Using Mutation Analysis for a Model-Clone Detector Comparison Framework

Matthew Stephan   Manar H. Alafi   Andrew Stevenson   James R. Cordy

School of Computing, Queen's University, Kingston, Canada
{stephan,alafi,stevenson,cordy}@cs.queensu.ca

Abstract—Model-clone detection is a relatively new area and there are a number of different approaches in the literature. As the area continues to mature, it becomes necessary to evaluate these approaches against each other and validate new ones that are introduced. We present a mutation-analysis based model-clone detection framework that attempts to automate and standardize the process of comparing multiple Simulink model-clone detection tools or variations of the same tool. By having such a framework, new research directions in the area of model-clone detection can be facilitated as the framework can be used to validate new techniques as they arise. We begin by presenting challenges unique to model-clone tool comparison including recall calculation, the nature of the clones, and the clone report representation. We propose our framework, which we believe will address these challenges. This is followed by a presentation of the mutation operators that we plan to inject into our Simulink models that will introduce variations of all the different model clone types that can then be searched for by each respective model-clone detector.

I. Introduction

Model-Driven Engineering (MDE) is fairly prevalent in Software Engineering and has seen substantial adoption in the recent past. This can be witnessed in communications, automotive, and other embedded software areas. As the projects created through MDE begin to age and continually grow, the necessity of detecting model clones becomes more important. Model-clone detection, a relatively new research area, involves finding sets of software models that are identical or similar to one another with respect to some measure of similarity. The results from model-clone detection can be used to reduce redundancy, aid in system understanding and refactoring, and identify similar system components after error detection. Thus, it is of great interest to many in the Software Engineering community including our industrial partners. There are a number of approaches created that perform model-clone detection, however it is not clear which approach is best suited to what situations and how to evaluate individual tools. As such, there is a need for a standard way of comparing different model-clone detectors or the same model detector using different tuning parameters. Having a facility to do this will also help fuel research in this area as it will provide researchers a means with which they can try out and validate new techniques.

Previously, we discussed some initial ideas behind performing a qualitative evaluation of Simulink Model-clone detection approaches [1, 2], which was motivated by our desire to compare our new model-clone detection approach to other existing tools. While we are able to perform some basic qualitative analysis and find differences in the clones that can be detected by each tool, we still are faced with a number of challenges: recall computation; the different nature of the clones reported; and the different representations of the resulting clone classes and instances provided by each of the tools. These challenges are elaborated in Section III-A.

In this paper we intend to use mutation analysis and add a representation transformation step in order to devise a comparison framework that addresses the specific challenges in comparing model-clone detectors. Thus the contributions of this work are:

- A set of architectural mutational operators designed for Simulink model-clone detector comparison.
- A new framework that utilizes Simulink model mutation operators to allow for a more fair and quantitative analysis. Any model-clone detection tools that are developed subsequently can use the framework we are creating to evaluate themselves and to compare to existing tools.

II. Background

A. Model-Clone Detection

Model-clone detection entails discovering similar or identical groups of model elements. Unlike source code, which is represented as linear text, models are typically represented visually, as box-and-arrow diagrams. Model clones can thus be seen as similar sub graphs of these diagrams.

Code clone detection techniques can be categorized according to the types of clones they can identify [3]. In [2] we adopt a similar categorization for model clone types, which we list here briefly as the proposed mutational operators will be designed to emulate them. Type 1 (exact) model clones refer to identical model fragments, ignoring variations in visual presentation, layout, and formatting. Type 2 (renamed) model clones refer to structurally identical model fragments, ignoring variations in labels, values, types, and the variations from Type 1. Type 3 (near-miss) model clones refer to model fragments with further modifications, such as changes in position or connection with respect to other model fragments and small additions or removals of blocks or lines in addition to the variations from Type 1 and 2 clones.

Approaches intended to achieve model-clone detection can be classified into 2 categories: those that view detection as

a graph-matching problem and those that use the underlying textual representations. The most prevalent example of the graph-matching approach is *ConQAT* [4]. It performs model-clone detection on Simulink models by flattening the Simulink models' structure into graph nodes that are normalized and labeled to contain the information needed to measure similarity. As of now, they detect only exact clones. In contrast, *SIMONE* [2] is an example of an approach that uses the textual representations of Simulink models in order to find Type 1,2, and 3 clones. Potential clones identified by SIMONE are based on the Simulink hierarchical structure.

III. MODEL-CLONE DETECTION COMPARISON FRAMEWORK

In previous work [1], we identified six qualitative areas to evaluate in model clone detection tools: relevance, performance, clone detection type, user-interaction required, adaptability, and model-pattern granularity. Ideally, we want to be able to compare the strengths and weaknesses of each tool being compared with respect to these areas. In addition, by analyzing the top candidates in the field, we want to identify areas for improvement and suggest future research directions for model-clone detection in general. In this paper we use mutation analysis to help us evaluate some of these areas, specifically, relevance and recall computation; clone detection type; and model-patten granularity. Mutation analysis refers to a system evaluation that is based upon small modifications, or mutations, of a system's components and evaluating how the system handles these changes [5]. The mutations are derived from mutation operators that either are representative of common modifications made to a system's components or showcase an important property. So, for example, source-code mutations can be as simple as changing variable values or complex and tailored to a specific context, such as Java Concurrency [6].

A. Framework Design Challenges

In our early work manually comparing tools, we ran into a number of challenges. The first challenge was determining the recall for different tools: Specifically, of all the clones that exist in our systems, how many of them were reported by each tool. This proved difficult as we would first need to determine manually all the clones in our systems, which is impractical especially in the case of the large systems that will be provided from our industrial partners. This can be solved by introducing a specific mutation operation and having either zero or some baseline number of instances in our system that are already discovered by the respective tools to begin with. The key here is to generate and look for only the specific clones that come from one specific mutation operation at a time.

As we outlined earlier [2] and demonstrate in Figure 1, the second obstacle we found was coping with nested clones. Without going into too much detail, this issue arises because *SIMONE* reports only the outer most (sub-)system that satisfies the difference threshold whereas *ConQAT* reports identical clone groups and crosses subsystem boundaries. As shown
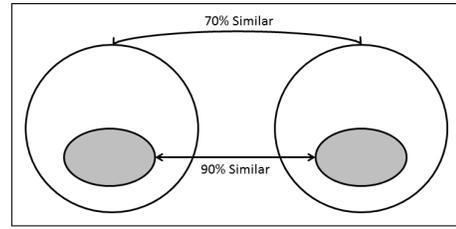

Fig. 1. Nested Clone Example [2]

in the figure, the outer circles, which represent a clone pair that is 70% similar, contain an inner clone pair with 90% similarity. If *ConQAT* reported the 90% similar (according to *SIMONE*) clone, we would not be able to find the equivalent clone in *SIMONE*'s result set as *SIMONE* only reports the 70% outer clone pair, as it is the largest subsystem meeting its threshold. Neither result is specifically undesirable, it just makes it difficult to find correspondences between the two tools. This issue of nested clones can be mitigated through mutation operations as well. Firstly, when comparing to threshold-configurable tools, the mutations being executed will be tailored to a specific level at a time and the tools will be set to look for only clones at that level. So, using the example in Figure 1, we would start with lower level clones, for example the 90% clone class in the figure, and ensure that the tools that are configurable are set to that threshold. For comparing a configurable tool and a tool that identifies only exact clone matches, we can adapt and extend the notion of *Fragment Containment* used for the source-code clone mutation analysis [7] by Roy and Cordy. Specifically, they note that if a detected clone contains the clone introduced through mutation, then it is acceptable to say that the mutant clone has been detected. This aligns with the definition of "killed" mutants traditionally used in mutation analysis as it is an example of the "non-overlap binary definition" of detection. They define *Fragment Containment* as it applies to code, but we must define it as it applies to models: *Fragment Containment* in models is the case where all blocks and lines belonging to the clone introduced through mutation are a subset of the blocks and lines of the detected clone instance.

Lastly, tools may provide different representations of their clone results. In our experiments, *ConQAT* represents clone classes and clone instances by grouping individual blocks into findings within finding groups in XML format. *SIMONE* also reports its clone classes and instances into XML, however, the schema of the XML has different elements including the textual source of the model, which comprises the majority of data in the file. We had to essentially do a manual comparison for completeness. We address this challenge by having our framework have a normalization facility that can transform clone result output into a common form, if necessary. For purposes of comparison, an XML format that lists only the blocks and lines sorted by both clone class and respective clone instances should suffice. For each new tool being compared, a transformation will have to be written only once, possibly with TXL [8], that takes output from the new tool and transforms it into this format.
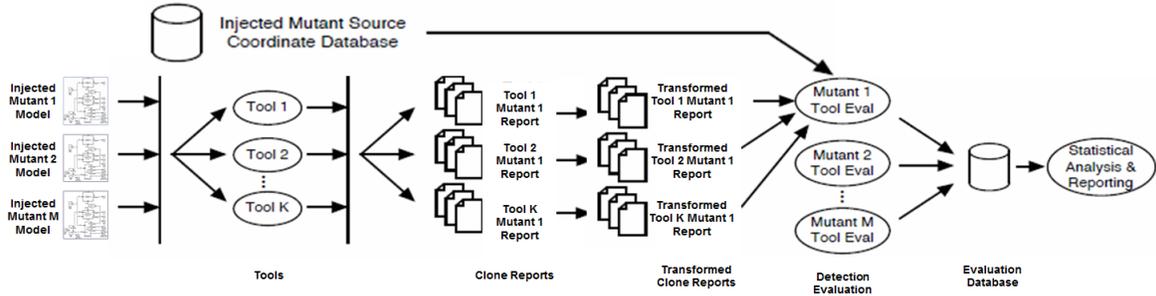
Fig. 2.   Proposed Model-Clone Detection Comparison Framework. Adapted from [3]

B. Framework Details

In this section we present our proposed framework, shown in Figure 2. While the general layout is similar to any mutational framework used for test case generation and tool evaluation in a corresponding source-code framework, such as the one from Roy and Cordy [7], our framework differs from all other techniques as it is addresses the model-specific challenges we presented earlier.

As shown in Figure 2, the process begins with each mutant-injected model being given as input to each of the respective model-clone detectors. The tools are then executed on each mutated model: As mentioned previously, any tool that has configurable similarity parameters will have them set to match the specific clone being injected. The output of this step are clone reports. New to this framework is the transformation of clone reports into the form we discussed previously. This is necessary in a model version of a framework as both the nature and representations of the model clones may differ and impact the evaluation. The manual creation of a transformation will have to be done only when a new tool is being used in the framework or an existing tool changes its output format. Otherwise, the transformation is automatic in that it can be executed on each of the resulting clone reports as they are generated. The transformed clone reports can then be subject to detection evaluation, including recall calculation, which should now be much more straightforward given that the blocks and connectors (lines) involved in the clone instances can be compared against those in the database, which contains all our mutated clones in the same format. As done in a source-code framework, the evaluation database stores the results of the detection evaluation that are then fed into a statistical analysis and reporting facility.

C. Simulink Model Mutation Operators

In this section we present a set of structural mutational operators we believe are essential to evaluate Simulink model-clone detection tools. Our proposed set of mutations are drawn from our own observations of potential model edit operations encountered in a large set of publicly available Simulink models available to us via both the Mathworks Simulink demos and Matlab Central. The development of *SIMONE* went through iterations of improvement. In each stage we enabled the tool to detect model-clone types we adapted from the literature that corresponded to source-code clone types.

1) Changing the Layout and Ordering of Elements: Mutational operators that enable the detection for Type 1 clones are mainly related to filtering models' layout and presentation attributes. Based on our experience with *SIMONE* development, filtering layout attributes significantly improved recall in finding exact and near-miss exact subsystem clones in the automotive example system. Furthermore, this allowed us to find larger subsystem clones, covering a much larger proportion of the extracted subsystems. We have done manual validation of all clones found in the automotive example models, showing that all were valid subsystem clones. However, this was fairly cumbersome thus exhibiting to us the importance of having an automatic framework.

We noticed that in the textual representation of each subsystem in Simulink the sequence of its underlying model elements may not be the same, even in identical subsystems. Sorting significantly increased the number of exact subsystem clones found, but allowed much larger near-miss clones to be identified, thus, reducing the total number reported. Based on our experiences with filtering and sorting, respectively, possible mutational operators to detect Type 1 model clones are:

- Change colour, position, size, and other layout attributes.
- Reorder blocks, lines, ports, and branches.

2) Renaming and Value Modification of Elements: To enable detection of Type 2 model clones, we require mutational operators that change the name and value attributes of the model elements. *SIMONE* anonymizes all names and values associated with elements and blocks, preserving only Block-Type and LineType elements for comparison. This allows for detection of type 2 and near-miss type 2 (type 3-2) subsystem clones in Simulink models. Therefore, possible mutational operators to detect Type 2 clones are:

- Rename Block
- Rename Lines
- Change Block Value

Renaming allowed us to find over 1,900 near-miss type 2 (type 3-2) subsystem clones in the automotive model set, covering 75% of the subsystems. This is consistent with the large number of different versions of models in the demonstration examples.

3) Change Subsystems Structure: Mutational operators to detect any addition, deletion, or moving of model elements within a subsystem are required to detect Type 3 model

clones. *SIMONE* is able to detect clones in this category by allowing for a difference threshold of 30% using the renaming techniques described previously. It should be noted that any addition or deletion of model elements must take into consideration preservation of model connectivity. Thus an "adding blocks" operator must consider adding only source blocks with its ports and connections to the rest of the model. This also applies to destination blocks. Such changes will have the minimum impact and should avoid an unconnected model. Possible Mutational operators are:

- Add or Delete block as destination. This involves sink blocks, lines (signals), and required ports.
- Add or Delete block as a source. This involves source blocks, lines (signals), and required ports.
- Change block type
- Change subsystem clone hierarchy

D. Planned Evaluation and Experiments

So far we have been evaluating results manually. For *SIMONE*, experiment results were reported in our previous work [2]. This included a preliminary experiment to compare SIMONE to ConQAT in which results were evaluated manually. We have since automated the process of normalizing clone reports as outlined in this paper. An implementation of the framework is in progress, after which, we will use the framework to validate the results we have so far from our clone detector. Afterwards we will compare *SIMONE* with *ConQAT*. Also, we have developed multiple versions of *SIMONE*, thus we can use the framework to evaluate each version and to compare results.

IV. RELATED WORK

Other Simulink model mutation frameworks have been proposed by Zhan and Clark [9] and He et al. [10]. Unlike our framework, their motivation is testing and fault analysis of concrete Simulink models, and ensuring sufficient test-case coverage for the generated mutants. Their mutation operators modify the signal carried on wires between blocks. We are more interested in general mutations that modify the architectural structure of the model itself. In essence, our proposed framework attempts to mutate a Simulink model's design-time properties while, in contrast, the frameworks proposed by Zhan and Clark and He et al. mutate a model's run-time properties. That being said, sometimes a change in signal goes hand-in-hand with a change in structure, and vice versa. So, it may be interesting for us to apply some of their mutators in our work.

Roy and Cordy [3] proposed a mutation-based approach for source code clones. While the motivation is similar and we adapted the same idea, the mutation operators proposed for source-code clones were validated using "realistic" programmer edit scenarios that cannot be carried over to a model mutation framework. As such, we generated mutation operators based on our experience working with Simulink models and creating our Simulink model detector, *SIMONE*. We also added an additional clone report transformation step to our framework.

V. CONCLUSION

The challenges we encountered in our early attempts at comparing model-clone detectors made it very difficult to obtain any useful insights. There was also a lot of manual work required including our attempts to match resulting clones and clone classes from one tool to another and getting the representations to correspond in some meaningful way. In this paper, we presented a new approach for evaluating and comparing model-clone detectors that is based on mutation analysis and also clone representation transformation. Extending what is done in a source-code framework, we add model-appropriate mutators and add an extra step of normalization of the resulting clone classes and instances found. We have begun implementing this framework: the mutators are currently being developed and the transformation step is complete for the tools we are comparing currently. The advances provided by this framework address the challenges of a manual comparison and provide a standard and extendable way of evaluating and comparing model-clone detectors. By having this facility, the field of model-clone detection can continue to grow as new tools and approaches can be self-evaluated and compared to others.

REFERENCES

[1] M. Stephan, M. Alafi, A. Stevenson, and J. Cordy, "Towards qualitative comparison of simulink model clone detection approaches," in *IWSC*. IEEE, 2012, pp. 84–85.

[2] M. H. Alalfi, J. R. Cordy, T. R. Dean, M. Stephan, and A. Stevenson, "Models are code too: Near-miss clone detection for simulink models," in *ICSM*, vol. 12, 2012.

[3] C. Roy, J. 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.

[4] F. Deissenboeck, B. Hummel, E. Jurgens, B. Schatz, S. Wagner, J. Girard, and S. Teuchert, "Clone detection in automotive model-based development," in *ICSE*, 2009, pp. 603–612.

[5] A. Acree, T. Budd, R. DeMillo, R. Lipton, and F. Sayward, "Mutation analysis." DTIC Document, Tech. Rep., 1979.

[6] J. Bradbury, J. Cordy, and J. Dingel, "Mutation operators for concurrent Java (J2SE 5.0)," in *Second Workshop on Mutation Analysis*. IEEE, 2006.

[7] C. Roy and J. Cordy, "A mutation/injection-based automatic framework for evaluating code clone detection tools," in *International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2009, pp. 157–166.

[8] J. Cordy, "The TXL source transformation language," *Science of Computer Programming*, vol. 61, no. 3, pp. 190–210, 2006.

[9] Y. Zhan and J. Clark, "Search-based mutation testing for simulink models," in *Genetic and evolutionary computation conference*. ACM, 2005, pp. 1061–1068.

[10] N. He, P. Rümmer, and D. Kroening, "Test-case generation for embedded simulink via formal concept analysis," in *48th Design Automation Conference*. ACM, 2011, pp. 224–229.