CDD: A Code Clone Detection DSL for Java

Anonymous Author(s)

Abstract

Duplication in source code is generally considered an undesirable pattern, because it unnecessarily increases system volume and is prone to cause bugs. Many studies propose definitions and techniques to detect such duplication issues. The clones detected by these definitions and techniques differ greatly. This is because there is a trade-off between completeness and number of false-positives. Additionally, clone detection may have different purposes, which may require different clone detection techniques.

We propose a DSL for clone detection, that allows for the specification of clone definitions. Using this DSL, code clone definitions can be easily expressed, whilst not having to worry about the actual clone detection algorithm and its optimization. Our DSL allows to exclude expressions, statements and declarations from clone matching. Our DSL supports line-based and AST-based clone detection approaches. The DSL can also take into account context information of expressions.

We use the DSL to express type 1, 2 and 3 clone definitions. We then use these definitions to validate the DSL over a corpus of open-source Java projects. We compare these results with established clone detection tools. We find that our DSL matches the output of the control tools, while allowing for a much wider spectrum of configurations.

Keywords clone detection, domain specific language, language engineering, meta-programming

1 Introduction

Code clones argue about the duplicate code present in the source code of software systems. An abundant number of clone detection tools and techniques have been proposed in the literature due to the many applications and benefits of clone detection [24]. Clones are most often the result of code reuse by copying and pasting existing code, among other reasons [20]. Clones are often seen as harmful [10, 17, 20], but can also have positive effects on system maintainability [1, 12, 20].

Clone detection is applied in various domains. This includes (automated) refactoring, education (plagiarism detection) [28], (legacy code) modernization [15] and maintainability analysis [6]. For these varying purposes, many clone detection tools have been proposed: a 2013 survey by Rattan et al. [19] surveyed more than 70 clone detection tools. Comparisons show large differences in recall and precisions of these tools [25].

To allow easy experimentation with code clone defitions and methods, we propose a DSL to configure clone detection.

finish this

2 Background

We describe terminology, definitions and techniques that are commonly used in code clone detection.

2.1 Code clone terminology

To argue about code clones, we use the following terminology:

Code fragment [24]: A continuous region of source code. Specified by the triple (l, s, e), including the source file l, the start line and column s, and the end line and column e.

Clone class [20]: A set of similar code fragments. The definition of similarity depends on the desired granularity of clones.

Clone instance [20]: A cloned code fragment that is part of a clone class.

2.2 Clone type definitions

Most modern clone detection tools detect clones by the following clone type definitions [25]:

Type 1: Code fragments that are syntactically identical, except for differences in white space, layout and comments.

Type 2: Code fragments that are syntactically identical, except for differences in identifier names, literal values, white space, layout and comments.

Type 3: Code fragments that are syntactically similar with differences at the statement level. The fragments may differ by the addition, removal or modification of statements.

2.3 Clone detection techniques

Different clone detection techniques have been proposed in literature [22]:

Textual: Code lines are textually compared. Advantages are that textual approaches are often simpler, easier to implement and are independent of language. Disadvantages are that it is harder to exclude certain constructs from matching.

Lexical (Token): Lexical approaches first tokenize the source code, so certain groups of tokens can be excluded from matching [25]. The disadvantage is that tokens must be specified for each language.

Syntactic (Tree): AST-based approaches first parse the source code into an AST. This allows for subtree based comparison and mapping context of clones [5].

Syntactic (Metric): In metric-based clone detection techniques, a number of metrics are computed for each fragment

PL'18, January 01–03, 2018, New York, NY, USA
2018.

of code to find similar fragments by comparing metric vectors instead of comparing code or ASTs directly.
Semantic (Graph): A graph-based clone detection technique uses a graph to represent the data and control flow of a program.
Semantic (Hybrid): Hybrid approaches use a combination of before-mentioned techniques to speed up detection of clones and find clones with a variety of characteristics.

3 Language Design

We propose the Clone Detection DSL (CDD)[1], a DSL that allows to build a clone detection script to analyze Java systems.

3.1 Clone Size

An important feature of many clone detection tools, to be able to find clones that have a high relevance, is to be able to configure what characteristics clones should have for them to be considered. Some clone detection tools use a threshold that specifies the minimum number of lines [3, 4, 9, 27]. Other clone detection tools specify the minimum number of tokens [8, 11, 26]. Some AST-based clone detectors use the minimum number of statements [7].

Based on such thresholds, clones that are too small are filtered. How these thresholds are configured has a big impact on the results and their relevance [20]. Our DSL allows for any combination of metric values to configure the minimum size of a clone. For this, we introduce the **Size** keyword. This is an example of how the **Size** keyword can be configured:

```
Size (4 lines & 50 tokens) | (6 lines & 40 tokens)
```

In this example, we seek clones that either have many tokens but fewer lines, or many lines but fewer tokens. We allow the following keywords to be used in the condition of the **Size** threshold:

- *lines*: Number of lines in clone instance
- *tokens*: Number of tokens in clone instance
- *statements*: Number of statements in clone instance
- *declarations*: Number of declarations in clone instance
- *complexity*: The cyclomatic complexity [14] found in a clone instance.

Normally, these keywords denote the value of the metric of each clone instance. However, when prepended by *"class_"*, they denote the value of the metric for all clone instances in a clone class combined. This means that *lines* specifies the minimum number of lines in each clone instance, and *class_lines* specifies the minimum number of lines in all clone instances of a clone class combined.

[1]Link to the GitHub repository is omitted to adhere to the double-blind review rules. Link to full source code of the DSL would be included in the camera-ready version.

This allows for easy experiments of combinations of metrics to determine whether a clone should be considered.

3.2 Match Granularity

The **Match** keyword describes on what granularity clones will be matched. CDD can find clones that span physical lines of code, logical lines of code and in subtrees of the AST of a program.

3.2.1 Physical Lines of Code (LoC)

Physical lines of code are the lines of code as displayed in the editor of a program, excluding whitespace and comments. Code lines are compared by comparing tokens on one line with another. The main advantage of LOC is that, if statements are relatively equally distributed over lines, each line

The main disadvantage of LOC is that its granularity depends on the coding style used by the programmer of the source code. A programmer that adheres to a maximum line length might break up statements more often than those who do not [16]. Because of that, using LOC might give different results based on the design of the code.

Another disadvantage is that changing the style of the code between fragments might result in duplicate code not being considered a clone. For instance, if in one fragment a statement is on a single line and in another the same statement is distributed over two lines, it will not be detected as a clone.

3.2.2 Logical Lines of Code (LLoC)

These are logical lines of code, where each statement and some declarations are counted as a logical line of code. In the study "A SLOC Counting Standard" [16], Nguyen et al. discuss which declarations and statements should be considered as a logical line of code.

The main advantage of LLoC over LoC is that it is less dependent on coding style. If two statements are equal, independent of how they are distributed over physical lines, they will be considered clones. Some also argue that LLoC gives a more accurate indication of volume [16].

3.2.3 Subtree

A third approach to compare blocks of code is to compare subtrees of the AST of the source code. The advantage of this is that the result may be more useful for refactoring, as clones that are found are often easier to refactor. The disadvantage is that some clones may not be found. An example of this is shown in Fig. 1. In this figure, both the for-loop and the if-statement do not match completely between the two fragments. Because of that, when comparing subtrees, both would not indicate a match.

3.3 Search Root

Some clone detection tools only search for clones inside of method bodies [18]. To allow for such targeted analyses, we

```java
for(int i = 0; i<apples.length; i++){
  Apple thisApple = apple[i];
  if(thisApple.isEdible()){
    thisApple.eat();
    Logger.log("Apple eaten");
  }
}
```

```java
for(int i = 0; i<apples.length; i++){
  Apple thisApple = apple[i];
  if(thisApple.isEdible()){
    thisApple.eat();
  }
}
```

Figure 1. Example of clone that may not be detected when comparing subtrees.

introduce the **SearchRoot** keyword into CDD. This keyword determines where the script will start looking for clones. For instance, if we want to limit our analysis to methods and constructors, we can specify the following:

SearchRoot MethodDeclaration, ConstructorDeclaration

The *SearchRoot* keyword takes as an argument a comma separated list of nodes. The names that are being used here, *MethodDeclaration* and *ConstructorDeclaration*, are the names of the method and constructor declaration nodes. The names of these nodes can be the name of any node specified in the JavaParser library [23]. This is because CDD uses JavaParser under the hood to parse an AST and perform clone detection.

3.4 Exclusions

Type 2 clones allow variance in identifiers and literals. To allow to exclude nodes in CDD, there is the **DoNotCompare** keyword. This keyword allows to specify a list of nodes that should be excluded from matching.

DoNotCompare SimpleName, LiteralExpr

This keyword is different from the *Exclude* keyword in that the *Exclude* keyword specifies nodes that will not be included in clones, whereas *DoNotCompare* nodes will be included in clones but do not have to match for them to be considered a clone.

3.5 Compare Rules

Finally, CDD allows to specify any number of compare rules. explain The syntax for these rules is simple but powerful.

4 Example scripts

To experiment with the CDD language, we convert current clone type definitions to CDD code. The only required keywords in the CDD language are **Project**, **Size** and **Match**. For the minimum size of type 1 clones many studies use *6 lines* [2, 6, 21]. We choose to compare **LoC**, but the other options would also be valid (the definition of type 1 clones is ambiguous in this aspect). We exclude package and import declarations, as clones found in those declarations are often not useful [13]. The CDD code for this is as follows:

---

type1.cdd

Project "/path/to/java/project"
Size 6 **lines**
Match LoC
Exclude PackageDeclaration, ImportDeclaration

For type 2, we allow variance in identifiers and literals. For this, we can use the **DoNotCompare** keyword: all subnodes of these nodes are not compared between lines when comparing lines to determine whether they are similar. The CDD code for type 2 is as follows:

---

type2.cdd

Project "/path/to/java/project"
Size 6 **lines**
Match SLOC
Exclude PackageDeclaration, ImportDeclaration
DoNotCompare SimpleName, LiteralExpr

For type 3, we allow statements to differ. To implement this, we specify a percentage of variance that we allow statements to differ (in this case 10%). This results in the following code:

---

type3.cdd

Project "/path/to/java/project"
Size 6 **nodes**
Match Subtree
Exclude PackageDeclaration, ImportDeclaration
DoNotCompare SimpleName, LiteralExpr
Compare ∗ **Subnodes** Statement 10% /∗ We allow 10% variance in Statements ∗ /

5 Improving Existing Clone Types with CDD

Bla bla refactoring oriented clone types bla bla HOI

```
type1r.cdd
  Project "/Users/sbaars/Documents/Kim/jhotdraw"
  Size (6 nodes & 20 tokens) | (5 nodes & 25 tokens) | (4 nodes & 30 tokens) | (3 nodes & 35 tokens) | (2 nodes & 40 tokens) | (1 nodes & 50 tokens)
  Match LLOC
  Exclude PackageDeclaration, ImportDeclaration
  Compare ReferenceType, LiteralExpr, MethodCallExpr
FQI

type2r.cdd
  Project "/Users/sbaars/Documents/Kim/jhotdraw"
  Size (6 nodes & 20 tokens) | (5 nodes & 25 tokens) | (4 nodes & 30 tokens) | (3 nodes & 35 tokens) | (2 nodes & 40 tokens) | (1 nodes & 50 tokens)
  Match LLOC
  Exclude PackageDeclaration, ImportDeclaration
  Compare ReferenceType /* Reference Types */, LiteralExpr /* Literals */, MethodCallExpr /* Method Calls */
FQI
  Compare LiteralExpr /* Literals */ StringMatch 10%
  Compare NameExpr /* Variables */, MethodCallExpr /* Method Calls */ CompleteMatch 10%

type3r.cdd
  Project "/Users/sbaars/Documents/Kim/jhotdraw"
  Size (6 nodes & 20 tokens) | (5 nodes & 25 tokens) | (4 nodes & 30 tokens) | (3 nodes & 35 tokens) | (2 nodes & 40 tokens) | (1 nodes & 50 tokens)
  Match LLOC
  Exclude PackageDeclaration, ImportDeclaration
  Compare ReferenceType /* Reference Types */, LiteralExpr /* Literals */, MethodCallExpr /* Method Calls */
FQI
  Compare LiteralExpr /* Literals */ StringMatch 10%
  Compare NameExpr /* Variables */, MethodCallExpr /* Method Calls */ CompleteMatch 10%
  Compare * Subnodes Statement 10% /* We allow 10% variance in Statements */
```

6 Discussion

7 Conclusion

7.1 Threats to validity

References

[1] Lerina Aversano, Luigi Cerulo, and Massimiliano Di Penta. 2007. How clones are maintained: An empirical study. In *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*. IEEE, 81–90.

[2] Magiel Bruntink, Arie Van Deursen, Tom Tourwe, and Remco van Engelen. 2004. An evaluation of clone detection techniques for crosscutting concerns. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* IEEE, 200–209.

[3] James R Cordy and Chanchal K Roy. 2011. The NiCad clone detector. In *2011 IEEE 19th International Conference on Program Comprehension.* IEEE, 219–220.

[4] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. 1999. A language independent approach for detecting duplicated code. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*. IEEE, 109–118.

[5] Francesca Arcelli Fontana and Marco Zanoni. 2015. A duplicated code refactoring advisor. In *International Conference on Agile Software Development*. Springer, 3–14.

[6] Ilja Heitlager, Tobias Kuipers, and Joost Visser. 2007. A practical model for measuring maintainability. In *6th international conference on the quality of information and communications technology (QUATIC 2007)*. IEEE, 30–39.

[7] Yoshiki Higo and Shinji Kusumoto. 2009. Enhancing quality of code clone detection with program dependency graph. In *2009 16th Working Conference on Reverse Engineering*. IEEE, 315–316.

[8] 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*. IEEE Computer Society, 96–105.

[9] Elmar Juergens, Florian Deissenboeck, and Benjamin Hummel. 2009. CloneDetective-A workbench for clone detection research. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 603–606.

[10] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. 2009. Do code clones matter?. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 485–495.

[11] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670.

[12] Cory Kapser and Michael W Godfrey. 2006. "Cloning considered harmful" considered harmful. In *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*. Citeseer, 19–28.

[13] Rainer Koschke. 2012. Large-scale inter-system clone detection using suffix trees. In *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, 309–318.

[14] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320.

[15] Fanqi Meng, Zhaoyang Qu, and Xiaoli Guo. 2013. Refactoring model of legacy software in smart grid based on cloned codes detection. *International Journal of Computer Science Issues (IJCSI)* 10, 1 (2013), 296.

[16] Vu Nguyen, Sophia Deeds-Rubin, Thomas Tan, and Barry Boehm. 2007. A SLOC counting standard. In *Cocomo ii forum*, Vol. 2007. Citeseer, 1–16.

[17] J. Ostberg and S. Wagner. 2014. On Automatically Collectable Metrics for Software Maintainability Evaluation. In *2014 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement*. 32–37. https://doi.org/10.1109/IWSM.Mensura.2014.19

[18] Chaiyong Ragkhitwetsagul and Jens Krinke. 2019. Siamese: scalable and incremental code clone search via multiple code representations. *Empirical Software Engineering* (2019), 1–49.

[19] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. 2013. Software clone detection: A systematic review. *Information and Software Technology* 55, 7 (2013), 1165–1199.

[20] Chanchal Kumar Roy and James R Cordy. 2007. A survey on software clone detection research. *Queen's School of Computing TR* 541, 115 (2007), 64–68.

[21] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. SourcererCC: scaling code clone detection to big-code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 1157–1168.

[22] Abdullah Sheneamer and Jugal Kalita. 2016. A survey of software clone detection techniques. *International Journal of Computer Applications* 137, 10 (2016), 1–21.

[23] Nicholas Smith, Danny van Bruggen, and Federico Tomassetti. 2018. JavaParser.

[24] Jeffrey Svajlenko and Chanchal Roy. 2019. The Mutation and Injection Framework: Evaluating Clone Detection Tools with Mutation Analysis. *IEEE Transactions on Software Engineering* (2019).

[25] Jeffrey Svajlenko and Chanchal K Roy. 2014. Evaluating modern clone detection tools. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 321–330.

[26] Warren Toomey. 2012. Ctcompare: Code clone detection using hashed token sequences. In *2012 6th International Workshop on Software Clones (IWSC)*. IEEE, 92–93.

[27] Md Sharif Uddin, Chanchal K Roy, and Kevin A Schneider. 2013. Simcad: An extensible and faster clone detection tool for large scale software systems. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 236–238.

[28] Vera Wahler, Dietmar Seipel, J Wolff, and Gregor Fischer. 2004. Clone detection in source code by frequent itemset techniques. In *Source Code Analysis and Manipulation, Fourth IEEE International Workshop on*. IEEE, 128–135.