to further improve the detection of potential code clones. With introduce weights of individual components in CFG, to reflect their semantic meaning in the system. For instance: what responsibilities the method has (database persistence, communication with the user, etc.), data attributed to the method (the input and output of the method), and the enterprise-specific implications of the method (security, entry-point, etc.). These enhanced CFGs are compared one with another by a global similarity function. Our similarity function runs in $O(n)$ and comparing each CFG pair results in $O(n^2)$ combinations. This method keeps both the time complexity and the number of CFGs n low. According to the results from our similarity function, we can categorize the code clones. This paper brings detailed results from a case study on an extensive third-party and heavily distributed enterprise application benchmark. We present results from the testing in Section 4.2, where we further elaborate on the false-positive ratio and other statistical values. Further, we conducted stress tests to establish the time efficiency of our approach, and we share results in Section 4.2.

This paper advances knowledge in the field of semantic code clone detection in enterprise applications. Such advancement is needed to better cope with enterprise software architectural degradation (Baabad et al., 2020) and enable providers to reduce maintenance costs or address several desired metrics. In Section 2, this paper outlines the state of the art. The other approaches to code clone detection focus on general code clone detection methodologies and then narrowing to the focus of semantic code clones. Section 3 outlines our proposed detection method, followed by Section 4 – a case study on an enterprise application. Lastly, we summarize our experiment results and highlight important notes and concepts obtained through our research.

2 RELATED WORK

In this section, we discuss previous work in categorizing and detecting code clones. In particular, we consider the code clone background and detection; next, we consider program representation.

2.1 Code Clone Background and Detection

Similar or identical code blocks are called code clones and are usually copied from some source (Saini et al., 2018)(Walker et al., 2020). As a result, they are code fragments that bear similarities in structure or function. Specifically, there are 4 code clone types or classes(Saini et al., 2018; Roy and Cordy, 2007; Bellon et al., 2007; Koschke et al., 2012):

- Type 1 - exact clones
- Type 2 - parameterized clones
- Type 3 - near-miss clones
- Type 4 - semantic code clones

Type 1 is self-descriptive: a block of code is a type 1 clone if an exact copy of the source code can be found elsewhere. Type 2 clones are similar to type 1, with the caveat that variables or function calls may have different names. Type 3 clones are copied fragments that have had some statements injected or removed while retaining a similar structure. The focus for this paper, type 4 clones, or semantic clones, are those which have the same behavior but different structure or method of approach (Saini et al., 2018). Type 4 clones are unique because while types 1-3 depend solely on the code structure, type 4 clones depend on the actual results. This is an important distinction, particularly between type 3 and 4 clones; depending on which statements were modified, type 3 clones can appear quite similar but achieve different results, while type 4 clones can superficially look different while serving the same purpose (Fowler, 2002).

Much research has been contributed to code clone detection, from type 1 to type 4 alike. The research

of tools focused on enterprise systems is underrepresented, and their need is well justified. One tool named DSCCD (Dynamic and Static Code Clone Detector) (Nasirloo and Azimzadeh, 2018) can detect semantic code clones - the most challenging of the four types to detect - at a rate of accuracy of 66%. The tool was developed to weigh the benefits of run time versus the reduction of false positives. However, a thorough analysis of a codebase for semantic code clone detection is naturally a computation-time consuming task. The case study done over DSCCD had 12 semantic code clones written into it, and in order to get their 66% overall accuracy rating, it took over 426 seconds in one study (Nasirloo and Azimzadeh, 2018). They utilized both dynamic and static analysis via Program Dependency Graphs (PDGs) and Abstract Memory States (AMS). The PDGs provide a higher level flow analysis for the semantic clone detection, and the AMS provides a quick, lower-level analysis (Nasirloo and Azimzadeh, 2018). While using AMS helped lessen the run time and lowered false-positive rates, AMS methods cannot handle scopes beyond single methods. This flaw renders them much less useful for enterprise applications, where the flow of method calls is more important for determining duplicate behavior due to separation of concerns making some methods extremely short.

For example, consider the following; let there be some method A that performs an action a. Create some new method B such that B calls A and returns the action a. It is trivial to see that B does the same exact thing as A. However, if the flows of these two methods are not analyzed and compared, they will not be tagged as semantic clones though it is clearly evident that they are. Semantic clone detection should be agnostic of lines of code.

One approach that was able to resolve many of these concerns was made with deep learning implementation. Research by White et al. (White et al., 2016) led to the development of a deep learning algorithm capable of analyzing massive codebases with extremely low false-positive rates. White et al. managed to get 93% true positives, taking only around 3 seconds using a model trained off of Abstract Syntax Trees (ASTs) and 35 seconds using a proprietary greedy algorithm. White et al. discovered dozens of types 1 through 3 clones in multiple systems and a small number (5) of type 4 clones. Their approach could analyze systems with exceptionally large codebases (such as Hibernate or Java JDK); the caveat is that running on a new system requires training the model on that system. The training takes an equally exceptional amount of time. Their example of using Java JDK, with over 500,000 lines of code (not unreasonable for large enterprise applications), took 2,977 seconds to train via the less accurate AST method and 14,965 seconds via their greedy learning method. Considering that enterprise applications are explicitly based on business logic that may change and the constant evolution of such systems, any given enterprise application will have to be constantly re-fed into the model for re-training to provide an accurate model for what code clones may look like, requiring a high time investment.

White et al. are not the first to attempt code clone detection via machine learning approaches. Yu et al. proposed a similar method by running two simultaneous neural networks over each pair of code snippets and categorized them into one of the types of clones (Yu et al., 2019). The approach provided by Yu et al. is powerful, with accuracy above 96%. However, a similar pitfall of complexity and training time renders the approach unsuitable for enterprise solutions. Their algorithm required several hours for training, so their utility as an enterprise application code clone-detection tool is severely limited.

Other attempts at machine learning-based analysis, such as that by Sheneamer et al. (Sheneamer and Kalita, 2016), which uses 15 machine learning algorithms, and Buch et al. (Buch and Andrzejak, 2019) are competitive in accuracy and performance at run time. However, they are system-specific in that they similarly need to be trained for each code base and do not consider the meta-information provided by enterprise structures and patterns.

The tool CCFinder by Kamiya et al. is an example of code clone detection that has been implemented and can discover types 1 through 3 code clones efficiently and effectively (Kamiya et al., 2001). Kamiya et al. focus heavily on maintainability and can help users determine if a code clone is safe to remove or reduce with impact to the system (Kamiya et al., 2001). However, they acknowledge inter-method flows are challenging to capture, and they focus exclusively on source code analysis. Thus this tool is not beneficial for large and complex enterprise systems that are dependent on inter-flow communication. So, even tools that are fantastic for types 1 through 3 code clones may not provide useful analysis for enterprise applications.

There are dozens of proposed code clone detection methods (Walker et al., 2020), many even specializing in semantic types. However, for the moment, these methods are strictly theoretical and academic, with no way to easily reproduce shared results, as the implementations and benchmarks have not been made available. Other tools such as Agec, by Kamiya et al.

(Kamiya, 2013) and the algorithm by Tekchandani et al. (Tekchandani et al., 2018) provide code clone detection solutions for type 4 specifically, but also fall short in these same ways.

2.2 Program Representation

When gathering semantic code clones of an enterprise system, a method of representing the program is needed. One option is to use Control-Flow Graphs (CFGs), a type of call graph where the nodes represent the system's methods and the edges are calls to other methods. We prefer this method of program representation over other methods (token-based (Basit and Jarzabek, 2007), Abstract Syntax Tree (AST)(Baxter et al., 1998), Abstract State Memory (ASM)(Agapitos et al., 2011), Program Dependency Graph (PDG)(Higo and Kusumoto, 2009), etc.) since it can capture more meta-information regarding the context of the code clones or methods with regards to enterprise architecture (i.e., component types of assessed code).

Awareness of system meta-information in the analysis process could open new perspectives to code clone detection in the enterprise systems. This could utilize GRASP patterns (Larman, 2003) and base on the application layer they are at (Fowler, 2002). This information that is only accessible in a higher-level abstraction such as a CFG could allow our program to eventually be filtered to only analyze service modules or controller modules, helping developers decide whether their service/controller/etc. Classes could be further split or merged to improve cohesion and reduce unnecessary coupling. Performance would greatly improve if users could filter code clone detection by concern. Besides, the code clone detection would be more meaningful knowing where in the application they are found and in which context. This could be accomplished through augmented CFGs with the additional meta-information used when developing enterprise applications (i.e., component type annotation, indicating a service, controller or entity). For instance, annotations in Java code, provided by the common and widely used Spring Framework (VMware, 2020), or similar means in other enterprise frameworks.

Our related work found AST and token-based approaches are popular program representation methods for finding code clones, both semantic and syntactic, so our choice to use CFGs is explicit and oriented particularly toward enterprise applications. We pose that the benefit to a CFG (a type of control-flow graph) over an AST is that AST provides too low-level of a depiction of the program and can consume too much memory to analyze. This is especially true since we are not interested in clone detection that requires such low-level, syntactic knowledge.

Code clone detection is a widely studied field, but it lacks in its depth concerning enterprise applications. It is not the case that business domain-related code clones have never been researched. However, many glances into this field left many questions unanswered and only emphasized the need for such a tool than provided one (Koschke et al., 2012).

3 PROPOSED METHOD

The proposed method to detect semantic code clones focuses primarily on the semantic meaning of a CFG rather than on the structure itself. The CFGs are used to represent the enterprise system. Semantic properties (hereby referred to only as properties) are derived from the metadata associated with each method in the form of configuration files or annotations. We examine each graph's properties by applying a global similarity function, as shown in the Definition 1. Properties of the CFG bring higher value to identifying code clones because programs 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 (Fowler, 2002). In other words, not every structural repetition of code is a code clone, but a semantic repetition is very likely to be a code clone. Our approach consists of four stages, graph transformation, graph quantification, similarity comparison, and classification, as shown in Figure 1.

Definition 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, AND $sim_i(a_i,b_i)$ is a local similarity function taking attributes i of cases A and B.

In the first transformation phase, we transform Java source code into a CFG. We used Java Reflection and Javassist (JBoss, 2020) libraries to scan the code for all declared methods; then, we get each method call within its body for each declared method. We used the depth-first search to construct a graph for each method that does not have a parent method call. Such a method is an entry-point to the enterprise application. Starting with these entry points, we expand the CFGs to include all the methods that get called, eventually covering all components of the system. For

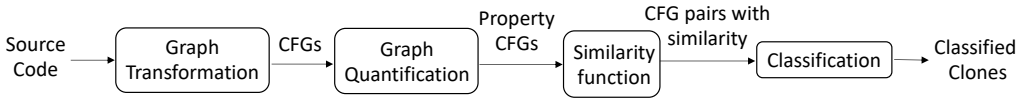

Figure 1: Schema of the algorithm.

illustration consider an example of a 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 some third-party API, and the second to *PosRepository*. The schema of this code is depicted in the Listing 1, and the resulting CFG is in Figure 2.

```
1   @Controller
2   public class PosController{
3       @Autowired
4       private PosService service;
5
6       @RequestMapping(value = "/pos",
7               method = RequestMethod.POST)
8       public Pos create(@RequestBody Pos p) {
9           return service.save(p);
10      }
11  }
12
13  @Service
14  public class PosService {
15      @Autowired
16      private PosRepository repository;
17
18      public Pos savePos(Pos pos){
19          Props p = restTemplate
20                      .postObject("/props");
21          pos.setProps(p);
22          return repository.save(pos);
23      }
24  }
25
26  @Repository
27  public interface PosRepository {
28      Pos save(Pos pos);
29  }
```

Listing 1: Source code example. Note `Pos` is a domain object representing a point-of-sale system.

In the next phase, we need to associate each CFG with a set of properties as shown in Figure 2. First, we identify the method types of each method involved in the CFG. We present the method type categories in Table 1. We can base our identification on analyzing standard enterprise annotations. Methods are associated with annotations that depict the type. For instance, annotations `@Controller` and `@RestController` defines controller method type in Spring Boot projects. Annotation `@Repository` signifies repository type. Next, we can associate a set of properties P with each method. The properties set is different for each method type with some overlapping properties. For instance, we associate each method type with a method name, return type, and arguments. The metadata depicts a method's role in the system (database connector, entrypoint, etc.). For

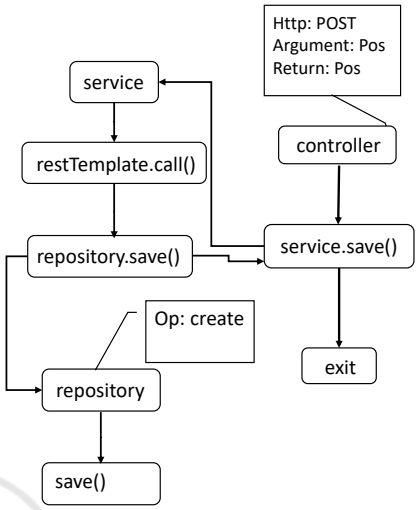

Figure 2: Example of control-flow graph.

instance, controller methods have properties HTTP method that signifies HTTP messages that the method handles. Thus, our utilization of CFGs provides additional meaning that determines individual component's roles in the system.

Table 1: Classification of properties.

Method Type	Similarity Name	Weight	Properties
Controller	ctr	0.4	arguments, return type, HTTP method, method name
Message calls	rfc	0.4	HTTP type, arguments return entity, method name
Repository	rp	0.2	arguments, return type, database operation, method name

Next, we assign a similarity score between 0 and 1 to each pair of CFGs, using the global similarity function G as given in Definition 1. The function depends on each method time, multiplying each result by the weight coefficient found in Table 1; this weight represents that method type's relative importance. *Con-*

troller and *Message call* methods are weighted highly at 0.4, since they respectively define the API of the program's entry points and the locations where external interactions are made, both of which highly impact a system's behavior. *Repository*-type methods persist data to stable storage, and while not insignificant, their properties are often subsumed into the corresponding *Controller* methods, so they are weighted lower at 0.2.

The usage of the similarity functions for each method type is shown in Definition 2. All three functions consider common elements such as the method name, arguments, and return type, but each is differentiated by specific considerations for each method type.

Definition 2.

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

The *ctr* function targets *Controller* methods, which represent endpoints that accept input to the service. REST-based endpoints are often differentiated in purpose by the HTTP method type (`GET`, `POST`, `PUT`, `DELETE`), so HTTP type is considered here. Additionally, any Role-Based Access Control measures are most often applied at the endpoint level. Therefore, we examine any required roles to gain access to the endpoint (such as *user* or *admin*) when considering similarity of controllers.

Remote function calls are governed by the *rfc* function. To compare these function calls, *rfc* specifically considers any IP addresses and ports involved, as well as the HTTP type and type and number of arguments of the call and the expected return type from the call.

When comparing two function calls, similarity function *rfc* takes into account IP address, port, HTTP type, argument types, and return type. Lastly, the similarity function *rp* compares methods working with databases by evaluating database operations, as shown in Table 1.

Once the similarity scores are calculated, the last step is to classify the CFG pairs. Different ranges of the similarity value correspond to three different categorizations: identical clones or clones that differ in one property, clones that differ in multiple properties, and non-clones. The exact ranges are given in Table 2.

Table 2: Classification of code clones and non-clones.

Classification Type	Global Similarity	Characteristics
A	1.0 - 0.91	Same or differs in one property
B	0.9 - 0.81	Differs in multiple properties in one method type
Non-clone	0.8 - 0.0	Not considered a clone

4 CASE STUDY

We conducted the following analysis to prove that the global and local similarity model with control-flow graphs' properties works sufficiently in finding semantic code clones in real microservice systems.

4.1 The Benchmark

To avoid any bias, for our case study, we used a public, third-party, medium-size microservice benchmark system developed by Zhou et al. (Zhou et al., 2018b). The benchmark uses microservice architecture and Spring Boot (VMware, 2020) with a set of API methods using standard procedures of multilayered applications such as controllers, services, repositories. The benchmark depicted in Figure 3 is composed of 37 microservices, and it provides comprehensive functionality for ticket train purchase (Zhou et al., 2018b). The authors created the application to analyze log outputs from the running application to detect faults (Zhou et al., 2019; Zhou et al., 2018; Zhou et al., 2018a). We analyzed the application in a study to detect semantic code clones across the application. In the next part, we will discuss an example of a semantic code clone and our study's overall results.

4.2 The Study

We present an example of derived properties from two CFGs. Both the CFG_A and CFG_B are derived from the benchmark as shown in Figure 4. Both of the CFGs have the same input (*String orderId*), but differs in output (objects *FoodOrder* and *Order*), use the same HTTP method *GET* and fetch the object type with the same database operation *READ* and same parameter $orderId$.

Properties of both graphs CFG_A and CFG_B from the Table 4 were evaluated by local similarity functions detailed in Table 3. Similarity function rfc give a full match result, whereas the similarity function ctr

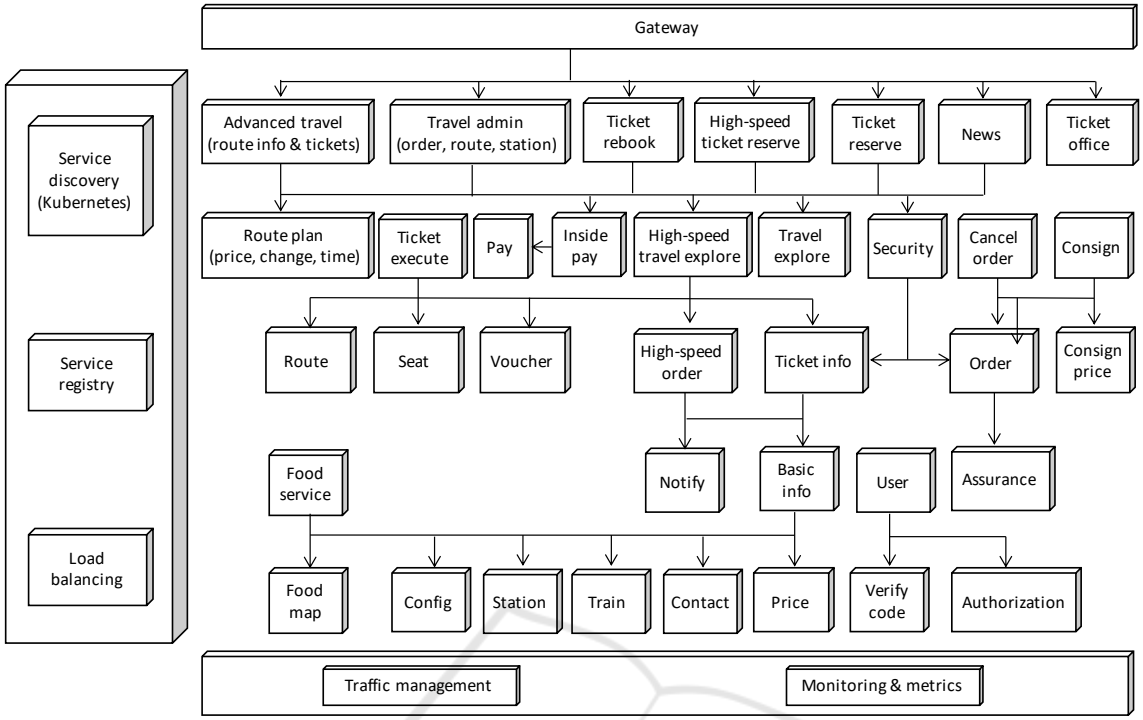

Figure 3: Benchmark microservice overview (Zhou et al., 2018b).

Table 3: Similarity function results of benchmark system.

Method Type	Abbr.	Similarity	Weight	Weighed SIM
Controller	ctr	0.66	0.4	0.264
Message calls	rfc	1	0.4	0.4
Repository	rp	0.5	0.2	0.1
total		**2.16**		**0.764**

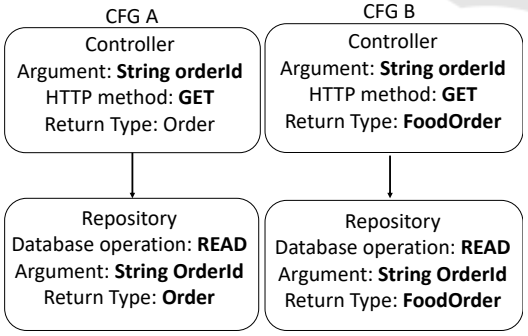

Figure 4: Example of properties of 2 CFGs (Note: in this particular case there are no REST methods).

and *rp* shows a lower match value due to the different return type.

Table 3 shows total similarity 2.16 and weighted similarity 0.764, or 76.4%. On our scale defined in the previous section and highlighted in Table 2 this value does not fall under the Type A or Type B category. Thus, this is not an example of a code clone. We weighed all of the similarities in order to reflect their importance in the system. For example, controllers are critical since they define what data is accepted and produced. Methods working databases and services that make REST calls also have high significance as these operations are specific to the enterprise system's business rules.

We derived all 238 CFGs from the TrainTicket application, which comes out to 27221 combinations in total. After applying similarity functions on each pair, there are 56 CFGs making 28 pairs that we classified as code clones. They fell into respective categories as shown in Table 5, which shows that 2 pairs of CFGs were strongly similar and 26 were fairly similar. Strongly similar accounts for 0.84% of all CFGs.

We verified our approach by manual review of the TrainTicket application by multiple reviewers. We divided reviewers into two groups. The first group verified results produced by our method, and the second group gathered manually results from TrainTicket without knowing results from our approach. The

Table 4: First column depicts statistical values of true-positive (TP), false-positive (FP), true-negative (TN) and false-negative (FN). Second column shows values for automatic approach, manual approach is depicted in the third column.

	Automatic	Manual
TP	28	24
FP	10	0
TN	27183	27193
FN	0	4
Accuracy	0.999	0.999
Precision	0.736	1
Recall	1	0.85

Table 5: CFG clones percentage.

Clone Type	Total Nr	Percentage
Type A	2	0.84 %
Type B	26	10.92 %
No Clone	202	88.24 %

first group distinguished our method's true and false positives, while the second group established missed code clones by our method. Results from Table 4 shows that our method found all code clones present in TrainTicket, while wrongly categorizing 10 code samples as clones. Manual review missed 4 code clones in the codebase. This shows that our method has a tendency to include false positives where more than one in three is not a code clone, but it ultimately includes all code clones in the application.

We also calculated statistical values of accuracy, precision, and recall. Accuracy is a measure of correctly classified cases among all cases. Both manual and automatic approaches have almost perfect accuracy due to a high number of combinations. More telling is precision and recall values. Precision shows the ratio between relevant and retrieved instances, and our method has a competitive precision value of 0.736. On the other hand, recall models the ability to identify only relevant code clones, and here our method proved to be more competitive than the manual approach.

This is in part caused by having a high sensitivity or weight based on input and output types. Types of arguments and return values are important because the same constructs intended for other data types will tend to have the exact same behavior; for example, a repository method to save a question to the questions table will semantically behave the same as a repository method to save a test to the tests table, and 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

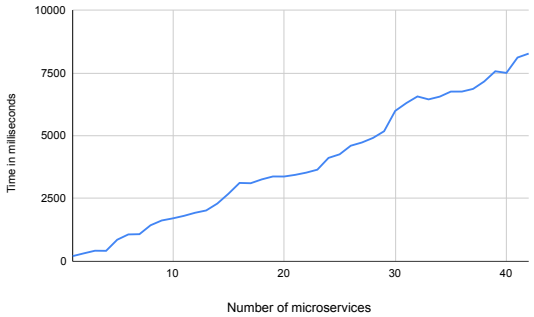

Figure 5: Time analysis of the approach with respect to number of microservices.

clones in our results.

We also look at the distribution of code clones in microservices. We associated the Type-A and Type-B clones with a particular microservice. Next, we calculated the proportions shared with other microservices. Table 6 shows microservices with type A and B clones. There are 13 microservice pairs that share some CFGs as column Nr. depicts. This column is only for reference purposes. The second and third columns show a pair of microservices that are similar to each other. The first row of the Table 6 shows that ts-contacts-service shares 37.5% of CFGs with admin-basic-info-service. Microservice pairs on row 6, 7, and 8 share substantial functionality. Pairs in rows 8 - 13 shows microservices that share CFGs one with the other, creating a cluster of the same functionality. Pairs in rows 2 - 5 show how one microservice can be similar to multiple other microservices.

Table 6: CFG Type A and B clones per module distribution in the benchmark from Figure 3.

Nr	MS A	MS B	Sim
1	ts-contacts	admin-basic-info	37.5 %
2	ts-config	ts-train	16.6 %
3	ts-config	admin-basic-info	16.6 %
4	ts-config	ts-travel2	33.3 %
5	ts-config	ts-travel	33.3 %
6	ts-order-other	ts-order	87.5 %
7	ts-preserve	preserve-other	50.0 %
8	ts-security	ts-train	50.0 %
9	ts-security	ts-seat	16.6 %
10	ts-train	ts-seat	16.6 %
11	ts-train	ts-travel2	16.6 %
12	ts-train	ts-travel	16.6 %
13	ts-travel2	ts-travel	66.6 %

We also tested our approach for the time efficiency of our solution in terms of code analysis. We ran our solution on an increasingly higher number of microservices, starting from one and up to forty-two. It

enabled us to observe the time efficiency of our system on small, medium, and large-sized systems and potentially establish a trend for even larger projects. We used microservices from the TrainTicket project for our experiment. We run our analysis on each number of microservices 10 times to avoid any software or hardware deviations. We used a system running operating system Ubuntu 20.10 LTS with Intel processor 11th Gen Intel® Core™ i7-1165G7 @ 2.80GHz × 8, 32 GB RAM and 500 GB SSD. The results from the experiment are plotted in Figure 5. One microservice needed 208 milliseconds to analyze, while forty-two 8200 ms, which is approximately 200 ms per one additional microservice.
The case study with results can be found on GitHub at https://github.com/iresbaylor/semantic-code-clones.

4.3 Result Discussion

Our main goal was to verify that control-flow graphs with properties are a good match for semantic code clone identification, which we identified manually. We compared manual and automatic approaches and showed that our method tends to include false positives but can detect all code clones from the codebase, where a manual approach fell short. We were able to ensure the time complexity of $O(n^2)$, where n is the number of CFGs. This is due to deriving only CFGs from the whole code base and then comparing them with each other. We derived only 238 instances from the code base of thousands of lines of code. This reduction enabled the algorithm to process each microservice in roughly 200 ms. We were also able to cluster code clones related to microservices and show overlaps of functionality in percentage ratio.

4.4 Threats to Validity

The main validity threat is the way how we set the thresholds in our method and how extensible our method is for other frameworks. We also elaborate on our custom thresholds used in the proposed method.

4.4.1 Internal Threats

Our proposed method uses several constants for the detection. We set weights for local similarity functions that correspond to real importance in enterprise systems. However, we did not execute the experiment under various settings in an attempt to produce an optimal solution. Semantic-type clones require a low threshold in order to detect. Therefore we set the classification classes to be within the first third of our scale. We observed that found clones had a very distinctive nature; Type A clones were exact clones only, Type B clones delegated the task on another service. These distinctive characteristics ensure that we detect these occurrences only. We utilized our knowledge of enterprise systems and code clones to set these constants.

Our proposed method solely focuses on semantic code clones for three different types of methods that are associated with enterprise application layers. However, there can be other utility or helper methods within the application. Detecting code clones for those methods are more straightforward and extensively studied in previous research works.

Our solution relies on standard practices for enterprise application development. Thus non-standard practices can influence the accuracy of our method. For example, it is possible to perform a delete operation on the database through a `GET` API call, where standard practice is to use a `DELETE` API call. If two methods with different HTTP types perform the same operation, their similarity will be low and require a high threshold value to detect as a code clone. Similarly, two different methods with the same name, parameters, and HTTP type can perform entirely different operations. Their similarity value will be high and will cause false-positive detection. However, this indicates a poor coding practice and will require a low threshold value to be ignored from detection.

4.4.2 External Threats

We conducted the tests on a real-world application. We picked the application from this publication (Zhou et al., 2018c). Our proposed method utilizes enterprise standards, and thus it can be facilitated on any project that uses the same standards. We focused on Java and utilized Java Parser for static analysis, and our case study demonstrates that we can successfully analyze such a Java-based system following enterprise standards. Microservices can also be written in other languages, and if the language has a parser, the properties can be derived and later integrated with our interfaces. Since the benchmark system does follow enterprise standards, structurally similar systems in other languages can indeed be analyzed with our method if such parsers are available.

5 CONCLUSION

Our semantic code clone detection method targets enterprise applications, a massive industry, yet an area that has been sparsely studied. Our method CFG-based method captures broader information about the system regarding its architecture, which provides

wider means to analyze more criteria and calculate more metrics at a more efficient rate than if we were to use a storage-intensive method such as ASTs or tokens.

The goal of finding code clones in any codebase is not a trivial one; finding semantic code clones via some form of graph traversal is an NP-Complete problem. Therefore, our ability to produce reasonable results efficiently with the complexity of $O(n^2)$ is impressive when considering that our method of CFG generation produces only tens of operations n. Our method of building CFGs is also efficient, needing to scan the codebase only once using a depth-first search to check all methods and build out their children list.

Another benefit to our approach is the extensibility. When it comes to enterprise applications, extensibility is a marketable property. The success of microservice architectures stands behind the principle of extensibility. Thus, it would not be too much of an investment at both a micro and macro level to transform this tool into a microservice that other microservices could utilize (Walker and Cerny, 2020). The macro-extensibility can become a module in some other suite and is not the only type of extensibility present. More micro-extensibility exists since, to expand on this tool, developers need only to add new local similarity functions to capture new metrics or other kinds of enterprise framework properties. So, the inner workings of our tool itself are similarly extensible. A tool like ours could be an essential boon to quality assurance teams for software-providing companies worldwide.

For future work, new metrics or other programming language support could be added. For example, there could be a measurement for the system's procedural entropy by running checks on each git commit and calculating the degradation and code clone accumulation over time. In the future, we could implement the means to measuring greater distances between CFGs using the meta-information and enterprise design patterns to analyze whether a controller class is behaving too much like a service class or etc., the possibilities for introducing new metrics are endless thanks to our method of developing an enterprise application code clone detection tool using enterprise architecture methodologies.

ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1854049 and a grant from Red Hat Research.

REFERENCES

Agapitos, A., O'Neill, M., and Brabazon, A. (2011). Stateful program representations for evolving technical trading rules. pages 199–200.

Baabad, A., Zulzalil, H. B., Hassan, S., and Baharom, S. B. (2020). Software architecture degradation in open source software: A systematic literature review. *IEEE Access*, 8:173681–173709.

Banker, R., Datar, S., Kemerer, C., and Zweig, D. (1993). Software complexity and maintenance costs.

Basit, H. A. and Jarzabek, S. (2007). Efficient Token Based Clone Detection with Flexible Tokenization. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 513–516, New York, NY, USA. ACM. event-place: Dubrovnik, Croatia.

Baxter, I. D., Yahin, A., Moura, L., Sant'Anna, M., and Bier, L. (1998). Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377.

Bellon, S., Koschke, R., Antoniol, G., Krinke, J., and Merlo, E. (2007). Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, 33(9):577–591.

Buch, L. and Andrzejak, A. (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)*, pages 95–104.

Doig, C. (2015). Calculating the total cost of ownership for enterprise software.

Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., USA.

He, W. and Xu, L. D. (2014). Integration of Distributed Enterprise Applications: A Survey. *IEEE Transactions on Industrial Informatics*, 10(1):35–42.

Higo, Y. and Kusumoto, S. (2009). Enhancing Quality of Code Clone Detection with Program Dependency Graph. In *2009 16th Working Conference on Reverse Engineering*, pages 315–316.

JBoss (2020). Javassist : Java bytecode engineering toolkit. https://www.javassist.org. Accessed 14 August 2020.

Jin, A. (2014). DCOM Technical Overview.

Kamiya, T. (2013). Agec: An execution-semantic clone detection tool. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 227–229.

Kamiya, T., Kusumoto, S., and Inoue, K. (2001). A Token based Code Clone Detection Technique and Its Evaluation.

Kapser, C. and Godfrey, M. (2003). *Toward a Taxonomy of Clones in Source Code: A Case Study*.

Koschke, R., Baxter, I., Conradt, M., and Cordy, J. (2012). Software Clone Management Towards Industrial Application (Dagstuhl Seminar 12071). *Dagstuhl Reports*, 2(2):21–57.

Krigsman, M. (2015). Danger zone: Enterprise maintenance and support.

Larman, C. (2003). *Agile and Iterative Development: A Manager's Guide*. Pearson Education.

Nasirloo, H. and Azimzadeh, F. (2018). Semantic code clone detection using abstract memory states and program dependency graphs. In *2018 4th International Conference on Web Research (ICWR)*, pages 19–27.

Oracle (2020). Java Platform, Enterprise Edition (Java EE) | Oracle Technology Network | Oracle.

Roy, C., Cordy, J., and Koschke, R. (2009). Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495.

Roy, C. K. and Cordy, J. R. (2007). A Survey on Software Clone Detection Research. *School of Computing TR 2007-541, Queen's University*, 115.

Saini, N., Singh, S., and Suman (2018). Code Clones: Detection and Management. *Procedia Computer Science*, 132:718–727.

Sheneamer, A. and Kalita, J. (2016). Semantic Clone Detection Using Machine Learning. In *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 1024–1028.

Svacina, J., Simmons, J., and Cerný, T. (2020). Semantic code clone detection for enterprise applications. In Hung, C., Cerný, T., Shin, D., and Bechini, A., editors, *SAC '20: The 35th ACM/SIGAPP Symposium on Applied Computing, online event, [Brno, Czech Republic], March 30 - April 3, 2020*, pages 129–131. ACM.

Tekchandani, R., Bhatia, R., and Singh, M. (2018). Semantic code clone detection for Internet of Things applications using reaching definition and liveness analysis. *The Journal of Supercomputing*, 74(9):4199–4226.

VMware, I. (2020). Spring Projects.

Walker, A. and Cerny, T. (2020). On cloud computing infrastructure for existing code-clone detection algorithmss. *ACM SIGAPP Applied Computing Review*, 20(1):in print.

Walker, A., Cerny, T., and Song, E. (2020). Open-source tools and benchmarks for code-clone detection: past, present, and future trends. *ACM SIGAPP Applied Computing Review*, 19(4):28–39.

White, M., Tufano, M., Vendome, C., and Poshyvanyk, D. (2016). Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, pages 87–98, Singapore, Singapore. ACM Press.

Yu, H., Lam, W., Chen, L., Li, G., Xie, T., and Wang, Q. (2019). Neural Detection of Semantic Code Clones Via Tree-Based Convolution. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 70–80.

Zhou, X., Peng, X., Xie, T., Sun, J., Ji, C., Li, W., and Ding, D. (2018). Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, pages 1–1.

Zhou, X., Peng, X., Xie, T., Sun, J., Ji, C., Liu, D., Xiang, Q., and He, C. (2019). Latent error prediction and fault localization for microservice applications by learning from system trace logs. In Dumas, M., Pfahl, D., Apel, S., and Russo, A., editors, *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, pages 683–694. ACM.

Zhou, X., Peng, X., Xie, T., Sun, J., Li, W., Ji, C., and Ding, D. (2018a). Delta debugging microservice systems. In Huchard, M., Kästner, C., and Fraser, G., editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 802–807. ACM.

Zhou, X., Peng, X., Xie, T., Sun, J., Xu, C., Ji, C., and Zhao, W. (2018b). Benchmarking microservice systems for software engineering research. In Chaudron, M., Crnkovic, I., Chechik, M., and Harman, M., editors, *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 323–324. ACM.

Zhou, X., Peng, X., Xie, T., Sun, J., Xu, C., Ji, C., and Zhao, W. (2018c). Benchmarking microservice systems for software engineering research. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, page 323–324, New York, NY, USA. Association for Computing Machinery.