Reasoning about Global Clones

Scalable Semantic Clone Detection

Philipp Schugerl, Juergen Rilling
Department of Computer Science and Software Eng.
Concordia University
Montreal, Canada
{p_schuge, rilling}@cse.concordia.ca

Philippe Charland
System of Systems Section
Defence R&D Canada – Valcartier
Quebec, Canada
philippe.charland@drdc-rddc.gc.ca

Abstract— **The Semantic Web is slowly transforming the Web as we know it into a machine understandable pool of information that can be consumed and reasoned about by various clients. Source code is no exception to this trend and various communities have proposed standards to share code as linked data. With the availability of large amounts of open source code published in publicly accessible repositories, the introduction of massive horizontal scaling frameworks, and cloud computing infrastructures, a new era of software mining across information silos is reshaping the software engineering landscape. Given these technological advances, analyzing code at a global scale, across systems, projects and organizational boundaries, becomes feasible. In this paper, we introduce a clone detection algorithm and its implementation that can scale to such large global datasets, by modeling clones using description logic and applying a horizontal scaling Semantic Web reasoner. We demonstrate how our simple feature vector that only uses control statements, data types and method calls, can yield results similar to other popular clone detection tools. Our approach does not only allow us to reliably identify clones in a global context. By using a semantic reasoner, it also allows us to expand clone detection to a new class of semantic clones. We have compared our algorithm to some of the leading clone detection tools (DECKARD, CCFinder, JCD, and Simian) in order to validate our approach and show the differences in detected clones and performance.**

Keywords-clone detection; Semantic Web; reasoner; MapReduce

I. INTRODUCTION

Although recent work shows that not all code clones are dangerous or a bad practice at all [1], having knowledge about source code clones can be beneficial during both implementation and maintenance to lower the costs associated with these tasks [2]. While previous research has mainly focused on code clones on a per-project basis, the availability of Semantic Web standards and the resulting wider spread publication of code and its semantics as linked data have opened up new possibilities for a more global perspective of source code analysis [11].

Performance has always been a distinguishing factor among code clone detection approaches and most noteworthy tools have been implemented keeping performance in mind. Nevertheless, these tools usually scale vertically by adding more CPUs and memory to a single machine, but not horizontally when adding more machines. Therefore, they cannot be applied to the described global code clone problem that has to identify clones within large amounts of data. It is also not possible to run such tools on projects individually and combine the results at a later step as this would only lead to a collection of common clones (and not the identification of clones across project boundaries). In order to support horizontal forms of parallelization, typical algorithms that are known to scale horizontally for a given problem are adopted. A popular example of such an algorithm is the MapReduce framework which has been introduced by Google to aggregate and analyze the massive amounts of data generated by their web crawlers.

The Semantic Web has provided new tools and techniques that allow us to reason about information found on the Internet. Semantic Web reasoners are based on description logic and can automatically deduce information from stated facts. So far, clone detection approaches have made little use of the Semantic Web. The clone detection research community mainly uses the term “semantic-aware” to classify methodologies using program dependency graphs as mentioned in [23]. The work of Gabel et al. [25] is an example of such a semantic-aware approach that does not leverage the Semantic Web as an infrastructure.

In this paper, we introduce a novel approach to clone detection that is based on modeling source code using description logic and by applying a Semantic Web reasoner to identify similar code. Furthermore, it is in particular our novel feature vector using control statements, data types, and method calls that allows for an interesting global application of our code clone search.

The remainder of this paper is organized as follows. Section 2 gives background information about the three core topics of this paper, namely the MapReduce framework, Semantic Web, and clone detection. Section 3 describes invariance types of our clone model with respect to adding, removing, and changing statements in a code fragment. Section 4 elaborates on the reasoning process and how it scales horizontally. In order to validate our findings, we apply our methodology to the JDK and the Apache Commons catalog and compare our method to other clone detection tools in Section 5. Section 6 concludes with a summary of our approach and discusses future work.

II. Background

Clone detection over large amounts of data and across project boundaries on the Internet can only be achieved through a massive horizontal scaling approach. In this paper, we introduce a novel clone detection approach that is based on a Semantic Web reasoner implemented using the well scaling MapReduce framework.

A. Clone Detection

Clone detection techniques can generally be grouped by their representation of source code that is used to match code fragments. String-based clone detection tools compare files without taking into consideration their underlying semantics and therefore, have the advantage of working on any kind of file. Token-based approaches transform text into language specific tokens that can be matched using distance measures. Similarly, Abstract Syntax Tree (AST)-based methods include the semantics of the underlying code by fully parsing its structure according to the language specifications. Subtree matching is then performed to identify potential clones. At a byte/machine code level, one can identify clones by comparing compiler optimized instructions. Program Dependency Graph (PDG)-based tools analyze code at a higher structural level.

The following is a list of available clone detection tools, classified by their underlying detection approach:

- String (e.g., Simian [5], Duploc [6])
- Token (e.g., JPlag [7], CCFinder [8])
- AST (e.g., DECKARD [23], Baxter et al. [9])
- Byte code (e.g., JCD [10])
- PDG (Komondoor et al. [24])

Various studies comparing different clone detection methods exist. Burd and Bailey evaluated five clone detection techniques for maintenance [13]. Their findings suggest that CCFinder has one of the highest recognition rates of token-based tools. Koschke [14] analyzed various clone detection tools and found that AST-based methods, such as CloneDr (Baxter et al.), have the highest precision, while token-based approaches offer a better recall. In [15], it is argued that clone detection studies suffer from a lack of objectivity when annotating what constitutes a clone, since human reviewers are used. This is a finding also noted by Kapser in [16], who gives and excellent overview of currently used techniques and provides and empirical evaluation of code clone patterns.

In terms of large size code clone analysis (also referred to as Mega Software Engineering [11]) a distributed CCFinder has been implemented by Liverie et al. [12]. They have analyzed 400 million lines of code with a cluster of 80 machines. Their idea of applying CCFinder in a distributed manner through a master-slave architecture is not as fault tolerant as our MapReduce implementation and heavily relies on a fast shared file system. Consequently, their approach fails to detect clones other than those detected by CCFinder. In comparison to their method, our approach relies on standardized Semantic Web components that can be easily deployed in the cloud. [22] shows how an index for an instant code clone search can be built in a scalable way in seconds (measured using the JDK) by using AST vectors. Nevertheless, their proposed problem space of building an index is inherently simpler than identifying clone groups between projects in a scalable way, as introduced by our approach. DECKARD [23] is a distributed and scalable implementation using such an AST-based feature vector. As part of our approach, we replace most elements from [23] (e.g., declarations, conditions, increments) and introduce a new feature vector that not only contains simpler elements, but leads to a smaller and more efficient feature vector with the support for a semantic analysis of method calls.

B. MapReduce Framework

The MapReduce framework [17] is a program paradigm that has been used to implement horizontal scaling data driven applications. Data is split and distributed across multiple machines and each machine either performs a map or a reduce task. A master node is responsible for assigning data to idle machines and handling failure situations. In the map step, data is grouped by a key (in parallel) through a divide and conquer algorithm. The output is then written to a temporary (fast) storage.

The process can be represented as

$$(\alpha 1, \beta 1) \rightarrow \text{list}(\alpha 2, \beta 2) \qquad (1)$$

In the reduce step, each grouped data is processed ("reduced") in parallel to one or more output values. The step can be represented as

$$(\alpha 2, \text{list}(\beta 2)) \rightarrow \text{list}(\chi) \qquad (2)$$

There exist several prominent implementations of the MapReduce framework. The Apache Hadoop project [18] is an open source example that has reached version 2.0 and is known to be stable and fast [19]. It has also recently been integrated into Amazon's Elastic Cloud (EC2) [29] Web services that allow any user to create and rent clusters of Hadoop instances on demand.

C. Semantic Web

The Semantic Web [20] has emerged as a potential solution for addressing the ambiguity of data on the Internet by making Web content machine processable. It allows knowledge to be formally represented using logics such as description logic (DL). Semantic Web reasoners can infer logical consequences from asserted DL statements. In the context of this paper, the classification of ontologies is of particular interest. Therefore, the complete subsumption hierarchy between all concept names occurring within an ontology is calculated.

Urbani et al. [4] have shown that simple reasoning tasks can be performed using join-like operations within MapReduce. Their Webpie tool has been picked up by Mutharaju et al. [3], which provide an alternative formulation of the CEL [21] algorithm to fit within the MapReduce paradigm. Webpie has been shown to be scalable by calculating the closure of 100 billion triples [26].

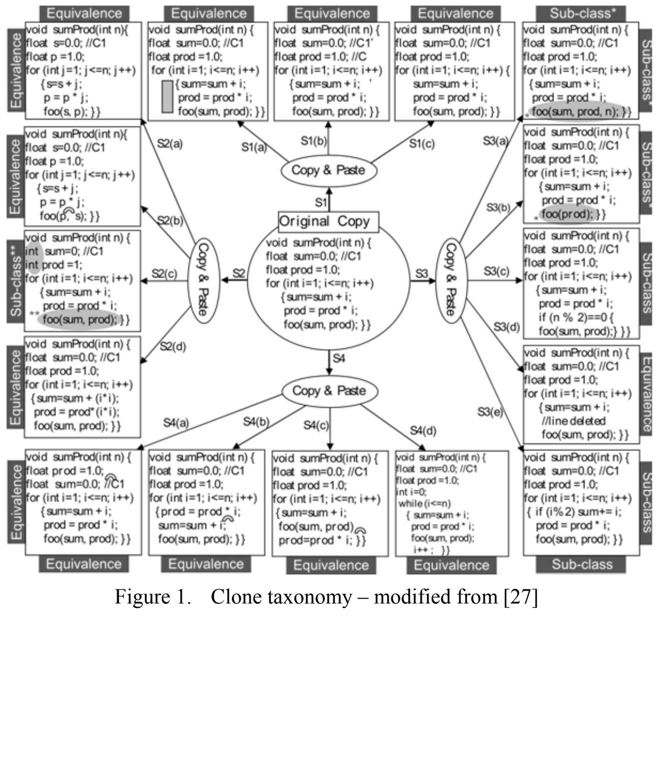
III. INVARIANCE

A crucial factor when trying to find duplicate code in massive amounts of data is the selection of the source code elements to be compared against each other. A compromise between granularity and size has to be found, in order to not hinder parallel processing. [23] uses a vector of AST elements to process source code. We argue that a much smaller vector is needed to reliably identify clones, if a more semantic processing of files is performed. Our approach is based on the data types used and methods called in control blocks as a compact but distinguishing factor between code. A block is therefore either a function/method, the biggest element of interest in our analysis, or a control block (condition or loop). While this approach is similar to AST-based implementations as it also relies on building an AST, the information used to identify what constitutes a clone is different. Instead of loosely matching AST identifiers, operations, and expressions potentially over the complete class, our approach only compares the set of data type names and method calls used over control blocks.

The introduced comparison of blocks by their data types used and methods called has an immediate benefit: the automatic invariance to certain code changes typical for code clones:

- Code order and use of parenthesis
- Renaming of identifiers
- Change of arithmetic operations and literals
- Formatting (spaces, etc.) and comments

We achieve further invariance of control block types by mapping switch and if statements as "conditions" and for and while statements as "loops", since they are interchangeable. The introduction of variables with identical types or multiple usage of the same method call are also covered, as we ignore those by using distinct sets of data types and method calls in our approach.



```
void sumProd(int n) {
float sum=0.0; //C1
float prod =1.0;
for (int i=1; i<=n; i++)
  {sum=sum + i;
   prod = prod * i;
   foo(sum, prod); }}
```

```
void sumProd(int n){
float s=0.0; //C1
float p =1.0;
for (int j=1; j<=n; j++)
  {s=s + j;
   p = p * j;
   foo(s, p); }}
```

```
void sumProd(int n) {
float sum=0.0; //C1
float prod =1.0;
 for (int i=1; i<=n; i++)
   {sum=sum + i;
    prod = prod * i;
    foo(sum, prod); }}
```

```
void sumProd(int n) {
float sum=0.0; //C1'
float prod =1.0; //C
for (int i=1; i<=n; i++)
  {sum=sum + i;  '
   prod = prod * i;
   foo(sum, prod); }}
```

```
void sumProd(int n) {
float sum=0.0; //C1
float prod =1.0;
for (int i=1; i<=n; i++) {
    sum=sum + i;
    prod = prod * i;
    foo(sum, prod); }}
```

```
void sumProd(int n) {
float sum=0.0; //C1
float prod =1.0;
for (int i=1; i<=n; i++)
  {sum=sum + i;
   prod = prod * i;
 * foo(sum, prod, n); }}
```

```
void sumProd(int n){
float s=0.0; //C1
float p =1.0;
for (int j=1; j<=n; j++)
  {s=s + j;
   p = p * j;
   foo(p, s); }}
```

```
void sumProd(int n) {
float sum=0.0; //C1
float prod =1.0;
for (int i=1; i<=n; i++)
    {sum=sum + i;
     prod = prod * i;
   * foo(prod); }}
```

```
void sumProd(int n) {
int sum=0; //C1
int prod =1;
for (int i=1; i<=n; i++)
   {sum=sum + i;
    prod = prod * i;
 ** foo(sum, prod); }}
```

```
void sumProd(int n) {
float sum=0.0; //C1
float prod =1.0;
for (int i=1; i<=n; i++)
   {sum=sum + i;
    prod = prod * i;
    if (n % 2)==0 {
    foo(sum, prod);} }}
```

```
void sumProd(int n) {
float sum=0.0; //C1
float prod =1.0;
for (int i=1; i<=n; i++)
 {sum=sum + (i*i);
  prod = prod*(i*i);
  foo(sum, prod); }}
```

```
void sumProd(int n) {
float sum=0.0; //C1
float prod =1.0;
for (int i=1; i<=n; i++)
    {sum=sum + i;
     //line deleted
     foo(sum, prod); }}
```

```
void sumProd(int n) {
float prod =1.0;
float sum=0.0; //C1
for (int i=1; i<=n; i++)
   {sum=sum + i;
    prod = prod * i;
    foo(sum, prod); }}
```

```
void sumProd(int n) {
float sum=0.0; //C1
float prod =1.0;
for (int i=1; i<=n; i++)
 {prod = prod * i;
  sum=sum + i;
  foo(sum, prod); }}
```

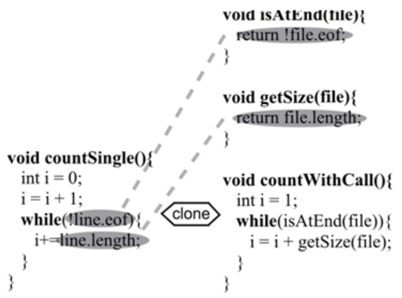
```
void sumProd(int n) {
float sum=0.0; //C1
float prod =1.0;
for (int i=1; i<=n; i++)
 {sum=sum + i;
  foo(sum, prod);
  prod=prod * i; }}
```

```
void sumProd(int n) {
float sum=0.0; //C1
float prod =1.0;
int i=0;
 while (i<=n)
   { sum=sum + i;
     prod = prod * i;
     foo(sum, prod);
      i++ ;  }}
```

```
void sumProd(int n) {
float sum=0.0; //C1
float prod =1.0;
for (int i=1; i<=n; i++)
{ if (i%2) sum+= i;
   prod = prod * i;
   foo(sum, prod); }}
```

Figure 1. Clone taxonomy – modified from [27]

This leaves the addition and removal of control blocks, as well as different data types and method calls, as the distinguishing elements for which a partial invariance is needed. We achieve such invariance automatically by the way we model our feature vector in description logic. Added or removed statements lead to a subclass relationship. We will discuss this type of invariance in the next section, as it relates to the reasoning services it needs.



```
void isAtEnd(file){
  return !file.eof;
}

void getSize(file){
  return file.length;
}
```

```
void countSingle(){
  int i = 0;
  i = i + 1;
  while(!line.eof){
    i+=line.length;
  }
}
```

```
void countWithCall(){
  int i = 1;
  while(isAtEnd(file)){
    i = i + getSize(file);
  }
}
```

Figure 2. Invariance to added sub-functions

Figure 1 shows different types of invariance that are needed for a robust detection of code clones. Most copy & paste changes introduced in Figure 1 are equivalent to the original copy, due to our model. S3(c) and S3(e) introduce a new control statement (*if*-block) and therefore, it can only be detected by using a reasoner. S4(d), on the other hand, remains trivially identical as we model *for* and *while* statements identically.

Special emphasis regarding our modifications has been put on S3(a), S3(b), and S2(c). The changes introduced in this code fragment might seem small, but the resulting change to the program flow can be significant. The methods *foo(float, float, int)*, *foo(float)*, and *foo(int, int)* can be overloaded with different functionality than the one provided by the original *foo(float, float)*. In our approach, each method is matched individually and the results are used during the reasoning process to recursively determine other resulting matches. In the previous example, this leads to a situation where the modified statement only matches the original copy if the different implementations of *foo* also match.

We also introduce a new type of invariance that has so far only been partially covered by byte code-based approaches, namely the invariance to the splitting of code into sub-functions. An example of such a clone is shown in Figure 2. In our approach, short methods contribute back their data types to the caller. In the example, this would mean that *file.eof* and *file.length* are merged into *countWithCall()*, making it easy to match it with the original *countSingle()*.

Note that the code in Figure 2 is simplified and would in theory still be detectable by most common clone detection tools, due to the small amount of code extracted into a sub-function. Nevertheless, more complex scenarios with complex (potentially nested) sub-functions remain undetectable by such tools. Byte code-based approaches are able to detect this class of clones if the compiler optimizes code in a way that the two methods have a similar byte code.

IV. Reasoning

To detect clones using a Semantic Web infrastructure, our model has to be transformed into concepts and relations. The OWL schema for modeling source code is rather simple: control blocks and data types are modeled as concepts, while roles represent their usage inside the block. An example of source code expressed in this syntax is shown below. The block name usually encodes the class and method name in order to be unique and allow for an easy identification of a corresponding statement. It is simplified in the example. BLCK_1 references one inner block BLCK_2 and calls the *toString()* method of class "RMI". It also uses a "java.lang.String" data type. BLCK_A is similar but has an inner BLCK_B. Block 1-2 and A-B are extracted from different files for which reason BLCK_2 and BLCK_B are defined separately.

$$\begin{aligned}
\text{BLCK\_1} \equiv\ & (\text{uses JAVA\_java\_lang\_String}) \wedge \\
& (\text{condition BLCK\_2}) \wedge \\
& (\text{calls JAVA\_RMI\_toString}) \\
\text{BLCK\_2} \equiv\ & (\text{uses JAVA\_Boolean}) \\
\text{BLCK\_A} \equiv\ & (\text{uses JAVA\_java\_lang\_String}) \wedge \\
& (\text{condition BLCK\_B}) \\
\text{BLCK\_B} \equiv\ & (\text{uses JAVA\_Boolean})
\end{aligned}$$

A reasoner can now simply infer that BLCK_2 and BLCK_B are in fact identical. This has the further consequence that BLCK_A is inferred as a subclass of BLCK1. This means that these blocks are clones of each other. Both class equivalence and subclass relationships determine a matching code clone in our methodology.

Methods are a common way to hide clones from detection. As discussed in the previous section and shown in Figure 2, splitting code into multiple sub-function/method calls is sufficient to prevent most clone detection tools from detecting this type of clones. In order to be able to detect such clones, one can leverage more advanced properties of the reasoner. For each function/method, we specify its used data types and blocks, like in the example below, as a subsumption. The reasoner can then infer that blocks calling this function/method inherit these elements.

$$\begin{aligned}
\text{BLCK\_1} \equiv\ & (\text{uses JAVA\_java\_lang\_String}) \wedge \\
& (\text{conditionalUses JAVA\_Boolean}) \wedge \\
& (\text{calls JAVA\_RMI\_toString}) \\
\text{BLCK\_C} \equiv\ & (\text{uses JAVA\_java\_lang\_String}) \wedge \\
& (\text{conditionalUses JAVA\_Boolean}) \wedge \\
& (\text{uses JAVA\_Int}) \\
\text{RETN\_X} \equiv\ & (\text{calls JAVA\_RMI\_toString}) \\
& \subseteq (\text{uses JAVA\_java\_lang\_String})
\end{aligned}$$

In the above example, "RMI_toString" is defined as using the data types "Int" and "java.lang.String". BLCK_1 calls this function/method. The reasoner can infer that BLCK_C is a subclass of BLCK_1, although the data type "Int" is never used in BLCK_1. Note how, without this third statement, BLCK_1 and BLCK_C would not be classified as clones (although they would share a common super class with "uses JAVA_java_lang_String" and "conditionalUses JAVA_ Boolean"). This demonstrates how we incrementally expand our knowledge base and reason about the source code. We parse source code file per file without compiling it or waiting for referenced classes to be found. In the above example, this means we could have parsed BLCK_1 and BLCK_C already, but have no idea about the content of data type "RMI" and its *toString()* method. Based on the open world assumption, information about it is unknown. We have implemented the reasoner outlined in detail in [3, 4] and extended its classification capabilities slightly in order to solve the problem of anonymous super classes. Alternatively, we can achieve the same results by adding additional facts to our knowledge base. For each class with at least 5 data types and/or method calls, we can add a permutation of each possible super class. This modification to our model would allow us to port our approach to other unmodified reasoners.

To summarize our approach (an overview is shown in Figure 3), we first crawl the Internet by visiting open source code repositories and parse each source code file separately, transforming its methods and control blocks into DL facts. This scales horizontally, as we let multiple machines crawl source code repositories in parallel and do not rely on compiling source code or a file order. We also do not need to parse all referenced code and can deal with incomplete information. We then use an EL+ reasoner that can scale horizontally using the MapReduce framework to classify our facts and infer subclass and equivalent class relationships in the ontology. Our ontology is designed so that these relationships determine whether a code clone between two source code fragments exists or not.

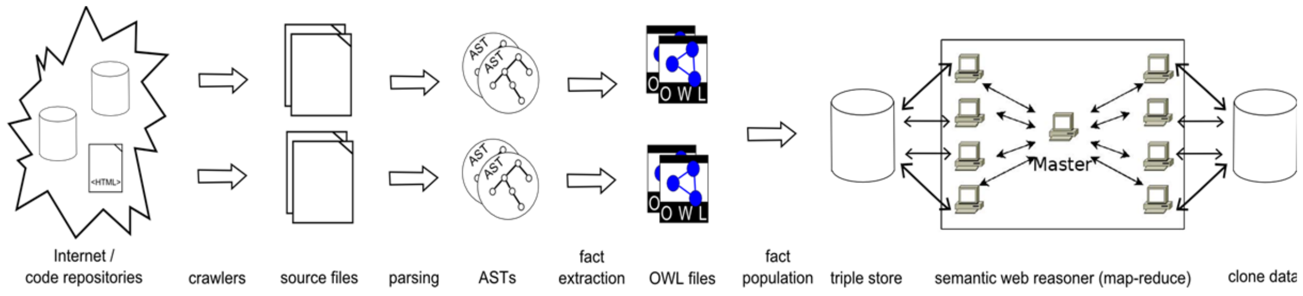


Figure 3. System overview

V. VALIDATION

In order to validate our approach, we have selected 4 popular clone detection tools, each using a different internal representation model and granularity, from the list presented in Section 2. Simian is a commercial String-based approach that is popular due to its integration in Eclipse. The open source CCFinder has shown remarkable recognition rates and is the best token-based approach available. JCD (Java Clone Detector) is a recent development from the University of Waterloo that matches Java pcode. DECKARD is a distributed implementation of an AST-based approach. Our approach does not detect clones spanning across multiple methods or clones smaller than a control statement. Therefore, we had to match the clones of other tools to the same detection level.

For our first evaluation, we randomly selected 97 files from the JDK 1.4 (swing package) and 620 Java files from the JDK 1.5 (javax and org packages). As mentioned in Section 4, our recognition performance depends on identifying the fully qualified type name of all used identifiers. As asterisk imports degrade our performance, we have selected only those files containing no asterisk imports. Although not infrequent within the JDK (around 380 in a sampled set of 1000), most modern Java programs do not have asterisk imports, as their imported data types are automatically managed by the IDE (e.g., Eclipse).

The parameters of JCD, Simian, CCFinder and DECKARD are based on the recommendations on their respective web sites and papers. While the amount of matches can vary based on these parameters, the association with cloned blocks remains relatively stable. All tools were tested on an Intel Core i5 2.53 GHz processor with 4 GB of RAM.

TABLE I. EVALUATION OF JDK 1.4 (SWING)

	dlclone	jcd	simian	ccfinder	deckard
Matches	1264	21	145	617	813
Blocks	1375	39	679	895	1263
Methods	603	0	337	473	663
Time	30s	52s	1s	18s	17s
Recall	0.79	0.02	0.40	0.53	0.74

TABLE II. EVALUATION OF JDK 1.5 (JAVAX, ORG)

	dlclone	jcd	simian	ccfinder	deckard
Matches	3919	2037	3381	2002	2034
Blocks	4066	1219	1569	3152	3572
Methods	1838	70	616	1628	1751
Time	1m10s	4m19s	2s	50s	48s
Recall	0.68	0.21	0.26	0.53	0.60

TABLE III. EVALUATION OF APACHE COMMONS

	dlclone	simian	ccfinder	deckard
Matches	16549	10250	7865	7980
Blocks	18078	6800	17092	16519
Methods	7729	2842	7374	8848
Time	6m31s	10s	4m10s	10m23s
Recall	0.61	0.23	0.57	0.56

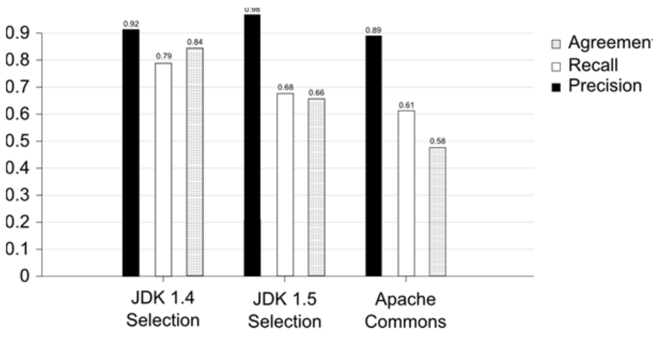


Figure 4. Precision and recall of DL-Clone

Tables II - IV show the number of detected matches and their respective mapping to blocks as well as a count of complete method clones. The number of matching blocks thereby is higher than the number of matches as one match might cover more than one block. CCFinder and DECKARD both detect large clones that often span across multiple methods. Once these clones are broken down to match complete methods, the number of detected clones between DL-Clone and CCFinder/DECKARD becomes similar. The algorithm used in JCD does not find clone blocks bigger than size N and does not try to expand a matching fragment until the maximum matching statements have been found. As a result, it has a poor performance when the detection criteria are complete clone methods. Simian only performs a string-based comparison of code fragments so a lower number of matching blocks and methods is not surprising. As JCD relies on a compilable source code, we have not run it over the Apache Commons that has complicated dependencies among the selected packages.

Precision and recall values have been determined by building an oracled set of clones and manually annotating the source code, similar to the clone tool evaluation of Bellon et al. [28]. The oracled set consists of a union of clone blocks detected by Simian, CCFinder, DECKARD, and JCD (where available). For the manual annotation of source code, we have built an online verification tool that automatically displays clone groups with their corresponding source code fragments and allows for a quick evaluation. Our evaluation rule is "Would you like to be informed of the following possible clones when changing the code of the original". We have annotated all blocks from the JDK selection and 15% of random blocks from the Apache Commons that only matched with our DL-Clone tool.

The results of our experiments indicate that our approach is capable of finding interesting clones that would be missed by other approaches. Figure 4 shows our obtained recall and precision values for the DL-Clone tool. We have also evaluated the inter-tool agreement between DL-Clone and all other tools (average) in terms of the code-blocks identified as clones. The number shows a high correlation that is obviously linked to the achievable recall, due to the oracled set we compare against.

A source of clones not identifiable by most other tools is the appearance of nested clones within a block such as similar code in different if-else statements. This has also been noted by [23], which mentions DECKARD as one of the only approaches handling this sort of clones.

VI. Discussion

In this paper, we have introduced a novel clone detection technique which is horizontally scalable and presents an efficient and novel feature vector consisting of data types, control blocks and method calls, to reliably identify code clones. A source of clones not identifiable by most other tools is the appearance of nested clones within a block such as similar code in different if-else statements. We are able to detect such clones as we match each if block with each other.

A threat to the validity of this approach are misclassified clones resulting from long initialization blocks where many data types are used but no methods are called on them. We might provide an additional handling for such clones in the future. Another, but less frequent source of misclassified clones, is coming from unmodeled control flows, such as break/continue statements, return statements, and exception throwing. The data types and method calls of such statements link into the parent block. Therefore, a block returning, for example, an Integer might be matched with a block assigning an Integer. This is also an area we would like to expand on in the future.

References

[1] Rahman, F. Bird, C. Devanbu, P. 2010. Clones: What is that smell? IEEE Working Conference on Mining Software Repositories (MSR), 2-3 May 2010, 72-81.

[2] Harder, J. Gode, N. 2010. Quo vadis, clone management? In Proceeding of IWSC '10 - 4th International Workshop on Software Clones. 2010, 85-86.

[3] Mutharaju, R. and Maier, F. A MapReduce Algorithm for EL+. 23rd International Workshop on Description Logics (DL2010), 2010, 464-474.

[4] Urbani, J., Kotoulas, S., Oren, E. & van Harmelen, F. Scalable Distributed Reasoning using MapReduce, In Proceedings of the 8th International Semantic Web Conference (ISWC '09), 2009.

[5] Simian Clone Detection Tool, http://www.redhillconsulting.com.au/, last visited October 2010

[6] Stephane Ducasse, Matthias Rieger, and Serge Demeyer. 1999. A Language Independent Approach for Detecting Duplicated Code. In Proceedings of the IEEE International Conference on Software Maintenance (ICSM '99)., USA, 109-116

[7] Prechelt, L. Malpohl, G. Philippsen, M. JPlag: Finding plagiarisms among a set of programs. Technical Report 2000-1, Fakultat fur Informatik, Universitat Karlsruhe, Germany, March 2000

[8] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. Transactions on Software Engineering, 8(7):654–670, 2002.

[9] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In Proceedings of the 14th IEEE International Conference on Software Maintenance (ICSM-98), 16–20 November, 1998, Bethesda, MD, USA, pages 368–377.

[10] Ian J. Davis and Michael W. Godfrey. 2010. Clone detection by exploiting assembler. In Proceedings of the 4th International Workshop on Software Clones (IWSC '10). ACM, New York, NY, USA, 77-78.

[11] K. Inoue, P. Garg, H. Iida, K. Matsumoto, and K. Torii. Mega software engineering. In Proc. of the 6th International PROFES (Product Focused Software Process Improvement) Conference, pages 399–413, Oulu, Finland, 2005.

[12] Livieri, S. Higo, Y. Matushita, M. Inoue, K. Very-large scale code clone analysis and visualization of open source program using distributed ccfinder: D-ccfinder, In Proc. of the 29th International Conference on Software Engineering, 2007

[13] Burd, E. Bailey, J. Evaluating clone detection tools for use during preventative maintenance. In Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'02), 2002, 36–43

[14] Koschke, R. Falke, R. Frenzel, P. Clone detection using abstract syntax suffix trees. In Proceedings of the 13th Working Conference on Reverse Engineering (WCRE), 253–262, 2006.

[15] Walenstein, A. Jyoti, N. Li, J., Yun Yang, Lakhotia, A.. Problems creating task-relevant clone detection reference data. In Proceedings of the 10th Working Conference on Reverse Engineering (WCRE-03), 285–294, 2003.

[16] Kapser, C. Toward an Understanding of Software Code Cloning as a Development Practice, Doctor Thesis, Uni. of Waterloo, 2009

[17] Dean, J. Ghemawat, S. MapReduce: simplified data processing on large clusters. Communications of the ACM, 107-113, 2004

[18] Apache Hadoop Project, http://hadoop.apache.org/, last visited January 2011

[19] Taylor, R. An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics, BMC Bioinformatics, Vol. 11, No. Suppl 12. 2010

[20] Berners-Lee, T. Hendler, J. Lassila, O. The Semantic Web. Scientific American, May, 2001.

[21] Baader, F., Lutz, C., Suntisrivaraporn, B. CEL—A Polynomial-time Reasoner for Life Science Ontologies. In Proceedings of the 3rd International Joint Conference on Au- tomated Reasoning (IJCAR'06), volume 4130 of Lecture Notes in Artificial Intelligence, 287–291. Springer-Verlag, 2006.

[22] Mu-Woong Lee, Jong-Won Roh, Seung-won Hwang, and Sunghun Kim. 2010. Instant code clone search. In Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering (FSE '10), 167-176.

[23] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In Proceedings of the 29th international conference on Software Engineering (ICSE '07), 96-105

[24] Raghavan Komondoor and Susan Horwitz. 2001. Using Slicing to Identify Duplication in Source Code. In Proceedings of the 8th International Symposium on Static Analysis (SAS '01), Patrick Cousot (Ed.). Springer-Verlag, London, UK, 40-56.

[25] Mark Gabel, Lingxiao Jiang, and Zhendong Su. 2008. Scalable detection of semantic clones. In Proc. of the 30th international conference on Software engineering (ICSE '08), 321-330

[26] Urbani J., Kotoulas, S., Maaseen J., van Harmelen, F. & Bal, H. (2010), OWL reasoning with WebPIE: calculating the closure of 100 billion triples, In Proceedings of the ESWC '10.

[27] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Sci. Comput. Program. 74, 7 (May 2009), 470-495.

[28] Stefan Bellon. Detection of software clones — tool comparison experiment, http://www.bauhaus-stuttgart.de/clones, last visited July 2010.

[29] Amazon Elastic Computing, http://aws.amazon.com/ec2/, last visited January 2011.