A Systematic Review of Semantic Clone Detection Techniques in Software Systems

Ajad Kumar, Rashmi Yadav and Kuldeep Kumar
Department of Computer Science and Engineering, Dr B R Ambedkar National Institute of Technology, Jalandhar, Punjab, India
Email: ajadhtil747@gmail.com, rashmi.yadav95@gmail.com, kumark@nitj.ac.in

Abstract. Code clones are repeated program structures of significant similarities occurring in a software system. In various studies, it has been found that there is at least 20–30 percent of the code which is duplicated in a software system. These code clones induce software maintenance related difficulties that might lead to bug fixing, bug propagation, etc. that seed accretion of maintenance time, effort, and cost. To achieve better efficiency, it is required to detect all types of clones, i.e., syntactic and semantic clones that exist in a software system. Detection of semantic clones is considered to be a more difficult task in comparison to the syntactic clones because it requires semantic information of the given program rather than a similar structure. To detect semantic clones, techniques based on program dependency graphs, abstract syntax trees, machine learning, deep learning, support vector machines, etc. are very popular. This paper presents a systematic and extensive analysis of clone detection tools and techniques with a special focus laid on semantic clone detection. It also highlights the pros and cons of existing clone detection techniques along with the results viz. precision, recall, etc.

Keywords. code clone, semantic clone, clone detection, syntactic clone, systematic literature review.

1. Introduction and Motivation
Cloning is a trending practice in software industries as well as in the research realm. The existing code is widely reused from available resources that promote the code cloning phenomenon. It leads to a challenging problem of software maintenance. It requires a lot of time and effort which increases the developing cost of any tool, software, etc. There are mainly two types of clones: syntactic clone and semantic clone. Syntactic clone [1] includes Type-1, Type-2, Type-3 whereas semantic clone contains Type-4 clone. Type-1 clones can be defined as code fragments that are identical to each other with the alteration of white spaces, comments, layout, etc. Type-2 clones are described as code segments that are similar to each other, but there are slight adjustments in their name like renaming of identifiers, variables, etc. Hence, Type-2 clones are also known as renamed clones. Type-3 clones are near-miss clones that are similar code fragments but with some extra addition and deletion of statements. Semantic clones (or Type-4 clones) are those clones that perform a similar task but their syntactic structure may be different. A typical example of a semantic clone that discusses the swapping of two numbers using different logics is shown in Table 1.

In the past, several techniques [1, 2] are used to detect clones in the source code such as text-based, token-based, abstract syntax tree (AST)-based, program dependency graph (PDG)-based, etc. Recently, deep learning [3] and machine learning [4, 5] based techniques also play an important role in the detection of clones in the source code. In this paper, we furnish a brief and efficient literature review to help future researchers to get familiar with the techniques that are used to detect semantic
code clones. The primary intent of this study is to present an appropriate systematic and comparative analysis of semantic clone detection techniques with their respective advantages and disadvantages.

### Table 1. An example of a semantic clone.

| main () | main () |
|---------|---------|
| {int I, J; int temp; temp=I; I=J; J=temp; } | {int P, Q; P=P+Q; Q=P-Q; P=P-Q; } |

#### 1.1. Research Questions

**RQ1:** What are semantic clones and how they are different from the syntactic clone? Is it possible to detect semantic clones in software systems?

**RQ2:** What are the various ways available in the literature to detect semantic clones in software systems?

**RQ3:** What are the features that a semantic clone detection tool should have?

**RQ4:** What are the various semantic clone detection tools developed/available for use in the literature?

**RQ5:** What are the limitations of available semantic clone detection tools?

#### 1.2. Sources of information

We cater to the following standard sources to get the relevant information to perform a systematic literature review.

(i) Google Scholar (https://scholar.google.com/)

(ii) IEEE Xplore (https://ieeexplore.ieee.org/)

(iii) ScienceDirect (https://www.sciencedirect.com/)

(iv) Springer (https://link.springer.com/)

Keywords: We used the keywords “semantic clone detection”, “detection of semantic clones”, “semantic clone detection using machine learning” to find out potential research papers that are related to semantic clone detection.

#### 1.3. Paper inclusion-exclusion criteria

We used various criteria based on which a research paper is either included or excluded in the study.

(i) Inclusion criteria.

- Papers that present an approach to detect semantic clones in the source code.
- Papers that use machine learning, deep learning, or conventional techniques (such as those based on program dependency graph, abstract syntax tree, behaviour effect, program slicing) to detect semantic clones.
- Papers that are published in 2008 onwards.

(ii) Exclusion criteria.

- Papers that are published in other than the English language.
- Papers by the same authors targeting similar clone detection approaches.
- Papers that are published before 2008.
- Papers were published in the area of syntactic clone detection.

After performing the rejection based on year, abstract, title, and internal content, the final numbers of remaining studies is around 21. Figure 1 represents the year-wise distribution of papers selected for the study.
The rest of the paper is organized as follows: Section 2 provides a brief overview of the clone detection process. In Section 3, various clone detection techniques are discussed in detail. Section 4 presents a literature review and a tabular comparison of various semantic code clone detection techniques. Section 5 presents a comprehensive discussion and limitations of the literature review. Conclusion and future work are presented in Section 6.

Figure 1. Year-wise distribution of papers selected for the study

2. Clone Detection Process
Detection of clones undergoes through various steps [1, 2, 6] as pre-processing, transformation, match-detection, formatting, and post-processing (Figure 2).

2.1. Pre-processing
This phase takes source code as input for the detection process. This phase extracts the required parts of the code needed for clone detection and eliminates all those uninteresting parts from the code segment such as comments, header files, etc. It also determines the source unit where the comparison is performed for clone detection in the source code. The pre-processed code act as output for this phase.

2.2. Transformation
During this phase, the pre-processed code is transformed into various forms like token form, abstract syntax tree, program dependency graph, etc. that makes the comparison simple for clone detection in source code.

2.3. Match Detection
This phase takes transformed code as input. Matching is performed using some techniques such as similarity index, string matching algorithm, subgraphs matching. These techniques are used to find out clone and non-clone pairs that act as output.

2.4. Formatting
After matching is performed within the original transformed code, this phase identifies the location of clone pairs and clone classes in the source code. after matching is performed within the original transformed code.

2.5. Post-processing
Post-processing is also known as the filtering process because, in this phase, we remove all false clone pairs and the obtained clone pairs are of interest for further clone management.
Figure 2. Overview of clone detection process in source code

3. Clone Detection Techniques
A variety of techniques are used to detect clone, they are primarily text-based, token-based, metric-based, abstract syntax tree-based, program dependency graph-based, etc. These are the techniques that are used to detect both syntactic and semantic clone pairs in source code. Now a day’s machine learning, deep learning-based techniques are also proposed to detect semantic clones.

3.1 Text-based Clone Detection Techniques
In the text-based technique, after performing all pre-processing activities, the transformed code is matched with the source code line by line or text by text. The matched sequence of characters is known as the clone pair.

3.2 Token-based Clone Detection Techniques
In this technique of clone detection, a sequence of tokens of the source code is generated using a lexical parser. Many lexical tools such as ASTVisitor are available to generate streams of tokens of the source code. Then clones are detected by calculating similarity indexes among various token sequences.

3.3 Metric-based Clone Detection Techniques
In the metric-based clone detection technique, various metrics are calculated from the pre-processed code and these metrics are compared with each other. Based on the similarity in these metrics, the corresponding code fragments are identified as a clone pair or a non-clone pair. Various metrics like fan-out, MaCabe cyclomatic complexity of source code, the ratio of the number of the input variables to the output variables are considered to achieve the goal. Metric-based techniques have high scalability but low portability.
3.4 Abstract syntax tree-based Clone Detection
In this technique of clone detection, at first, an abstract syntax tree of the code is created by using a variety of tools available, and later on, a similar subtree is tried to be discovered out by using appropriate matching algorithms. If a match is found then, the corresponding code fragments are returned as a clone pair otherwise as a non-clone pair.

3.5 Program Dependency Graph-based Clone Detection
Program dependency graph-based clone detection is one of the best approaches to detect clones, especially semantic clones in source code because all of the other techniques are not able to detect semantic clone efficiently. The program dependency graph of any pre-processed source code shows the functional and data-flow of the source code that helps in the detection of semantic clones in the source code.

4. Literature Review
A review has been performed to analyse the different existing technologies that are used for clone detection in the segments of source code. It helps to identify many other challenges related to the identification of clones in the source code. This review is prepared using various research articles along with their number e.g. journal-3, conference-13, workshop-2, and symposium-3, which is published in a recent decade as shown in figure 3.

Thaller et al. [7] proposed a semantic clone detection technique using a probabilistic software model (PSM). The proposed approach confers better results in comparison to other tools like Oreo, precision, and recall as mentioned in Table 2. Using PSM, the authors find out the equivalence behaviour which is known as a similarity evaluation. This similarity evaluation is then used to detect semantic clones in the software. Chug et al. [6] presented a code clone detection technique using an abstract syntax tree Since abstract syntax tree is better suitable for the detection of Type-3 clones, they are not able to detect Type-4 clones with better accuracy. First, they created an abstract syntax tree of the code, then subtree matches are returned as clone pairs.

Ghosh and Kuttal [8] proposed an approach for detecting semantic clones using source code comments. This approach uses the idea of comments with LDA (Latent Dirichlet Allocation) and find a good result with precision 84% and recall 94%. Almori and Stephan [9] gave an introductory approach to detect semantic clones by using the slicing concept. The authors have used SrcSlice, a slicing tool that is used to know the slice profile of each variable, to detect semantic clones. The clone pairs are detected based on the similarity of the slice profiles of each variable.

Sheneamer and Kalita [5] presented a semantic clone detection using machine learning classification algorithms. The authors extracted the combined features from the abstract syntax tree and program dependency graph representation of the source code. By using these features, two different feature vectors are created as shown by equation 1 and equation 2.
These feature vectors are fed to a classifier one by one. Xgboost, random forest, and rotation forest classification algorithms are used to classify them as clone pairs or non-clone pairs. This technique can detect both syntactic and semantic clones, the precision value and recall value are mentioned in Table 2.

Li et al. [3] implemented a token-based clone detection tool, CCleaner, using deep learning. CCleaner first extracts a set of features that can be used to identify clone pairs easily from the source code by generating tokens using a lexical parser. These tokens are classified into eight categories to create eight disparate features, which acts as an input for the deep neural network and produces output as a clone and non-clone pair. This tool can detect syntactic clones with good precision and recall but is not able to detect semantic clones with good accuracy.

Jadon [4] proposed a technique for detecting clones. It consists of two phases. The first phase includes the generation of feature sets using a parser. The parser takes the C source file as input and generates features such as loops, return statements, function calls, recursion function, statement declarations, variable, identifiers, etc. Phase 2 classifies the clone and nonclone pairs using feature sets and support vector-based classification algorithm. The accuracy of the proposed technique increases with the increase in the number of instances in the training set. Yang et al. [10] proposed a machine learning-based classification model for code clones. Initially, a user detects clones using an existing clone detection tool. Some of these detected clones are marked as true or false clones by the user based on his/her experience. Using those marked clones, FICA ranks to other unmarked clones using machine learning. The tool learns based on these results and gives better results next time. This tool has an accuracy of approximately 70-90% and works for C language.

Priyambadha and Rochimah [11] discussed an approach for detecting semantic clones using input, output, and effect in the various void and non-parameterized methods by using the concept of the program dependency graph. The program dependency graph of any fragment represents the data dependency relationship that exists in the program. It helps in knowing the input, output, and effect behaviour of any fragment. The authors tested their approach on the jDraw dataset and found good accuracy of about 89%. Kurtz and Shihab [12] presented a tool named as CCCD (Concolic Code Clone Detection) to detect semantic clones. CCCD uses the concept of concolic analysis to identify code clone. This tool combines the method and concolic information to generate the final output as a clone or non-clone pair.

Kamiya [13] proposed a tool named Agec that detects semantic clones from the Java bytecode. Their approach is based on the first generation of execution sequences, extracting n-grams from the execution sequences, identifying similar n-grams from the different locations, and then return similar n-grams as a clone pair. The generation of n-grams takes place by static analysis of the Java bytecode. This tool works only for the interpretation of Java bytecode and not scalable for big projects. Yu et al. [14] proposed a tree-based convolution to detect semantic clones by using an abstract syntax tree of the subject program. The obtained results are comparable to other traditional approaches.

Elva and Leavens [15] presented a semantic clone detection tool, JSCTracker, for Java source code. This tool is based on the concept of input-output-effect behaviour of code fragments. If two code fragments have identical input-output-effect behaviour, then those code fragments are returned as semantic a clone pair. This paper gives better results with small test cases but is not scalable and their precision value is about 27-65%. Yoshioka et al. [16] presented a tool Takana, for scalable detection of semantic clones based on two-stage clustering. In the first stage of clustering, the authors have created a cluster based on the characteristics of various code fragments, and in the later stage, the result of the first stage is finely classified to get more precise clusters that act as clone pair set. This tool takes less execution time in comparison to other tools like Scorpio, Deckard.

Schugerl et al. [17] proposed an approach to detect semantic clones that use data types, control blocks, and method calls as features vectors to detect the presence of clone in code fragments. This
approach has an advantage over the other tools because it detects nested clones also if any found. Kim et al. [18] implemented a semantic clone detector MeCC that is based on the concept of comparison of abstract memory states of programs that are already computed by a semantic-based static analyser. The tool has been compared with the other tools and finds better results in comparison to other tools.

The tool named EqMiner [19], proposed by Jiang et al., is based on input-output behaviour. If two code fragments have the same input-output behaviour, then those code fragments are returned as a clone pair. This tool is scalable so it suitable for large projects also. However, it works with only C language. Gabel et al. [20] proposed an algorithm to detect semantic clones that reduces a difficult-graph similarity problem into a simple-tree similarity problem. This paper uses the concept of a program dependency graph that gives semantic information of the program and also shows data flow in the program. Dandois and Vanhoof [21] proposed a semantic code clone detection technique in logic programs.

5. Analysis and Limitations
The selection of an appropriate tool and technique plays a significant role in determining the types of clones we are interested in. This review has the foremost motivation to present a systematic literature review of semantic clones. Table 2 presents a summary of selected semantic clone detection techniques. From this survey, we would be able to analyse that to detect semantic clones, various techniques based on program dependency graph, tree-based, metric-based, program slicing, input-output-effect behaviour are presented in the literature.

Table 2 Comparative analysis of selected semantic clone detection techniques.

| Author          | Technique                        | Tool          | Advantage                                                                 | Disadvantage                                                                 | Recall  | Precision |
|-----------------|----------------------------------|---------------|---------------------------------------------------------------------------|------------------------------------------------------------------------------|---------|------------|
| Thaller et al. [7], 2020 | Probabilistic software mode | SCD-PDM       | It can detect semantic clone (T4) with good accuracy.                     | 1: It is not able to detect syntactic clones with good accuracy. 2: It cannot be applied to running source code. | T4-79.7% | 99.10%     |
| Ghosh and Kuttal [8], 2018 | Source code comments with Latent Dirichlet Allocation (LDA) | -             | The approach performed better than PDG based approaches.                  | -                                                                           | 94%     | 84%        |
| Linqing et al. [3], 2017 | Token-based using deep neural network | CCLearner     | 1: It can detect syntactic clone with good accuracy. 2: It is a fast tool in comparison to other token-based tools. | 1: It can detect semantic clones with very little accuracy. 2: This tool is not able to extract the semantic features of the program. | T1-100% | 93%        |
| Elva and Leavens [15], 2012 | Based on the concept of input-output-effect behavior of program | JSCTracker    | 1: It can detect all clones with average accuracy.                         | 1: It is not scalable for a big test case. 2: It interprets only Java byte   | NA      | 27-65%     |
| Name                  | Description                                                                 | Tool                                                                 | Features                                                                 | Accuracy  |
|-----------------------|-----------------------------------------------------------------------------|----------------------------------------------------------------------|--------------------------------------------------------------------------|-----------|
| Kim et al., [18], 2011 | Semantic-based static analyzer.                                             | MeCC                                                                | 1: This tool can detect Type-3 and Type-4 clone with better accuracy. 2: It is used to detect bugs in a fragment. | NA        |
|                       |                                                                             |                                                                     | 1: It is not able to detect finer granular clones present in the program. 2: It produces more false positives. |           |
| Sheneamer and Kalita [5], 2016 | Using combined features from an abstract syntax tree, a metric-based program dependency graph | Using equation 1                                                    | 1: It can detect Type-3 and type-4 clones with good accuracy. 2: It considers most of the semantic features of a program. 3: This tool is scalable. | MT3-82% VST3-94% ST3-85% WT3/4-82% |
|                       |                                                                             |                                                                     | 1: It is not able to detect Type1 and Type2 clone. 2: It takes time to execute because of PDG. | MT3-89% VST3-96% ST3-90% WT3/4-94% |
| Sheneamer and Kalita [5], 2016 | Using combined features from an abstract syntax tree, a metric-based program dependency graph | Using equation-2                                                   | 1: It can detect Type-3 and type-4 clones with good accuracy. 2: It considers most of the semantic features of a program. 3: This tool is scalable. | MT3-91% VST3-96% ST3-90% WT3/4-94% |
| Kamiya et al., [13], 2013 | Using the concept of n-grams.                                               | Agec                                                               | It is scalable for a big repository.                                    | NA        |
|                       |                                                                             |                                                                     | This tool works only for Java byte code.                                |           |
| Jiyang et al., [19], 2019 | Based on input-output behavior                                              | EqMiner                                                           | 1: This tool works only for C language 2: It requires a lot of space to interpret input-output behavior. | NA        |
| Priyambadha et al. [11], 2014 | Program dependency graph based approach                                     | -                                                                  | Accuracy is high. The approach has not been validated on medium or large size systems. | 89%       |
5.1 Answering of Research Questions

RQ1: What are semantic clones and how they are different from the syntactic clone? Is it possible to detect semantic clones in software systems?

Answer: Semantic clones are those software clones which have the same semantic meaning but can have different syntactic structure while syntactic clones have a similar structure. Yes, it is possible to detect semantic clones in software systems. Various techniques as discussed in the paper are proposed in the past for detecting semantic clones.

RQ2: What are the various ways available in the literature to detect semantic clones in software systems?

Answer: There are various techniques available in the literature to detect semantic clones in software systems like using abstract syntax tree, program dependency graph, machine learning, deep learning, input-output-effect behaviour, n-grams, software probabilistic model, program slicing, etc.

RQ3: What are the features that a semantic clone detection tool should have?

Answer: A good semantic clone detection tool should have high precision, recall, f-score, scalable, and portable.

RQ4: What are the various semantic clone detection tools developed/available for use in the literature?

Answer: Some of the popular semantic clone detection tools available in the literature are CCCD (Concolic Code Clone Detection), Takana, JSCTracker, EqMiner, Agec, MeCC (Memory Comparison clone), Fica, CCleaner, SCD-PDM, etc.

RQ5: What are the limitations of available semantic clone detection tools?

Answer: Some semantic detection tools have a low value of precision, recall, and f-score and take more time to execute. Further, it is emphasized that those detection tools are competent to detect semantic clones for a particular language, which means not compatible with all the languages and also has problems of scalability, portability. Most of the tools are working for clone detection in C, C++, and Java, etc., but not working well for python, .Net, C#, etc.

From the literature review, we are also able to analyse that code cloning has a lot of issues such as bug fixing, bug propagation that require a lot of effort during software maintenance. So, to improve efficiency and code quality, we would require detecting all duplicated code segments present in the source code. Further, from some studies, it can be winded up the following conclusions:

- Text-based approaches are capable to detect exact clone.
- Token-based approaches are competent to detect all types of clones but they are not proficient to detect semantic clones with good value of accuracy.
- Abstract syntax tree-based tools can detect semantic clones as well as syntactic clones but these tools are not scalable.
- Tree-based and metric-based techniques can detect syntactic clones and semantic clones but they have low accuracy in comparison to the program dependency graph.
- Program dependency graph-based approaches can detect semantic clones with good accuracy. But the major limitation of this approach is that it takes a lot of time because it requires the generation of a program dependency graph that is a time-consuming process. Sometimes, it is observed that program dependency graph-based tools are highly scalable and portable rather than other approaches like text-based, token-based, etc.
- Some tools use combined features (extracting from abstract syntax tree and program dependency graph) to detect semantic clones with good accuracy, precision, and recall. We found that some of the tools are using machine learning, deep learning-based approaches to
detect semantic clones and those approaches can detect semantic clones with good value of the result.

6. Conclusion and Future Scope

This research study broadly examines how code clones are detected and which tools and techniques are available for the same purpose. We surveyed 23 research articles, published in top-tiered conferences, journals, workshops, and symposiums during 2008-2020, in the domain of semantic code clone detection. Various tools are available that are implemented to detect clone and non-clone pairs in software systems that have many advantages as cloning increases needless maintenance cost, numbers of lines of code, bug propagation, and sometimes it shrinks the further development in the existing source code.

It is observed that a lot of tools are available that can detect syntactic clones very easily but the detection of semantic clones is a complex process. Text-based, token-based, and metric-based tools are not able to detect semantic clones with great accuracy. Finding appropriate cost-effective, efficient, and competent tools is a tedious task. Relatively, machine learning-based techniques to detect clones have shown promising results by outperforming the other traditional techniques.

The characteristics of a good tool are high precision, high recall, portability, robust and it should be scalable. For this reason, only a few studies are dealing with the detection of semantic clones along with certain drawbacks. Hence, a major future research direction is to develop a scalable tool and technique using hybridization to effectively detect semantic clones with great ease. Hence, the findings of this paper would be extremely advantageous. Besides, Java and C/C++ languages are considered for code clone detection in most of the literature. In this regard, other programming languages like C#, Python, etc. can also become an essential part of the area of code clone detection to meet the real and current demands of the software industry.

Acknowledgments
This work has been supported by a research grant from the Science and Engineering Research Board (SERB), Department of Science and Technology, Government of India under the Early Career Research Award Scheme.

7. References

[1] Min H and Li Ping Z 2019 Proceedings of the 2019 3rd International Conference on Management Engineering, Software Engineering and Service Sciences - ICMSS 2019 (New York, New York, USA: ACM Press) pp 9–16 ISBN 9781450361897 URL http://dl.acm.org/citation.cfm?doid=3312662.3312707
[2] Rattan D, Bhatia R and Singh M 2013 Information and Software Technology 55 1165–1199 ISSN 09505849 URL https://linkinghub.elsevier.com/retrieve/pii/S0950584913000323
[3] Li L, Feng H, Zhuang W, Meng N and Ryder B 2017 Proceedings - 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017 ISBN 9781538609927
[4] Jadon S 2016 2016 International Conference on Computing, Communication and Automation (ICCCA) (IEEE) pp 399–303 ISBN 978-1-5090-1666-2 URL http://ieeexplore.ieee.org/document/7813733/
[5] Sheneamer A and Kalita J 2016 2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA) (IEEE) pp 1024–1028 ISBN 978-1-5090-6167-9 URL http://ieeexplore.ieee.org/document/7838289/
[6] Chugh G, Mahajan D, Sehgal N, Pauland A and Budhiraja L 2019 International Research Journal of Engineering and Technology (IRJET) 06(4) 3962–3967 ISSN 2395-0056
[7] Thaller H, Linsbauer L and Egyed A 2020 2020 IEEE 14th International Workshop on Software Clones (IWSC) (IEEE) pp 64–69 ISBN 978-1-7281-6269-0 (Preprint 2001.07399) URL https://ieeexplore.ieee.org/document/9047635/
[8] Ghosh A and Kuttal S K 2018 2018 IEEE Symposium on Visual Languages and HumanCentric Computing (VL/HCC) (IEEE) pp 315–317 ISBN 978-1-5386-4235-1 ISSN 19436106 URL https://ieeexplore.ieee.org/document/8506550/
[8] Alomari H W and Stephan M 2018 2018 IEEE 12th International Workshop on Software Clones (IWSC) (IEEE) pp 58–59 ISBN 978-1-5386-6430-8 URL http://ieeexplore.ieee.org/document/8327320/
[9] Yang J, Hotta K, Higo Y, Igaki H and Kusumoto S 2015 Empirical Software Engineering 20 1095–1125 ISSN 1382-3256 URL http://link.springer.com/10.1007/s10664-014-9316-x
[10] Priyambadha B and Rochimah S 2014 2014 International Conference on Data and Software Engineering (ICODSE) (IEEE) pp 1–6 ISBN 978-1-4799-8175-5 URL http://ieeexplore.ieee.org/document/7062689/
[11] Krutz D E and Shihab E 2013 2013 20th Working Conference on Reverse Engineering (WCRE) (IEEE) pp 489–490 ISBN 978-1-4799-2931-3 ISSN 10951350 URL http://ieeexplore.ieee.org/document/6671332/
[12] Kamiya T 2013 2013 21st International Conference on Program Comprehension (ICPC) (IEEE) pp 227–229 ISBN 978-1-4673-3092-3 URL http://ieeexplore.ieee.org/document/6613854/
[13] Yu H, Lam W, Chen L, Li G, Xie T and Wang Q 2019 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC) (IEEE) pp 70–80 ISBN 978-1-7281-1519-1 URL https://ieeexplore.ieee.org/document/8813290/
[14] Elva R and Leavens G T 2012 2012 6th International Workshop on Software Clones (IWSC) (IEEE) pp 80–81 ISBN 978-1-4673-1795-5 URL http://ieeexplore.ieee.org/document/6227874/
[15] Yoshioka S, Yoshida N, Fushida K and Iida H 2011 International Symposium on Software Reliability Engineering (ISSRE 2011)
[16] Schugerl P, Rilling J and Charland P 2011 2011 IEEE 35th Annual Computer Software and Applications Conference (IEEE) pp 486–491 ISBN 978-1-4577-0544-1 ISSN 07303157 URL http://ieeexplore.ieee.org/document/6032385/
[17] Kim H, Jung Y, Kim S and Yi K 2011 Proceeding of the 33rd international conference on Software engineering - ICSE ’11 (New York, New York, USA: ACM Press) p 301 ISBN 9781450304450 ISSN 02705257 URL http://portal.acm.org/citation.cfm?doid=1985793.1985835
[18] Jiang L and Su Z 2009 Proceedings of the eighteenth international symposium on Software testing and analysis - ISSTA ’09 (New York, New York, USA: ACM Press) p 81 ISBN 9781605583389 URL http://portal.acm.org/citation.cfm?doid=1572272.1572283
[19] Gabel M, Jiang L and Su Z 2008 Proceedings of the 13th international conference on Software engineering - ICSE ’08 (New York, New York, USA: ACM Press) p 321 ISBN 9781605580791 ISSN 02705257 URL http://portal.acm.org/citation.cfm?doid=1368088.1368132
[20] Dandois C and Vanhoof W 2013 Logic-Based Program Synthesis and Transformation vol 7844 ed Albert E (Springer Berlin Heidelberg) chap 4, pp 35–50 ISBN 978-3-642-38196-6 URL http://dx.doi.org/10.1007/978-3-642-38197-3_4 http://link.springer.com/10.1007/978-3-642-38197-3_4