Model-Based Clone Detection in Logisim

Benjamin Vandersmissen
Brent Van Bladel
Serge De Meyer

May 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 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.

In this paper we will propose a modified version of the e-scan algorithm introduced by Pham et al and use it to discuss the feasability of clone detection in Logisim models based on a few real-world examples.

The rest of the paper is structured as following. In section 2 we introduce related work, section 3 provides some background for the process, section 4 explains the experimental setup, 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 Related Work

2.1 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.

2.2 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 Background

This section introduces some terms and concepts that will be used throughout the rest of the paper.

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*.

A Clone Fragment of size k is a connected subgraph containing k edges, that is an exact clone of another subgraph in the graph.

A Clone Candidate of size k is a connected subgraph of our graph containing k edges, that could be a clone of another subgraph.

A Clone Group of size k is a collection of multiple non-overlapping clone fragments of size k that are clones of eachother. A Clone Group CG is covered by another Clone Group CG', if for each clone C in CG there is a clone in CG' that contains C. If a Clone Group is covered, we can discard it from the output without loss of information.

A Clone Layer L_k is a collection of all Clone Groups of size k.

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.

Monotonicity means that if a graph G has clones, a subgraph SG of G will have clones as well. If a graph G has no clones, all graphs G' that contain G, will have no clones as well.

3.1 About Clones

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. This is because in Logisim, every component has a positional attribute. One could argue that a change in this attribute means that the clone should be of type B, but in Logisim it is functionally impossible to have two components at the same position. This would mean there are no type A clones, so we discount the position attribute from the general rule of thumb.

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

4 Experimental Setup

4.1 Assumptions

For our canonical labeling function to work, we assume that our graph has no loops. Loops are sometimes possible in the Logisim environment, but they are almost never justifiable to use.

4.2 Preprocessing

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 $component \in components$ do
    for $outport \in component.out$ do
        connected_wires ← wires connected to outport
        for $component2 \in components$ do
            for $inport \in component2.in$ 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
```

4.3 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
```

4.3.1 Finding Clone Groups and pruning

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
```

4.3.2 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 noncloned 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 $\in$ candidate_list do
2:     for candidate2 $\in$ cloned_edges do
3:         if candidate1 can connect to candidate2 then
4:             new candidate $\leftarrow$ candidate1 $\cup$ candidate2
5:             populate parent $\rightarrow$ children relation
6:         end if
7:     end for
8: end for
```

4.3.3 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 $\in L_{k-1}$ 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 Results

5.1 Experiment 1

The first experiment consist of two files both containing an implementation of a 1-bit adder done by two different groups, based on the same circuit. Each model contains 15 components and 23 connections, resulting in a total graph of 30 components and 46 connections.

The runtime of this first experiment was approximately 20 minutes and during this timeframe a total of 1.331.349 unique candidates were generated, ultimately resulting in one clone group of size 23.

We found that both files form a single clone pair. This was the expected results, as both models were based on the same circuit.

5.2 Experiment 2

The second experiment consists of two files by different groups, implementing an assortment of different 1-bit circuits. The total graph of these models consists of 110 components and 162 connections.

The runtime of this experiment was approximately 5 minutes and generated a total of 671.501 unique candidates resulting in 206 clone groups of varying sizes, ranging from 7(the minimum size) to 17 connections.

Because our algorithm is an adapted version of the e-scan algorithm, we know the clone detection is complete. As expected, the largest clone groups are found between the circuits that implement more or less the same functionality, such as the 1-bit adder and 1-bit full adder.

5.3 Results

The results of two experiments show that the runtime (influenced by the generated candidates) is more dependent on the sizes of the clones in the graph rather than the amount of components and connections or even the amount of clones.

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. 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.

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