Model Clone Detector Evaluation Using Mutation Analysis

Matthew Stephan

School of Computing, Queen's University, Kingston, Canada
matthew.stephan@queensu.ca

***Abstract*—Model Clone Detection is a growing area within the field of software model maintenance. New model clone detection techniques and tools for different types of models are being created, however, there is no clear way of objectively and quantitatively evaluating and comparing them. In this paper, we provide a synopsis of our work in devising and validating an evaluation framework that uses Mutation Analysis to provide such a facility. In order to demonstrate the framework's feasibility and also walk through its steps, we implement a framework implementation for evaluating Simulink model clone detectors. This includes a taxonomy of Simulink mutations, Simulink clone report transformations, and more. We outline how the framework calculates precision and recall, and do so on multiple Simulink model clone detectors. In addition, we also discuss areas of future work, including semantic clone mutations, and developing framework implementations for other model types, like UML. Lastly, we address some lessons we learned during the Ph.D. process; such as partitioning the work into logical, self-contained, milestones; and being open and willing to engage in other research. We hope that our framework will help cultivate further research gains in Model Clone Detection.**

I. Introduction

Model Driven Engineering (MDE) is becoming increasingly prevalent in the software engineering community, especially in embedded, communication, and automotive domains. As software projects are built using model-based paradigms and age and evolve, analysis of models becomes a crucial step in MDE's continued adoption and success. One type of analysis that has been valuable for both maintenance [1], [2] and evolution [3], is Model Clone Detection. Model Clone Detection involves identifying similar model fragments within a given context.

While the area of Model Clone Detection is experiencing quite a bit of growth, including new innovations and model types [4], one area that must be improved is the evaluation of tools and approaches [5]. Specifically, as we encountered during our early work evaluating a Simulink model clone detector [5], it is difficult to quantitatively compare tools or even refine a tool's configurable settings.

In this paper, we provide a synopsis of our work completed as part of a Ph.D. thesis [6] that contributes to the field of Model Clone Detection by providing a framework for quantitatively evaluating model clone detectors by using Mutation Analysis. This includes a description of the framework and a prototype of the framework we built to work with Simulink data-flow models. In addition, we outline future areas of research we believe are viable, and also impart some lessons we learned from our experiences.

Section II provides background and related work. Section III gives a high-level overview of our framework. Section IV discusses the Simulink mutations we created, while Section V presents steps we took in implementing the evaluation phase of framework for Simulink, the results of which are presented in Section VI. Sections VII, VIII, and IX present future work related to our research, lessons we learned along the way, and our conclusions, respectively.

II. Background and Related Work

In this section we provide background information on the material we will be covering in this paper. We also mention and compare work related to our research.

A. Model Clone Detection

Although a much newer area than its code-clone counterpart [7], there are notable Model Clone Detection advancements and approaches [4]. The most prevalent type of models analyzed are Simulink data-flow models, however techniques exist for general UML [8], State machines [9], and sequence diagrams [10]. It is generally accepted that there are three types of model clones [2]:

Type 1 - Exact Clones
: Type 1, or exact, clones are identical except for changes in position, colour, and other layout-related aspects.

Type 2 - Renamed Clones
: Type 2, or renamed, clones are the same allowing for variation in both model element names and values, and those properties of Type 1 clones.

Type 3 - Near-Miss Clones
: Type 3, or near-miss, clones allow for structural variation in addition to the differences accounted for in Type 1 and 2 clones. This may include additional or missing elements, changes in order, and other structural changes. In addition, near-miss clones are often associated with a similarity/difference threshold that indicates how much variation is allowed for a clone pair.

There exists Type 4, semantic, clones [11] that are structurally different but semantically equivalent. This is a newer idea, however, and has not yet been implemented in any model clone detector. We discuss it later in Section VII.

Two Simulink model clone detectors able to detect these types of clones are Simone [2] and ConQAT [12]. Simone preprocesses the underlying textual representations of Simulink models and detects near-miss Simulink clones up to a 30% difference threshold. ConQAT is a graph-based approach that flattens the Simulink system hierarchy and normalizes blocks into meaningful labels. ConQAT uses heuristics to find the largest common sub-graphs within Simulink models. There are a few other Simulink model clone detectors [13], [14] however they are less mature and unavailable.

B. Simulink

Simulink models include three levels of granularity: whole models, (sub) systems, and blocks. Blocks are the basic elements in Simulink and have a type and associated parameters that represent their semantics. They come from libraries and are connected to other blocks via lines that represent signals. Blocks are contained in systems and (sub) systems are contained in model files or in other systems. Engineers modify Simulink models using the Matlab environment. The underlying textual representation of the models is in MDL format, or XML format in the case of the latest version of Simulink.

C. Mutation Analysis

Mutation Analysis entails evaluating software by modifying, or mutating, artifacts and seeing how the software responds to these changes [15]. Mutation operators are used to illustrate an important system property or emulate specific future modifications to a system. Most work in Mutation Analysis thus far modifies code to test various aspects of a system. Often this includes evaluating the completeness of test suites by injecting potential errors and observing how a test suite covers them.

1) Model Mutation: Model Mutation Analysis is a new sub-area within Mutation Analysis and involves model mutations on model-driven systems. This includes state chart mutations [16] and agent-based models [17]. There is some work on Simulink Model Mutations [18], [19], [20] that describes mutations intended to mutate a model's run-time properties. These mutations are focused on mutating the signal carried between blocks on the wires rather than the structure of a Simulink system. Our mutations are structural in nature because model clones, thus far, relate to model structure rather than semantics/signals. However, their mutations can be classified using the taxonomy we describe later in this paper.

D. Code-Clone Detection Framework

Roy and Cordy [21] proposed a mutation-based approach for comparing and evaluating source code clone detectors, which they recently implemented [22]. We adapt the same general idea for our work, however, the mutation operators proposed for source code clones are based on code edits that cannot be converted to model mutations. For example, things like white space, comment changes, changes within program lines, and others do not have a direct translation to the modeling domain. In addition, they were not faced with the model-/domain- specific challenges that we identify in Section III.

III. Overview of Framework

We originally presented our framework in the New Ideas and Emerging Results track at ICSE [23] and further developed it in the thesis [6].

There are three main challenges to be addressed by the framework [23]:

Determining Recall
: Ascertaining if all the clones that should be detected are detected.

Nested Clones
: Tools may report only larger containing clones rather than smaller, more identical, nested clones. Comparing tools that report nested clones differently is an issue.

Clone Report Format
: Model clone detectors return their results in various forms. In order to analyze and compare tools, a relatively consistent clone report form must be obtained.

In this paper, we discuss the framework at a high level as laid out in Figure 1. The specific details of how the three challenges are addressed in the thesis [6] and our previous work [23]. The first two stages comprise the Mutation phase. The first step involves selecting the systems or models to mutate, which can be performed manually or randomly. For our Simulink implementation, we randomly selected a number of systems given a specific model. Each system is copied so it can be mutated and analyzed as a potential clone pair. For our framework, we decided that it made the most sense to duplicate the systems rather than inject the mutants directly for two reasons: 1) Including the entire higher-level context exacerbates the nested clone problem, and 2) injection into a duplicated higher-level context does not make sense in a modelling language such as Simulink because systems must be connected. Also, copying and modifying a system is a much more natural, copy-and-paste-like, operation, which mimics actual model maintenance. The second step in the framework involves mutating the systems. This can be done in different ways, depending on the types of models being mutated. Ideally, the mutations should be random in both the elements they mutate and how they mutate. In our implementation, we run each applicable mutation on each system being mutated.

The latter four chevrons in Figure 1 represent the Evaluation phase. Each model clone detector and configuration that one wants to evaluate must be run by the user on the systems that have been mutated. This will result in clone reports that likely will have to be transformed into a form conducive for tool evaluation. This can be done through any number of transformation methods, such as the one we used, TXL [24]. These transformed reports can be analyzed to assess recall and precision, as we do for Simulink and discuss in Section V. The last step in the framework involves presenting the results to the user by showing both an overall view of the results, and more detailed results for specific systems, including how well each tool performed on those systems.

IV. Simulink Mutations Developed for our Prototype

In this section, we summarize the Simulink mutations we created and employed in developing our Simulink implementation of the framework. We introduced, published, and validated the Simulink mutation classes for injecting clones in the International Workshop on Mutation Analysis [25]. Validation

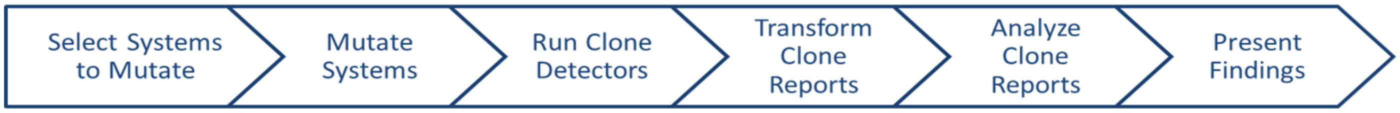

Fig. 1. High Level Overview of Framework Process

TABLE I. SIMULINK MUTATION CLASSES [25]

Mutation Key	Title	Clone Type
mMLA	Modification of Layout Attribute	Type 1
mRUE	Reordering Underlying Elements	
mRBL	Renaming a Block or Line	Type 2
mCBV	Changing a Block's Value	
mADBD	Add or Delete Block as Destination	Type 3
mADBS	Add or Delete Block as Source	
mCBT	Changing a Block's Type	
mCSCH	Changing a Subsystem's Clone Hierarchy	

was done through an evolution study. We then elaborated on the class definitions and implemented corresponding mutation operators in the thesis [6].

When creating our taxonomy of Simulink model mutations for clone injection we wanted mutation classes that 1) injected various types of clones and 2) were representative of how an engineer would modify a Simulink system. Based on our experience working with Simulink, writing a grammar for our clone detector [2], and our discussions with Simulink users, we came up with a classification. We present our classification once more in Table I, which contains the title and key of each mutation class and the type of model clone it injects. The full details and implementations of the classes, including why the clones injected are realistic and useful ones, can be found in our corresponding paper [25] and thesis [6].

As shown, each type of model clone was accounted for by the classes. In order to accomplish our second goal of having mutation classes that are reflective of real Simulink edit operations, we performed a model evolution study. Specifically, we looked at three Simulink projects, two open-source and one industrial, that had three or more versions each to see if all the model edits across versions could be classified using our taxonomy. Overall, the classes fit quite well with the observed model evolution, with the only exceptions being unconnected annotation or reference blocks. Details of this study can be found in our paper and thesis.

V. FRAMEWORK EVALUATION PROCESS

This section describes the specific steps we took in order to realize the Evaluation phase of the framework for our Simulink implementation of it. More detailed descriptions and examples can be found in the thesis [6].

A. *Persisting Mutant Metadata*

An important requirement of the framework is recording the kind and location of the mutations that have been injected into systems. In order to accomplish this in our Simulink implementation we record this information in XML files. One file is created for each System that is mutated and contains a single "original" element and one-to-many "mutant" elements. Both of these elements contain a subsystem attribute that identifies the full path to the original subsystem or the one that has been mutated, respectively. They both contain "block" elements that have a "path" attribute that correspond to the blocks within the systems and their respective paths. The schema for these files can be found in the thesis.

B. *Clone Report Transformation*

After model clone detectors are run on the mutated systems, the clone reports are transformed to a standard format so that they can be analyzed. In the case of our Simulink prototype, our target clone report form was extended from a format we came up with previously during a model clone evolution study [3]. At a high level, it is an XML file that contains "class" elements, representing clone classes, which contain "source" elements representing clone instances within each respective clone class. Each source element must contain "block" elements, with their fully qualified paths as an attribute, that correspond to the Simulink blocks within the clone instance.

The Simone clone detector's reports are made available in both HTML and XML format, with or without the model source, and with or without clone classes. The option closest to what we require for our framework is the XML format, with source, and with the clones sorted into classes. Thus, we wrote a TXL transformation to convert Simone reports into a report we could analyze and compare. This included extracting the full path to each block from the models, something which was not available in the original Simone reports.

ConQAT's Simulink model clone reports are in XML format and are made viewable through an HTML interface. ConQAT reports clone classes, termed "finding-group"s; and has each clone instance, or "finding", contain its respective Simulink blocks, which are represented as "qualified-name" elements with path attributes. This format was much closer to what we required for our framework, so we use a combination of the Unix stream editor and Perl commands rather than TXL.

C. *Recall and Precision Calculations*

We briefly summarize how we implemented recall and precision calculations through our framework. We omit some details here for brevity, which can be found in the thesis.

1) Recall: All the Simulink model clone tools we encountered provide clone classes, but not all explicitly identify clone pairs. So, for the Simulink framework implementation's recall calculation, we check if the original system and mutated system belong to the same class. This is consistent with how it is done in the code clone domain [21]. As formalized in Equation 1, where M is the mutant and OS refers to the original, unmutated, system; for each mutant metadata file, we iterate through all *mutant* elements and check for their coexistence with the SIS detailed in the *original* element in the transformed clone report. This is done by investigating each clone class reported by each specific tool or tool configuration.

We then calculate the total recall for a tool run by summing all the mutants detected for all the SIS and divide that by all the mutants injected, as shown formulaically in Equation 2. In this case, MI refers to the mutants injected via the framework implementation. We also explicitly list the mutants that were missed for each specific system.

$$Recall_{M,OS} = \begin{cases} 1, & \text{if M\&OS belong to same Clone Class;} \\ 0, & \text{Otherwise.} \end{cases} \tag{1}$$

$$Recall_{ToolRun} = \sum_{i=1}^{SIS} \frac{\sum_{j=1}^{MI(i)} Recall_{M(j),OS(i)}}{\sum MI} \tag{2}$$

2) Precision: Analogous to what is done in the code-clone domain, we form model clone pairs from the constituents within each model clone class and use that in the precision calculation as the denominator. In order to validate the clone pairs, we exploit the knowledge we have about the systems being mutated and how they are being mutated. So, using the block paths, we validate pairs that have only 30% or less difference in the identified blocks, as we previously established this as a reasonable difference threshold [2]. A key point to remember here is that our model clone pair validator is not a clone detector.

Once clone pairs have been validated, total precision can then be calculated as the summation of all valid clone pairs over all reported clone pairs across all systems, as demonstrated in Equation 3.

$$Precision_{ToolRun} = \sum_{i=1}^{SIS} \frac{CP(i)_{valid}}{CP(i)_{reported}} \tag{3}$$

VI. Simulink Framework Implementation Results

This section presents our experiments with our framework implementation for Simulink models. In this case we summarize our findings and leave the detailed examples in the thesis [6].

The mutation operators we implemented mutate randomly each time. For example, we mutate a random block within a system, add a block between two random blocks or as a destination from a random line, delete a random block, et cetera. As such, the models we select should not have too much impact on the evaluation of tools. However, it is important to see how the mutations effect different systems and systems of varying sizes.

Table II displays information about the models we mutate to demonstrate our prototype. The "Project" column indicates what project the model belongs to, while the "Model" column indicates what specific models were used. The PowerWindow (PW) project is the automotive demonstration model set that comes with Simulink. It has only one model, so we decided to use two different versions of that. The Advanced Vehicle Simulator (AVS) system is a large scale open-source Simulink project[1]. For the AVS project, we decided to go with two of the larger library models that seemed to contain rich systems. The "# of Systems Mutated" column indicates the number of randomly selected systems that we mutated. For our experiments, we had our program select a dozen or so systems from each model, if that many existed. We could have selected more, however the key number is the amount of mutations injected, which is listed in the "# of Mutations Injected" column. With all but the first version of the original and relatively small PowerWindow model, we ended up with roughly 150 random mutations injected from 14 different mutation operators for each model, for a total of 546 mutations.

[1]http://sourceforge.net/projects/adv-vehicle-sim/?source=dlp

TABLE II. Models Mutated by Prototype for Tool Evaluation

Project	Model	# Systems Mutated	# Mutations Injected
PW	PowerWindow(V1)	7	83
	PowerWindow(V3)	13	153
AVS	fc_KTH_lib	12	146
	lib_fuel_Cell	13	164

TABLE III. Results of ConQAT Evaluation

Model	Recall	Precision
PowerWindow(V1)	33%	99%
PowerWindow(V3)	23%	100%
fc_KTH_lib	34%	100%
lib_fuel_Cell	35%	89%

A. ConQAT Results

Table III illustrates the evaluation of ConQAT's handling of our mutated models. It is very important to mention that the way ConQAT is implemented at this time, it is only capable of detecting type 1 and some type 2 clones. It would be possible for them to eventually detect near-miss type 3 clones, as we discussed in our initial evaluation paper [2]. While we were aware of this early on, having this framework in place allows us to actually quantify it. Because roughly two thirds of our mutations are type 3, it is expected that ConQAT's recall will be about 33%.

B. Simone Results

1) Simone with Default Settings: Table IV presents the analysis of Simone's clone detection on the models from our experiments using the default settings. Seeing as Simone is intended to detect all three types of clones, one would hope that it has relatively high recall. A number from the table that immediately jumps out is the 77% precision in the lib_fuel_cell model. We notice that there are a number of systems that were randomly selected and mutated that had zero clones reported by Simone, thus bringing down the recall significantly. One of the tunable parameters within Simone is the minimum number of source lines that must comprise a system in order for it to be considered for clone detection. During our construction of Simone and early experimentation, we decided that smaller systems clones were likely trivial and not very prevalent in larger systems. Whether or not it is appropriate to configure the minimum number of lines to be lower for system clones, which is more of a semantic question for engineers, the key takeaway here is that the question arose because of metrics reported by our framework prototype. So, after executing our Simulink framework implementation on a specific set of models, engineers can decide if they want to configure Simone differently.

TABLE IV. Results of Simone Evaluation

Model	Recall	Precision
PowerWindow(V1)	98%	100%
PowerWindow(V3)	99%	95%
fc_KTH_lib	97%	94%
lib_fuel_Cell	77%	97%

TABLE V. Results of Simone Evaluation at 20% Difference

Model	Recall	Precision
PowerWindow(V1)	96%	100%
PowerWindow(V3)	96%	98%
fc_KTH_lib	94%	97%
lib_fuel_Cell	77%	98%

2) Simone with 20% Difference Threshold: While part of our motivation in developing this framework was to provide a facility to compare and contrast tools, we also wanted to allow for tool developers to refine their own tools. As such, we use our Simulink prototype to demonstrate this ability. We thought it would be interesting to adjust the main parameter, the near-miss difference threshold. Specifically, we restrict Simone by configuring it to accept only 20% differences among systems rather than the default 30%. The results are presented in Table V and can be directly contrasted with Table IV. In general, when something is more particular about what it takes in, there is likely to be a decrease in recall and increase in precision [26]. This holds true with our experiment in changing Simone's main parameter. Comparing the two tables, we see a total drop of eight recall percentage points across all four models and an increase of seven precision points.

VII. Future Work

In this section we discuss interesting areas of future work and research questions that arose from our work.

A. Semantic Clones

As alluded to in Section II, Type 4 semantic model clones for Simulink is a relatively unexplored area that could be incorporated into our framework once more work is done on it. Type 4 clones could be treated the same way as the rest of the clones types in that they can be injected and validated for recall and precision, respectively. A key difference however, is that we would have to first do a search on the model, or sets of models, to find a valid source model to mutate since not all the semantic preserving transformations would work on every system. This more aligns with the definition of a model transformation than mutation, so this would require a dual approach. It is definitely possible though, since even some of the mutations we implemented thus far have some constraints on the systems they operate on, for example, the deleting a block in between mutation, changing a block's value, and changing a block's type. However, our constraints are likely not as complex as those involved in Type 4 clones would be.

B. Other Types of Models

Aside from choosing Simulink because it was of interest to our industrial partners, we also focused on it because Simulink model clone detection was far and away the most mature type of model clone detection. There have been attempts at other model types, however there are not many tools for these model types, rather, most model types have a single approach at this time.

We attempted to make the framework as generic as possible. In terms of mutation, the same process would apply to any model type once an appropriate granularity was selected as the focus of the mutations. So, perhaps, for UML structural models, each class diagram could be considered a system. Multiple class diagrams would be duplicated, mutated, and organized similarly. The evaluation aspects would remain the same.

For coming up with mutations, each type of model would require clear and consistent definitions of each type of clone. In general, it is likely that all models would share some notion of exact, Type 1, clones and near-miss, Type 3, clones. Type 2 may not be as evident, as not all model elements have values, or identifying and/or unique names. Once clone types are realized by tool researchers, an initial mutation list can be devised to inject clones and validated by means of doing a similar evolution experiment to ensure the mutations are realistic and cover any cases found in available model sets.

The process we took for calculating precision and recall in our prototype can be employed for any type of model. The key, as it was for us, would be to have some clone report format that can allow an evaluation tool to determine fragment containment and clone-pair validity. So clone-report transformation may be required for other model types, as it was for us.

C. Refining Precision and Clone Pair Validation

Similar to what was done in the code-clone tool evaluation approach [21], we created a model clone validation process for model clone pairs. However, validation in the modeling domain was slightly more complex. Rather than just comparing text similarity of reported clones, we had to determine how to validate two Simulink systems. Considering the only consistent information reported across tools, albeit not always explicitly, was block paths, we had to use those. This works since we are aware of the systems we are working with and the nature of the mutations occurring. That being said, this validation technique is not perfect and is a weakness of the approach currently. It would be better if we could devise some heuristic that is not clone detection, but could traverse the models quickly to validate a clone pair thus using structure, including block type, rather than qualified block paths.

VIII. Lessons Learned

There are two main lessons we have based on our experiences.

One thing that worked very well for us during the Ph.D. process was that we were able to partition the overall goal of our research into logical, and self contained, deliverables. This was beneficial as we could validate and publish each step along the way, which gave us the confidence required to continue. So, for example, our background was published as survey papers, our overview was published in an early achievements track, and our mutation classes were published and validated in an apt venue. This is a valuable skill to work on and hone during the process.

Secondly, students should be open and willing to engage in other research that comes up as a side effect of their main research. This will help diversify their portfolio and may help lead their research in directions that were not even imagined before. In our case, we did some side work on the evolution of model clones that resulted in one paper [3] and others that we are working on currently. The key skill developed in doing this was being able to see beyond the immediate boundaries of our research.

IX. Conclusion

In order to advance the research in Model Clone Detection, there must be a way to quantitatively evaluate and compare model clone detectors. In this paper, we provided a synopsis of our research that attempts to achieve that through a framework that uses Mutation Analysis. After introducing the unique challenges that the framework must address, we provided a high-level overview of the framework's process including the mutation and evaluation phases. We describe our Simulink model mutations that inject all the various types of model clones and represent realistic Simulink evolution. Our implementation of the framework for Simulink involved persisting mutant metadata; transforming clone reports for two tools; and calculating recall and precision by analyzing clone classes and validating clone pairs, respectively. We run experiments on Simone and ConQAT using four models and 546 mutations, and discover that ConQAT kills roughly one third of the mutations while Simone kills almost all of them. The precision of both tools is quite high.

Future work involves accounting for semantic clones by using model transformations in the framework in conjunction with mutations, developing framework implementations for other types of models once model clone detection research advances for those types, and refining our precision/clone pair validation approach by using graph traversal instead of paths. The two lessons we have for future students are to split up their work into self contained, ideally publishable, components; and to be open and willing to engage in secondary research and projects that come up. With our framework in place, we hope that future model clone detector researches will use it to evaluate and refine their tools in hopes of improving the field as a whole.

Acknowledgment

Thank you to my supervisor, James R. Cordy, for all his assistance and advice. Thanks to Andrew Stevenson for all his help with the TXL transformations and model clone report transformations. This work is supported by NSERC, the Natural Sciences and Engineering Research Council of Canada, as part of the NECSIS Automotive Partnership with General Motors, IBM Canada, and Malina Software Corp.

References

[1] F. Deissenboeck, B. Hummel, E. Juergens, M. Pfaehler, and B. Schaetz, "Model clone detection in practice," in *International Workshop on Software Clones (IWSC)*, 2010, pp. 57–64.

[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*, 2012, pp. 295–304.

[3] M. Stephan, M. H. Alalfi, J. R. Cordy, and A. Stevenson, "Evolution of model clones in simulink," in *Models 2013 - Models and Evolution*, 2013, pp. 38–47.

[4] M. Stephan and J. R. Cordy, "A survey of model comparison approaches and applications," in *MODELSWARD*, 2013.

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

[6] M. Stephan, "A mutation analysis based model clone detector evaluation framework," Ph.D. dissertation, Queen's University, 2014.

[7] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," Queen's University, Tech. Rep. 2007-541, 2007.

[8] H. Storrle, "Towards clone detection in uml domain models," in *European Conference on Software Architecture (ECSA): Companion Volume*, 2010, pp. 285–293.

[9] J. Chen, T. Dean, and M. H. Alalfi, "Clone detection in matlab stateflow models," in *International Workshop of Software Clones*, 2014, pp. 1–10, to appear.

[10] E. P. Antony, M. H. Alalfi, and J. R. Cordy, "An approach to clone detection in behavioural models," in *Working Conference on Reverse Engineering 2013*, 2013, pp. 472–476.

[11] B. Al-Batran, B. Schatz, and B. Hummel, "Semantic clone detection for model-based development of embedded systems," *Model Driven Engineering Languages and Systems*, pp. 258–272, 2011.

[12] F. Deissenboeck, B. Hummel, E. Juergens, B. Schaetz, S. Wagner, J.-F. Girard, and S. Teuchart, "Clone detection in automotive model-based development," in *ICSE*, 2009, pp. 603–612.

[13] N. Pham, H. Nguyen, T. Nguyen, J. Al-Kofahi, and T. Nguyen, "Complete and accurate clone detection in graph-based models," in *International Conference on Software Engineering (ICSE)*, 2009, pp. 276–286.

[14] H. Petersen, "Clone detection in Matlab Simulink models," Master's thesis, Technical University of Denmark, 2012, iMM-M. Sc.-2012-02, 2012.

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

[16] M. Trakhtenbrot, "Implementation-oriented mutation testing of statechart models," in *International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)*, 2010, pp. 120–125.

[17] S. F. Adra and P. McMinn, "Mutation operators for agent-based models," in *International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)*, 2010, pp. 151–156.

[18] Y. Zhan and J. Clark, "Search-based mutation testing for Simulink models," in *Genetic and Evolutionary Computation Conference*, 2005, pp. 1061–1068.

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

[20] R. F. Araujo, A. M. R. Vincenzi, F. Delebecque, J. C. Maldonado, and M. E. Delamaro, "Devising mutant operators for dynamic systems models by applying the HAZOP study," in *ICSEA 2011*, 2011, pp. 58–64.

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

[22] J. Svajlenko, C. K. Roy, and J. R. Cordy, "A mutation analysis based benchmarking framework for clone detectors," in *International Workshop on Software Clones (IWSC)*, 2013, pp. 8–9.

[23] M. Stephan, M. Alafi, A. Stevenson, and J. Cordy, "Using mutation analysis for a model-clone detector comparison framework," in *ICSE*, 2013, pp. 1277–1280.

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

[25] M. Stephan, M. Alalfi, and J. R. Cordy, "Towards a taxonomy for simulink model mutations," in *International Conference on Software Testing, Verification, and Validation 2014 (ICST) – Mutation Workshop*, 2014, pp. 206–215.

[26] M. K. Buckland and F. C. Gey, "The relationship between recall and precision," *JASIS*, vol. 45, no. 1, pp. 12–19, 1994.