Model-Based Clone Detection in Logisim

Benjamin Vandersmissen
Brent Van Bladel
Serge De Meyer

May 20, 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 for the Logisim modelling language and see if it is 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. 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.

In the field of model-based development, the phenomenom of clones can arise as well. As such, they will also form a problem during development and need to be dealt with, much in the same way as code clones.

Logisim**[1]** 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

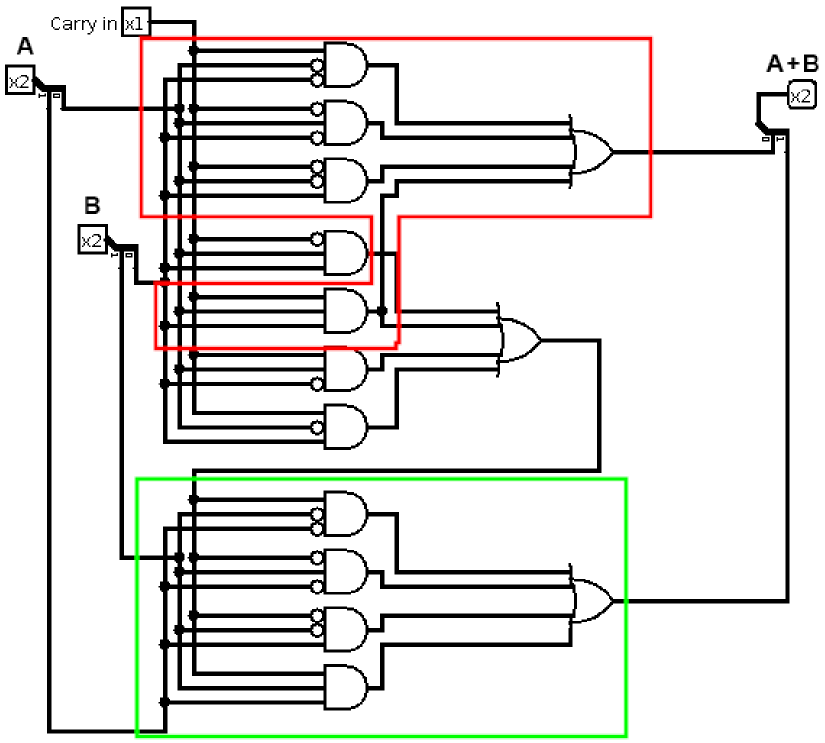

Figure 1: A two bit adder with no carry-out. The two duplicated fragments are indicated

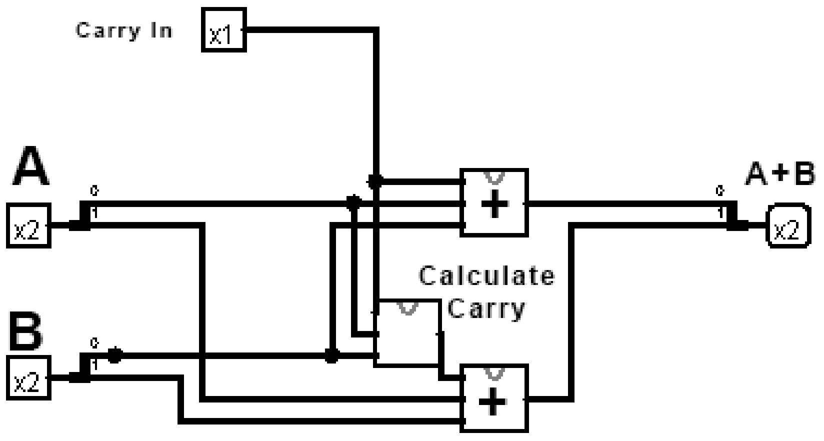

Figure 2: The same circuit as in figure **??**, but with the duplicated fragments simplified

Logisim is a graph-based modelling language. The most commonly used approach to detect clones in graphs is by generating candidates from known clones and testing these generated candidates. The current state-of-the-art algorithm that implements this approach is called e-scan, which was introduced by Pham et al.**[2]** and is used in ModelCD. The limiting factor in this approach is the number of candidates generated, because this is exponential. A model in Logisim generates a lot of candidates, in part due to components that can have a high connectivity (a lot of inputs) and due to having a small set of commonly used components. The challenge lies in minimizing the candidate generation step, for clone detection to be feasible in Logisim.

In this paper we investigate wether it is feasible to improve 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.

2 Related Work

The rest of the paper is structured as following. In section 2 we introduce related work, section 3 explains the algorithm, section 4 provides the experimental setup used, section 5 will be used to review a real-world use case, section 6 will give some conclusions and in section 7 we will propose future work.

2.1 Clones In General

As described by **H.Storrle (2013)** [3], 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.

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

2.2 Model Clone Detection

In the field of model-based clone detection, **Deissenboeck et al. (2008)** [4] proposed a fast heuristic to detect clones for graph-based models. **Pham et al. (2009)** [2] introduced ModelCD, a tool to detect exact and approximate clones in Matlab/Simulink models. **Nguyen H.A. et al. (2009)** [5] introduced Exas, a way to use feature vectors to find clones in structure-based software artifacts.

2.3 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 (**Johnson '93** [6]) and dotplots (**Ducasse et al.** [7]). Lexical approaches tokenize the code first and then the token sequence is scanned for duplicate subsequences, an efficient tool Dup **[8]** was introduced by Brenda Baker for this purpose. Syntactic approaches convert the source code to a parse tree or an Abstract Syntax Tree (AST) and use either tree matching algorihtms (**[9]**) or metrics-based methods (**[10]**). Semantic approaches use static program analysis by representing the source as a Prorgram Dependency Graph (PDG) and finding isomorphic subgraphs, such as **Krinke (2001)**[11]

3 Algorithm

3.1 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 posititions, 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

```
1:  for component ∈ components do
2:      for outport ∈ component.out do
3:          connected_wires ← wires connected to outport
4:          for component2 ∈ components do
5:              for inport ∈ component2.in do
6:                  if connected_wires connects to inport then
7:                      connect outport of component to inport of component2 in re-
                        sulting graph
8:                  end if
9:              end for
10:         end for
11:     end for
12: end for
```

3.2 Algorithm

The algorithm used is a modified version of the **escan** algorithm introduced in **Pham et al (2009) [2]**. 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 size 1, i.e. all clone candidates with one edge and generates $\mathrm{L_i}$ 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
```

3.2.1 Finding Clone Groups and pruning

Canonical labelling 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 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. Overlapping clones with the same label are also removed during this procedure.

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
```

3.2.2 Extending Candidate Set

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: 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
```

3.2.3 Removing Covered Clone Groups

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
```

4 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 datasets in our experiments. Dataset 1 is a triplet of 3 cloned, disjunct subgraphs, for a total of 9 nodes and 9 edges. Dataset 2 consists of two cloned models, in total having 20 nodes and 22 edges. This dataset has a lot of different components and a low connectivity. Dataset 3 has two duplicated 1-bit adders for a total of 18 nodes and 34 edges. These models have 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

5 Results

5.1 Overview

Dataset	e-scan	our algorithm
Dataset 1	90	36
Dataset 2	5446	2684
Dataset 3	6.023.656	153.032

Table 2: Amount of generated candidates for each dataset and algorithm

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 more than twice the amount, namely 5446. In this dataset, the original e-scan algorithm generated only around twice the amount of candidates. This is likely a 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 153.032 candidates. For this dataset, the difference between the depth-first approach and the breadth-first approach is even more staggering. 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.

5.2 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. 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.

6 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 Logisim models.

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.

7 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.

8 References

1. ”Logisim: A graphical system for logic circuit design and simulation.” Journal of Educational Resources in Computing 2:1, 2002, pages 5-16
2. Pham, Nam & Nguyen, Hoan & Nguyen, Tung & Al-kofahi, Jafar & Nguyen, Tien. (2009). Complete and accurate clone detection in graph-based models. 276-286. 10.1109/ICSE.2009.5070528
3. H.Storrle, Towards clone detection in UML domain models, in: Proceedings Software & Systems Modeling, Volume 12, Issue 2, 2013,pp.307-329.
4. Deissenboeck, Florian & Hummel, Benjamin & Jürgens, Elmar & Schätz, Bernhard & Wagner, Stefan & Girard, Jean-Francois & Teuchert, Stefan. (2008). Clone Detection in Automotive Model-Based Development.. Proceedings - International Conference on Software Engineering. 57-67. 10.1145/1368088.1368172.
5. Nguyen H.A., Nguyen T.T., Pham N.H., Al-Kofahi J.M., Nguyen T.N. (2009) Accurate and Efficient Structural Characteristic Feature Extraction for Clone Detection. In: Chechik M., Wirsing M. (eds) Fundamental Approaches to Software Engineering. FASE 2009. Lecture Notes in Computer Science, vol 5503. Springer, Berlin, Heidelberg
6. J. Johnson, Identifying redundancy in source code using fingerprints, in: Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research, CASCON 1993, 1993, pp. 171–183

7. S. Ducasse, M. Rieger, S. Demeyer, A language independent approach for detecting duplicated code, in: Proceedings of the 15th International Conference on Software Maintenance, ICSM 1999, 1999, pp. 109–118.

8. B. Baker, A program for identifying duplicated code, in: Proceedings of Computing Science and Statistics: 24th Symposium on the Interface, vol. 24, 1992, pp. 49–57.

9. I. Baxter, A. Yahin, L. Moura, M. Anna, Clone detection using abstract syntax trees, in: Proceedings of the 14th International Conference on Software Maintenance, ICSM 1998, 1998, pp. 368–377.

10. J. Mayrand, C. Leblanc, E. Merlo, Experiment on the automatic detection of function clones in a software system using metrics, in: Proceedings of the 12th International Conference on Software Maintenance, ICSM 1996, 1996, pp. 244–253.

11. J. Krinke, Identifying similar code with program dependence graphs, in: Proceedings of the 8th Working Conference on Reverse Engineering, WCRE 2001, 2001, pp. 301–309