Model Clone Detection in Logisim

Benjamin Vandersmissen
Brent Van Bladel
Serge De Meyer

June 3, 2020

abstract: Model based clone detection is a new and rising field within clone detection. Several techniques have already been developed and succesfully applied to Simulink and UML models. In this paper we will implement one of those techniques [1] and a modified version and compare them both for a number of Logisim models and discuss if these techniques are useful in practice.

1 Introduction

Code clones are similar or identical fragments of code in a codebase. They are either artifacts introduced by copy-paste behaviour or the result of unintended duplicate functionality. The presence of code clones can have negative effects on the maintainability and clearness of the code. [2] An example of of one such negative effect, is that changes to one fragment should be carried out in all the corresponding fragments. If not all fragments are changed, this might introduce unintended defects in the code. Consequently, we need to deal with this problem, this is where code clone detection comes into play.

The phenomenon of duplicated functionality isn't limited to software code development. [3, 4] In model-based development, duplicated functionality can also be introduced by copying model fragments. As such, they will also form a problem during development and need to be dealt with, much in the same way as code clones.

The modelling language we will discuss in this paper is Logisim. Logisim [5] is a graphical system for logic circuit design and simulation. It is used in classrooms to introduce the concept of logic circuits and provides an environment for students to design, implement and test the circuits learned during classes.

An example of a Logisim model with cloned functionality can be found in figure 1. The circuit in this model calculates the two-bit sum of 2 two-bit numbers. As indicated, part of this model is duplicated. More specifically, the calculation of the sum is the same for each bit. This introduces needless complexity and makes it easier for accidental defects to occur. If we replaced these clones by a single sub model, the circuit is a lot clearer and there is only one place where a defect could occur. A version where the duplicate components are simplified, can be found in figure 1

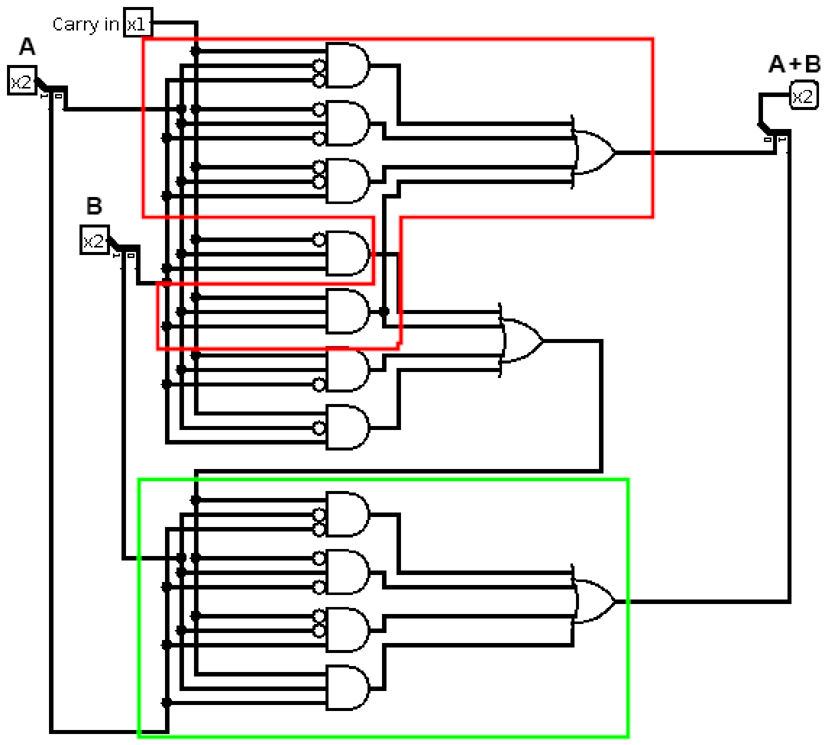

Figure 1: A two bit adder with no carry-out. The two duplicated fragments are indicated

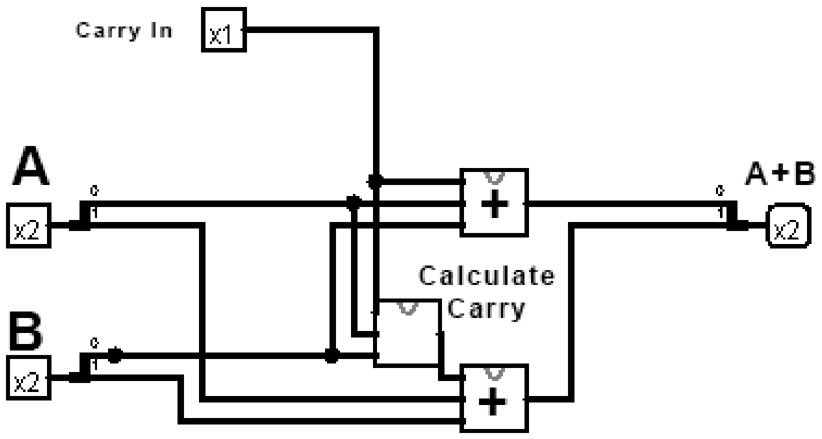

Figure 2: The same circuit as in figure 1, but with the duplicated fragments simplified

The most commonly used approach to detect clones in models is by converting the model to a graph and using a graph-mining algorithm to generate candidate clones. A candidate clone is a connected subgraph of the model. These candidates will then be tested for similar candidates in the same model, to find clones. The algorithm that we will study in this paper is called e-scan, one of the two components of ModelCD, which was introduced by Pham et al. [1]. e-scan is a way to detect exact clones in a complete and accurate way.

The limiting factor in this approach is the number of candidates generated, quadratic in function of the model size. We can see that in a worst case scenario with a fully connected graph, the total amount of subgraphs in a graph with n edges is $2^n - 1$, as this is the powerset of the edges minus the empty set. A model in Logisim has the potential to generate a lot of candidates, in part due to components that can have a high connectivity (multiple inputs and/or outputs) and due to having a small set of commonly used components, i.e. the simple logic gates (AND, OR, NOT, ...). The challenge lies in minimizing the candidate generation step, for clone detection to be feasible in Logisim.

In this paper we investigate whether it is feasible to adapt the current state-of-the-art clone detection approach for Logisim models. We propose a modified version of the e-scan algorithm and compare the amount of generated candidates by running both algorithms on a set of real-world Logisim models.

The rest of the paper is structured as following. In section 2, we give more information about model clones and the graph representation, in section 3 we introduce related work, section 4 explains the algorithm, section 5 provides the experimental setup used, section 6 is used to review a real-world use case, section

7 gives some conclusions and in section 8 we propose future work.

2 Model Clones and Graph representation

As described by H.Storrle (2013) [6], there are 4 types of model clones, roughly corresponding to the same categories in code clones:

- Type A: Exact Model Clones. Two models are exact model clones if they are fully identical in form and in attributes.
- Type B: Modified Model Clones, these are clones that have small changes in attributes.
- Type C: Renamed Model Clones. Clone Fragments that have large difference in attributes, with addition or removal of components also possible.
- Type D: Semantic Model Clones. Two models that implement the same functionality, but aren't derived from one another.

For our purpose, we will use a slightly adapted definition for Type A clones by not including a few attributes in this rule. The position, size and orientation of a component, while they are vital attributes to determine which connections are formed between components, can't be used to differentiate between two models after calculating the connections, because Logisim doesn't allow two components with the same position.

A Logisim model is represented as a sparse, labelled directed multi-graph. Each edge represents a connection between two components and is labelled with additional information for the connection, such as which output is connected to which input and the types of components.

Clones in this model are considered as non-overlapping connected subgraphs that are isomorphic. A clone group is a collection of identical clones. To reduce the amount of clone groups a user receives as feedback, we can remove covered clone groups without any loss of information. A clone group CG is covered, if there exists another clone group CG' of a higher degree, such that each clone in CG is a subgraph of a clone in CG'.

A clone Layer L_k contains al the cloned fragments with degree k. We can define a clone lattice as the layered graph of all clone layers $L_1, ..., L_k, ...$ with each node being a cloned fragment f_i with the subscript denoting the degree of the fragment. An edge in the clone lattice denotes a generating relationship between f_i and f_{i+1}, where we can generate f_{i+1} by adding a single edge. The e-scan algorithm traverses this clone lattice depth-first, which is more memory efficient, but causes some candidates to be generated multiple times resulting in long run times. In this paper, our approach will be to traverse the clone lattice breadth-first and report the differences.

In this feasibility study, we will focus our efforts on Type A clone detection.

3 Related Work

3.1 Code Clone Detection

Code Clone Detection can be roughly divided in 4 domains : textual, lexical, syntactic and semantic. Text-based clone detection work on the raw code using methods such as fingerprinting [7], dotplots [8] and pretty-printing [9]. Lexical approaches tokenize the code first and then the token sequence is scanned for duplicate subsequences, an efficient tool Dup [10] was introduced by Brenda Baker for this purpose using suffix trees. Kamiya et al. [11] combined suffix trees with transformation rules in their program CCFinder. Syntactic approaches convert the source code to a parse tree or an Abstract Syntax Tree (AST) and use either tree matching algorihtms [12] or metrics-based methods [13]. Semantic approaches use static program analysis by representing the source as a Program Dependency Graph (PDG) and finding isomorphic subgraphs, such as Krinke (2001) [14]. Hummel et al. (2010) [15] uses an index to provide scalable and incremental clone detection for large codebases. As code clone detection is a relatively mature field, multiple overviews and comparisons of the different approaches have been made, with no definitive ideal approach. [16–18]

3.2 Model Clone Detection

The field of model-based clone detection is relatively new and as such there are fewer approaches [3]. Most of research focuses on the simulink modelling language, as it is widely used in the field [19, 20]. Deissenboeck et al. (2008) [4] proposed a fast heuristic to detect clones for graph-based models, which was later implemented in the tool CloneDetective [21] and the tool ConQAT [4]. Al-Batran et al. (2011) improved upon this by first applying a set of normalisation rules to increase the performance. Pham et al. (2009) [1] introduced ModelCD, another tool to detect exact and approximate clones in Matlab Simulink models. Nguyen H.A. et al. (2009) [22] introduced Exas, as a way to use feature vectors to find clones in structure-based software artifacts. Hummel et al. (2011) [23] pioneered an index-based approach, by calculating hashes for model clones and storing them in a central index. Ö. Babur et al. (2019) [24] introduced a way to approach model clone detection as an information retrieval problem in the SAMOS framework and uses VSM and the dbscan clustering algorithm to find clones.

4 Algorithm

4.1 Preprocessing

A Logisim circuit is saved in a .CIRC file, which is a file format based on a fixed subset of the XML file format. This specification consists of two main parts, a part that describes the toolbars and default values for the used components and a part that describes the circuit with all its subcircuits, components and wires. For the clone detection algorithm, we only need this second part. In the next part, we will discuss this part of the specification in more detail.
A circuit is defined by the <*circuit*> tag which has a attribute *name*. A circuit contains wires and components, defined by the tags <*wire*> and <*comp*>.
A wire connects 2 different positition, so it has an attribute *from* and an attribute *to*.
A component has 3 attributes: *lib*, the library of the component, *loc*, the location, and *name* the class of the component.
Optional and class dependent attributes for a component can be defined by adding the tag <*a name= val=* > as a child of the component tag. Most of these attributes will not be used by the algorithm, save for *size*, *facing* and *inputs*.

Before we can run our graph-based approach, we first need to convert our Logisim model to a graph. Assume we have already parsed the XML file to generate a list of components and a list of wires. Then based on the size, orientation, number of input / output ports and position of components, the positions of inputs and outputs are calculated for each component. We then loop over each output port of each component to find the wires directly connected to the output port. Finally, we loop over each component and check if the wires connect to any input ports.

Algorithm 1 Generate the graph for a Logisim model

```
for component ∈ components do
    for outport ∈ component.out do
        connected_wires ← wires connected to outport
        for component2 ∈ components do
            for inport ∈ component2.in do
                if connected_wires connects to inport then
                    connect outport of component to inport of component2 in resulting graph
                end if
            end for
        end for
    end for
end for
```

4.2 Algorithm

The algorithm used is a modified version of the escan algorithm introduced in Pham et al (2009) [1]. The main difference between our algorithm and escan, is that escan finds clones in depth-first fashion, while our algorithm finds clones in breadth-first fashion.
The algorithm starts with a candidate set of degree 1, i.e. all clone candidates with one edge and generates L_i in iteration i.

Algorithm 2 Full Algorithm

```
1: candidate_set ← {all 1-candidates}
2: while candidate_set not empty do
3:     find_clones_and_prune(candidate_set)
4:     extend(candidate_set)
5:     filter_heuristic(candidate_set)
6: end while
7: for each $L_k$ do
8:     $CG \leftarrow CG \cup L_k$
9: end for
10: filter(CG)
11: return CG
```

4.2.1 Finding Clones and pruning

Canonical labelling [25] is a technique used to generate a unique label for a graph that is dependent on the structure of the graph. This means that two isomorphic graphs will have the same canonical labelling. We can use canonical labelling as a fast way to determine if two graphs are a clone pair, by comparing the label. Note that we assume for our implementation, that our circuit has no loops, i.e. we work with an acyclic graph.

We calculate the canonical labelling for each candidate in the candidate set and map the candidate to the label. This means that after a full iteration over the candidate set, we know the clone groups for the candidate sets. If a label has more than one candidate labeled to it, we know there are clones. We also know if a label has only one associated candidate, that we can prune that candidate from the candidate set, because it has no clones.

Pruning candidates with no clones is essential for the scalability for the algorithm. Due to monotonicity, if a graph G has no clones, all graphs containing G have no clones as well. As we are interested in clones, we don't need to generate candidates we know won't produce clones. This means the earlier we can prune a path, the less useless candidates we generate.

Algorithm 3 Finding clone groups and pruning

```
 1: for candidate ∈ candidate_list do
 2:     map candidate to canonical label
 3: end for
 4: for label ∈ canonical_labels do
 5:     if label has one associated candidate then
 6:         prune candidate from candidate list
 7:     else
 8:         create clone group from associated candidates
 9:     end if
10: end for
```

4.2.2 Extending Candidate Set

As we want to generate all clones in a model, we need a way to increase the degree of our candidate set, this is done in the extending phase, where we generate the candidates of a higher degree.

Extending the candidate set means going from candidates with *i* edges to candidates with *i+1* edges. We do this by adding an extra edge to each candidate. We can improve the performance of this part by only adding an edge that is a clone fragment. This is due to monotonicity [1]: A graph G containing a non-cloned edge E will never have an associated clone G'. We also populate the parent children relation, we use to efficiently remove covered Clone Groups.

Algorithm 4 Extending the clone groups

```
1: for candidate1 ∈ candidate_list do
2:     for candidate2 ∈ cloned_edges do
3:         if candidate1 can connect to candidate2 then
4:             new candidate ← candidate1 ∪ candidate2
5:             populate $parent \rightarrow children$ relation
6:         end if
7:     end for
8: end for
```

4.2.3 Removing Covered Clone Groups

As discussed earlier, we want to remove the covered clone groups from our output, as to not overwhelm the user with useless information.

After each iteration, we have generated L_k, we can remove most covered Clone-Groups of size k-1 using a heuristic. The heuristic is that if a clone group CG' generated from CG has the same amount of clones, CG is covered by CG'.We populate the parent-children relation during the extending phase, by creating a mapping from a label of size k-1, to each label of size k it generates. Afterwards, we loop over each CloneGroup in L_{k-1} and its generated children and remove the CloneGroup if the amount of clones is the same.

Algorithm 5 Remove covered clone groups

```
1: for group ∈ $L_{k-1}$ do
2:     if a child C has the same amount of clones then
3:         remove group from $L_{k-1}$
4:     end if
5: end for
```

5 Experimental Setup

The dataset used in the experiments was provided by the university of Antwerp. It is a collection of different low level circuits, such as adders, implemented by different groups of students.

We use three different models in our experiments. Model 1 is a triplet of 3 cloned, disjunct subgraphs, for a total of 9 nodes and 9 edges. Model 2 consists of two cloned parts, in total having 20 nodes and 22 edges. This model has a lot of different components and a low connectivity. Model 3 has two duplicated 1-bit adders for a total of 18 nodes and 34 edges. This model has a low variation in components and a high connectivity. An overview can be found in table 1.

Dataset	# nodes	# edges
Dataset 1	9	9
Dataset 2	20	22
Dataset 3	18	34

Table 1: Datasets with their nodes and edges

6 Results

6.1 Overview

Dataset	Generated Candidates	Time (s)	Peak memory (kB)
1	90	0.00	4212
2	5446	0.91	6800
3	6,023,656	1613	505,034

Table 2: e-scan (depth-first)

Dataset	Generated Candidates	Time (s)	Peak memory (kB)
1	36	0.00	4306
2	4842	0.75	8460
3	1,723,134	390	946,512

Table 3: our approach (breadth-first)

For Dataset 1, the modified e-scan algorithm generated 36 candidates, while the original algorithm generated 90. The total number of unique candidates is 21. We found that each implementation generates some candidates multiple times. The original e-scan algorithm had almost three times as much total generated candidates and generated 69 candidates more than the optimal solution, our modified version had an overhead of only 15 candidates, which is a reduction of almost 80%.

For dataset 2, the modified e-scan algorithm generated 2684 candidates, while the original e-scan algorithm generated slightly more, namely 4842. This is the result of the low connectivity in the model and the variation in components, with each component having less than two connections on average.

For dataset 3, the original e-scan algorithm generated 6.023.656 candidates, while our modified algorithm generated only 1.723.134 candidates. For this dataset, the difference between the depth-first approach and the breadth-first approach is more pronounced. This circuit has a small variation in components, paired with a relatively high connectivity, which leads to multiple candidates being generated a lot of times during the depth-first approach.

6.2 Discussion

The results of these experiments show that there is a clear advantage to using the breadth-first approach over the depth-first approach, because it fundamentaly generates less candidates, which in turn improves the performance, while still finding all the clone groups. This advantage diminishes if the connectivity is lower. Note that each dataset consisted purely out of cloned fragments. As most of the candidates are generated as subsets of cloned fragments, we can generalize from these conclusions for more extensive models.

The reason why the depth-first approach generates more candidates, is because each time we extend a clone, we generate all candidates for each clone in the clone group, as we need these for going down to the next level. The main issue is that we generate all candidates for the clone group each time we extend a clone. In a breadth first approach, we only need to generate candidates for each clone group once, as we generate all candidates of a level in a single pass.

7 Conclusion

While clone detection algorithms can be succesfully applied to the Logisim modelling language, due to the nature of Logisim, finding valuable clones is a time consuming procedure. In this paper, we have proposed a modified depth-first version of the e-scan algorithm, that generates candidates more time efficient at a higher memory cost, which makes it more suitable for small Logisim models. However, the performance of the algorithms is dependent on size of the models and also the size of the clones in the models. While we know the size of the model, it is impossible to know all clones in a model. [3]

Logisim is a low level modelling language, modelling logic circuits. As such, we are interested in finding large clones, involving multiple components. Such clones will also have a large amount of edges. Due to the nature of the algorithm, finding a clone group of n edges, means generating all possible subgraphs of size 1 to n-1. In the worst case scenario this means generating $\binom{n}{k}$ unique graphs with k edges, or $2^n - 1$ unique graphs and their unique labels for each fragment. This means that the practical limitations are very much dependent on the size and structure of a graph.

As such, complete clone detection is virtually impossible in larger Logisim models, as it is either a time heavy or memory heavy operation depending on the way we traverse the clone lattice. In case of large models, heuristic approaches [4] that limits the branching factor to 1 may be better suited, although then there is no way to accurately estimate the precision or recall.

8 Future work

The relative efficiency of clone detection in logisim versus clone detection in Simulink could be investigated. Simulink is more high-level, so useful clone fragments are likely a lower size and most clones wont have as many edges as in Logisim. Simulink models likely will also have a more diverse set of blocks, which would mean that there are more uniquely labelled edges, which would further increase the efficiency of the algorithm by reducing the amount of early candidates.

References

[1] N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen, "Complete and accurate clone detection in graph-based models," in *2009 IEEE 31st International Conference on Software Engineering*, pp. 276–286, IEEE, 2009.

[2] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, “Do code clones matter?,” in *2009 IEEE 31st International Conference on Software Engineering*, pp. 485–495, IEEE, 2009.

[3] F. Deissenboeck, B. Hummel, E. Juergens, M. Pfaehler, and B. Schaetz, “Model clone detection in practice,” in *Proceedings of the 4th International Workshop on Software Clones*, pp. 57–64, 2010.

[4] F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J.-F. Girard, and S. Teuchert, “Clone detection in automotive model-based development,” in *2008 ACM/IEEE 30th International Conference on Software Engineering*, pp. 603–612, IEEE, 2008.

[5] C. Burch, “Logisim: a graphical system for logic circuit design and simulation,” *Journal on Educational Resources in Computing (JERIC)*, vol. 2, no. 1, pp. 5–16, 2002.

[6] H. Störrle, “Towards clone detection in uml domain models,” *Software & Systems Modeling*, vol. 12, no. 2, pp. 307–329, 2013.

[7] J. H. Johnson, “Identifying redundancy in source code using fingerprints,” in *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering-Volume 1*, pp. 171–183, IBM Press, 1993.

[8] S. Ducasse, M. Rieger, and S. Demeyer, “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)*, pp. 109–118, IEEE, 1999.

[9] J. R. Cordy, T. R. Dean, and N. Synytskyy, “Practical language-independent detection of near-miss clones,” in *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, pp. 1–12, IBM Press, 2004.

[10] B. S. Baker, “A program for identifying duplicated code,” *Computing Science and Statistics*, pp. 49–49, 1993.

[11] T. Kamiya, S. Kusumoto, and K. Inoue, “Ccfinder: a multilinguistic token-based code clone detection system for large scale source code,” *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[12] I. D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier, “Clone detection using abstract syntax trees,” in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pp. 368–377, IEEE, 1998.

[13] J. Mayrand, C. Leblanc, and E. Merlo, “Experiment on the automatic detection of function clones in a software system using metrics.,” in *icsm*, vol. 96, p. 244, 1996.

[14] J. Krinke, "Identifying similar code with program dependence graphs," in *Proceedings Eighth Working Conference on Reverse Engineering*, pp. 301–309, IEEE, 2001.

[15] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-based code clone detection: incremental, distributed, scalable," in *2010 IEEE International Conference on Software Maintenance*, pp. 1–9, IEEE, 2010.

[16] F. Van Rysselberghe and S. Demeyer, "Evaluating clone detection techniques from a refactoring perspective," in *Proceedings. 19th International Conference on Automated Software Engineering, 2004.*, pp. 336–339, IEEE, 2004.

[17] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of computer programming*, vol. 74, no. 7, pp. 470–495, 2009.

[18] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.

[19] M. Stephan, M. H. Alafi, A. Stevenson, and J. R. Cordy, "Towards qualitative comparison of simulink model clone detection approaches," in *2012 6th International Workshop on Software Clones (IWSC)*, pp. 84–85, IEEE, 2012.

[20] H. Petersen, "Clone detection in matlab simulink models," *Master's thesis, Technical University of Denmark*, 2012.

[21] E. Juergens, F. Deissenboeck, and B. Hummel, "Clonedetective-a workbench for clone detection research," in *2009 IEEE 31st International Conference on Software Engineering*, pp. 603–606, IEEE, 2009.

[22] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Accurate and efficient structural characteristic feature extraction for clone detection," in *International Conference on Fundamental Approaches to Software Engineering*, pp. 440–455, Springer, 2009.

[23] B. Hummel, E. Juergens, and D. Steidl, "Index-based model clone detection," in *Proceedings of the 5th International Workshop on Software Clones*, pp. 21–27, 2011.

[24] Ö. Babur, L. Cleophas, and M. van den Brand, "Metamodel clone detection with samos," *Journal of Computer Languages*, 2019.

[25] L. Babai and E. M. Luks, "Canonical labeling of graphs," in *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pp. 171–183, 1983.