Received <day> <Month>, <year>; | Revised 7 February, 2018; | Accepted <day> <Month>, <year>

DOI: xxx/xxxx

SPECIAL ISSUE PAPER

MuMonDE : A Framework for Evaluating Model Clone Detectors using Model Mutation Analysis

Matthew Stephan*[1] | James R. Cordy[2]

[1]Department of Computer Science and Software Engineering, Miami University, Oxford, Ohio, USA
[2]School of Computing, Queen's University, Kingston, Ontario, Canada

Correspondence
*Matthew Stephan, 205 Benton Hall, 510 E. High St., Miami University, Oxford, Ohio, USA, 45056. Email: matthew.stephan@miamioh.edu

Abstract

Model-driven engineering is an increasingly prevalent approach in software engineering where models are the primary artifacts throughout a project's life cycle. A growing form of analysis and quality assurance in these projects is model clone detection, which identifies similar model elements. As model clone detection research and tools emerge, methods must be established to assess model clone detectors and techniques. In this paper, we describe the MuMonDE framework, which researchers and practitioners can employ to evaluate model clone detectors using mutation analysis on the models each detector is geared towards. MuMonDE applies mutation testing in a novel way by randomly mutating model elements within existing projects to emulate various types of clones that can exist within that domain. It consists of two main phases. The Mutation Phase involves determining the mutation targets, selecting the appropriate mutation operations, and injecting mutants. The second phase, Evaluation, involves detecting model clones, preprocessing clone reports, analyzing those reports to calculate recall and precision, and visualizing the data. We introduce MuMonDE by describing each phase in detail. We present our experiences and examples in successfully developing a MuMonDE implementation capable of evaluating Simulink model clone detectors. We validate MuMonDE by demonstrating its ability to answer evaluation questions and provide insights based on the data it generates. With this research employing mutation analysis, our goal is to improve model clone detection and its analytical capabilities thus improving model-driven engineering as a whole.

KEYWORDS:
mutation analysis; model-driven engineering; model clone detection; model clone detectors; mutation testing; tool evaluation

1 | INTRODUCTION

Mutation analysis in software engineering involves system analysts artificially modifying software systems and observing the systems' reactions to those changes [1]. Mutation testing applies mutation analysis for the purposes of testing a software system and measuring the quality of its test suite [2]. Traditionally, mutation testing is used on source code developed using third generation programming languages [3].

In many domains, the primary artifacts in the software engineering life cycle are not in the form of source code but rather higher-level abstract notations known as models [4]. This type of software construction, termed model-driven engineering [5], is experiencing wide adoption in embedded systems applications such as the telecommunications, automotive, and aerospace domains. Mutation analysis and testing concepts have been applied to models. For example, state chart validation [6], model transformation testing [7], and assessment of finite state machines [8]. However, mutation testing for models is still in its infancy compared to traditional mutation testing.

As model driven engineering's popularity increases [9], model quality assurance is becoming a key focus of the field [10]. This includes research on improving model driven engineering processes and products [11, 12]. One example of this is model clone detection, which provides engineers the ability to detect similar models and elements in modelling projects [13]. Model clone detection has a variety of uses including library extraction, model maintenance [14], and quality assurance [15, 16, 17]. Because there are many different modelling abstractions, research in model clone detection is evolving quickly and is yielding many different tools and techniques targeting different model types.

As more model clone detectors are created, the need for model clone detector evaluation becomes imperative. Existing techniques for evaluating clone detectors are insufficient as they target source code clones and are unable to handle model-specific concerns. To address this need, we propose a model-specific model clone detector evaluation process employing mutation testing: The MUtation analysis MOdel cloNe Detector Evaluation (MuMonDE) Framework. Based on our experiences successfully evaluating Simulink model clone detectors using this process, we generalize it to assist others in applying it to other software model types. This includes a step-wise description of MuMonDE along with examples from our successful application of it. Although the process has not been implemented for other model types, we include insights and suggestions on how it can be for the Unified Modelling Language (UML) and others.

1.1 | Contributions

This paper extends our previous work on developing a means for evaluating Simulink model clone detectors using mutation. Previous work includes,

1. Our first small position paper proposing the idea [18].
2. A taxonomy for classifying Simulink model mutations [19].
3. Our short doctoral symposium paper briefly summarizing our results analyzing Simulink model clone detectors, and presenting an overview of our takeaways from the experience [20].
4. A Ph.D. thesis describing our evaluation of our Simulink model clone detector, Simone [21], and others using model mutation [22].

This article builds upon that work by making the following novel contributions,

1. A generalized mutation analysis testing framework process, MuMonDE, which researchers and practitioners can follow to assess the quality of their model clone detection tools, and compare different tools and configurations.
2. A detailed and technical step-by-step process analysts can follow to develop assessment tools to evaluate model clone detectors and techniques. This is supported by a presentation of our experiences implementing MuMonDE for Simulink model clone detector evaluation, which we update and summarize from our past work.
3. A discussion of how MuMonDE can be applied to other model types, such as UML, threaded throughout the paper at each respective phase.
4. Validation of MuMonDE by assessing its engineering value. We demonstrate its ability to provide answers to evaluation questions we had while developing a model clone detector. This includes a presentation of our results from evaluating multiple Simulink model clone detectors and configurations.

The paper begins with background information on model clone detection and model mutations in Section 2. We discuss related clone detector evaluation research in Section 3. We describe the MuMonDE framework in Section 4, including a presentation of the two phases: Mutation and Evaluation. We validate MuMonDE in Section 5, and discuss future work and our conclusions in Section 6 and 7, respectively

2 | BACKGROUND

We begin with background information on model clone detection and model mutations, as these are fundamental aspects of MuMonDE.

2.1 | Model Clone Detection

As model-driven projects age they may exhibit properties similar to third-generation programming language projects. One such property is the emergence of clones, which are similar code fragments within a project. A code clone pair, or "clone", refers to two code portions that are similar according to some measure of similarity [23]. Clones can exist in software projects when the implementation of a similar concept is used in multiple places. They may be introduced because of poor reuse practices, time constraints, lack of knowledge about cloning, and other reasons [23]. A problem with clones is changing one concept may require updates in multiple places, which can be neglected or difficult. However, clones can also have a positive impact on software maintainability [24]. Regardless of how they are perceived, clone identification is important and its research is quite mature as demonstrated by the many techniques and tools [25].

The analogous problem of model clones refers to models that are similar according to some similarity definition. To find model clones, analysts must employ some form of model comparison to identify similarities and differences [26]. There are many different model comparison approaches and algorithms that work on a variety of model types [27, 28]. One algorithm for model comparison is similarity-based matching, which can be used to realize model clone detection. Due to the nature of models, techniques intended for code clones are not well suited to model clones [13]. For example, model clone detection is NP-complete because it is a largest common sub-graph problem [29], which involves looking for common sub-graphs in a single graph. Additional model clone identification challenges are described by both us [18] and Gold et al. [30]. In comparison to code clone detection, the research related to model clone detection is quite novel and limited. Model clones can be clustered and classified into groupings known as model clone classes, which characterize a repeated modelling pattern [31]. The benefits and drawbacks of cloning in models is an interesting discussion, but out of this work's scope as we focus solely on model clone detectors.

There are different model clone detection techniques that work with a variety of model types. The most mature model type considered for clone detection is Simulink [13, 21, 32, 33]. Simulink is a modelling language that describes data-flow models. Simulink models contain systems, which contain blocks and lines. Systems can also contain other (sub)systems. Newer techniques for other model languages exist including UML models [34, 35], and Stateflow [36, 37]. Gold et al. [38] devised a method for finding clones in Max/MSP patches, which are expressed graphically. Similar to Simulink and UML, a Max/MSP patch has boxes, representing operations and messages, and lines representing data flow.

2.1.1 | Model Clone Types

We define model clones types [21] consistently with model clone detection research. This is necessary as detector evaluation should account for clone type to provide better insights.

Type 1 - Identical Clones

Sets of model elements that are identical, ignoring layout, formatting, and any changes in visual aspects.

Type 2 - Renamed Clones

Sets of model elements that fit the description of Type 1 but allow for differing names and values.

Type 3 - Near-Miss Clones

Sets of model elements that are structurally different up to a specific threshold and can exhibit the same differences addressed by Type 1 and Type 2 clones. Examples of structural changes include different element ordering, elements that have been added or deleted, and more. Figure 1 presents a simple example of a model clone from the Simulink demonstration set of the Throttle Estimation and Speed Estimation systems. These systems are very similar. Differences include a new Simulink block and line added to the system on the right, and various naming and attribute changes to other blocks and lines.

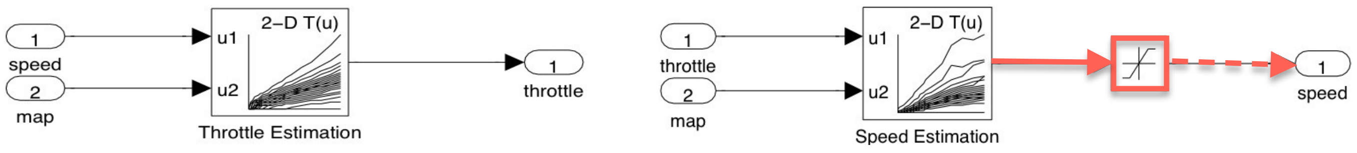


FIGURE 1 Example Type 3 Model Clone [22]

Type 4 - Semantic Clones

Sets of model elements that are significantly structurally different but are semantically equivalent [39]. They differ beyond a structural similarity threshold, thus are not classified as Types 1-3, but may have the same underlying behaviour. The notion of Type 4 clones is more novel than Types 1-3. Type 4 clones can be more difficult to detect, as direct syntactic comparison alone is not enough to identify clones.

We present model clone detection approaches to provide the reader with context on different approaches, and because we reference some of them later in the paper.

2.1.2 | Graph-Matching Model Clone Detection

CloneDetective's ConQAT [13] is the most mature approach. Drawing from ideas in graph theory, it works on Simulink models. It flattens all models and removes any unconnected lines. ConQAT assigns Simulink blocks and lines a label consisting of information considered important for comparison. The information assigned by ConQAT changes according to the type of blocks being searched for by the tool. For example, ConQAT labels describe the type of Simulink block, ignore block parameters, and consider block specific elements, such a gain block's multiplication factor. ConQAT identifies clones by iterating through node pairings using a breadth-first search. They cluster their clones based on the set of nodes identified in the clone pairs.

eScan and aScan are other graph-matching algorithms for detecting exact-matched and approximate clones, respectively [40]. Like ConQAT, exact-matched clones are pairs of model elements having the same size and aggregated labels, but in this case, the labels contain topological information. Approximate clones allow for differences in labels. aScan is able to detect approximate clones while ConQAT is not. Neither eScan or aScan are available for use nor supported anymore. Thus we were not able to test them using our Simulink MuMonDE implementation.

Peterson [33] has developed the Naive Clone Detector to detect exact Simulink clones. Like ConQAT, it uses graph-based modelling and Simulink information, but by contrast, it employs a top-down approach. It is a proprietary tool.

2.1.3 | Text-based Model Clone Detection

Simone detects near-miss clones in Simulink models by adapting existing code-clone techniques to work with textual representations of Simulink models while still being model sensitive [21]. It is an extension to the NICAD parser-based code clone detector [41] and uses a TXL [42] grammar for Simulink. This grammar accounts for all Simulink constructs, including models, systems, blocks, lines, ports, branches, and other components. It identifies model clones at the system level of granularity, as systems are the dominant unit of organization within Simulink. It identifies Type 1, 2, and 3 clones. Simone extensions include model clone detection tools for UML behavioral models [34] and Simulink Stateflow models [43].

Störrle [35] developed MQlone to detect model clones. They convert XMI files from UML CASE models and turn them into Prolog[1]. Once in Prolog, MQlone attempts to discover clones using static identifier matching combined with similarity metrics. Some of these metrics include size, containment relationships, and name similarity. They note that it is restricted to UML models, still has room for improvement, and is not validated by field studies.

2.2 | Model Mutation

We provide brief background on model mutation as it is a key aspect of MuMonDE. The general idea of model mutation is to apply mutation analysis to high-level software artifacts. Model mutations can modify both the syntax and semantics of models for a variety of purposes. The mutations must mutate in a way that does not invalidate the models' meta-model, which specifies

[1] www.swi-prolog.org

the modelling language and valid forms of the models [44]. Model mutations can involve changing connections, modifying elements' attributes, and adding or deleting elements. A simple example mutation might be changing the model on the left of Figure 1 by renaming "Throttle Estimation" to "Speed Estimation", as shown in the model on the right of the figure.

Sen and Baudry developed model mutations using graph grammars inferred from meta models to automatically synthesize models and their variations for development [44]. Tran et al. devised model mutations to facilitate Simulink refactoring [45]. They employ mutations that delete, copy, replace, and add model elements, and combine them for refactoring purposes.

Model mutations for the purpose of mutation testing includes work by Trakhtenbrot [46] for state chart mutations. Agent-based mutation operators were developed by Adra and McMinn for testing models in that domain [47]. Bartel et al. evaluate adaptive systems using model-driven mutations [48]. There is also the Simulink mutation taxonomy [19] that we devised for our Simulink MuMonDE implementation, which we discuss later.

Model mutations have also been employed for behavioral models through mutation and observation of systems' dynamic and run-time characteristics [49, 50, 51]. These mutation operators modify the signals between model elements. Zhan and Clark explicitly categorize their signal mutations into assignment, multiplication, and addition, representative of those respective operations on signals.

3 | RELATED WORK: EXISTING CLONE DETECTOR EVALUATION RESEARCH

Early experiments evaluating code clone detectors were completed by Bellon et al. [52]. They evaluated six code clone detectors on eight programs by using a human oracle to analyze clone candidates submitted from each detectors. Similar to MuMonDE, they inject clone pairs. However, Bellon's clone pairs are based on curated clones. Similarly, Schulze and Meyer [53] developed a framework to evaluate robustness of code clone detectors against code obfuscations. It considers various code modifications, such as loop and conditional transformations. It focuses on Type 4 clones. Ragkhitwetsagul et al. [54] also compare code clone detectors' abilities to detect obfuscation and source code normalization by considering Java source code scenarios. These methods differ from MuMonDE as they are intended strictly for code and consider clones based on code-specific modifications. The latter two approaches do not inject clones.

After Roy and Cordy qualitatively evaluated code clone detectors [25], they devised a technique for assessing source code clone detection tools [55] utilizing mutation analysis. This technique was developed into a tool by Svajlenko and Roy [56], which they later tested [57] using a benchmark of intra- and inter- project code clones [58]. This included code-specific mutations related to white space, string and numerical literals, comments, variables, and others. While MuMonDE's motivation is similar, Svajlenko and Roy's mutation operators were validated using realistic programmer code editing scenarios that are not applicable for a model mutation framework. Many of the notions, such as comments, changes inside of code lines, and white space do not have any direct correlation to models and modelling languages. Their approach is tailored specifically to code, and does not have to address the challenges unique to modelling that we discuss in this paper.

A related Eclipse Tool, developed by Uhrig and Schwagerl [59], evaluates matching algorithms for Eclipse Modelling Framework (EMF) models by testing against a benchmark. They combine both user involvement and automated testing, including a match model that formalizes matches. While there are key differences between model comparison and clones [26], it is possible that some of their work can be leveraged in MuMonDE by determining if their matching evaluation algorithms are applicable.

4 | THE MUMONDE FRAMEWORK

Consistent with many mutation analysis approaches [60], MuMonDE first performs mutations on existing systems, and then observes and evaluates how those mutations are handled by the mechanisms under study. Thus, we break MuMonDE into two main phases: the Mutation Phase and the Evaluation Phase. Each phase contains its own steps as illustrated in Figure 2 . The Mutation Phase has 3 sub steps, while the Evaluation Phase has 4 sub steps. In our approach, the mutations injected by MuMonDE implementations will be model mutations emulating the various types of model clones a detector is expected to find. MuMonDE evaluation entails scrutinizing the results of model clone detection by the respective tools, or tool configurations, on these mutated models.

There are three main problems MuMonDE addresses. The latter two are unique to model clone detection evaluation. These problems are defined in greater detail in our previous work [18].

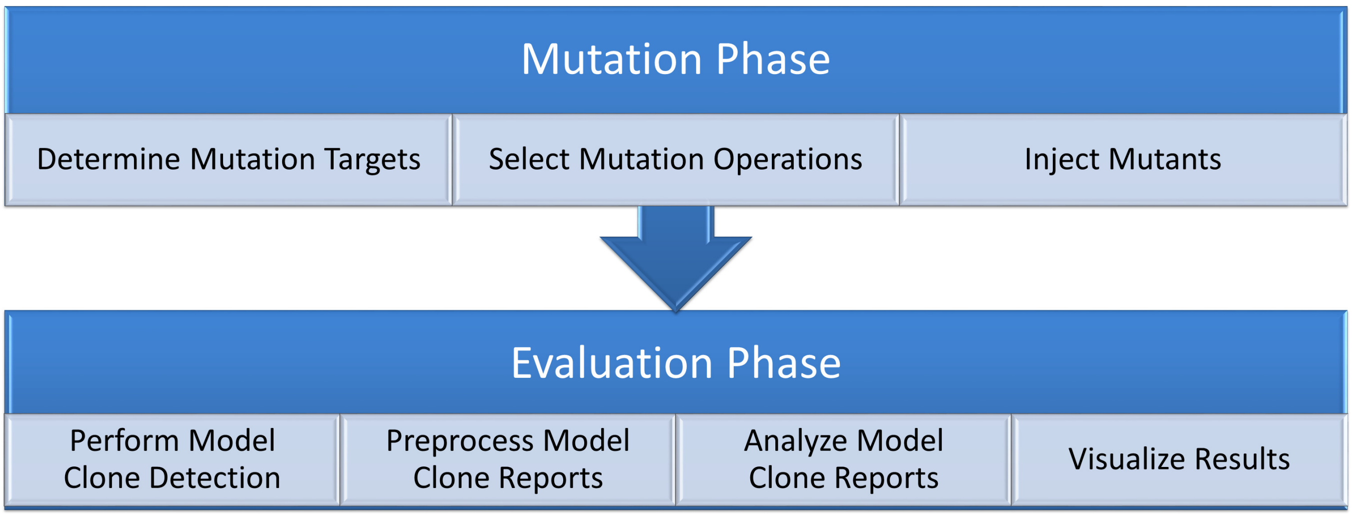

Mutation Phase
Determine Mutation Targets
Select Mutation Operations
Inject Mutants
Evaluation Phase
Perform Model Clone Detection
Preprocess Model Clone Reports
Analyze Model Clone Reports
Visualize Results

FIGURE 2 The MuMonDE Framework Phases and Steps

Automatic Recall Calculation

Are the model clones that should be detected actually detected by the tools being evaluated? Can this question be answered automatically, with reports and details provided to analysts?

Inner Clones

The hierarchical nature of models may cause detectors to identify only outer (larger) clones rather than inner, yet more similar, clones. If different tools handle nesting differently, there must be normalization to compare these tools.

Differences in Model Clone Report Representation

For code clones, all results can be in the form of source code or line numbers. For model clone detectors, it is plausible that analysts will encounter a variety of output formats, such as XML, lists of model elements, textual representations, or others. In order to compare approaches, there must be a process to transform the results to a common representation.

We present each of MuMonDE's generalized phases and steps in this section, supported by examples from our successful Simulink MuMonDE implementation. We include our insights and examples of how the MuMonDE process can be implemented for other model types.

4.1 | First Phase: Mutation

The first phase of MuMonDE involves mutating models and is summarized in Figure 3 . This figure uses systems as an example unit of granularity, since we considered Simulink (sub)system model clones in our MuMonDE implementation. Simulink model clone detectors normally detect Simulink (sub)system level clones, and Simulink (sub)systems are the units identified as most important to our industrial partners. However, systems can be replaced by whatever logical unit makes sense for each respective modelling domain. MuMonDE begins, as shown on the left of the figure, by selecting which X number of systems will have their model elements mutated. This is followed by having Y mutation operations, with a user-specified, Z, number of variants injected on copies of each selected system. Lastly, MuMonDE dictates placing the mutated systems in their respective folders, which are showcased in the right of the figure. We elaborate on these steps in the subsections that follow.

4.1.1 | Determine Mutation Targets

The first sub-step in Phase one of MuMonDE is selecting which specific elements to use as the original source model elements. This can be done at different granularity and either manually, semi-randomly, or randomly. For our Simulink MuMonDE implementation, for example, we manually select a Simulink model folder to use as the base. This is represented by the filled folder in the left of in Figure 3 . Our implementation then randomly selects a user-specified and configurable number of Simulink

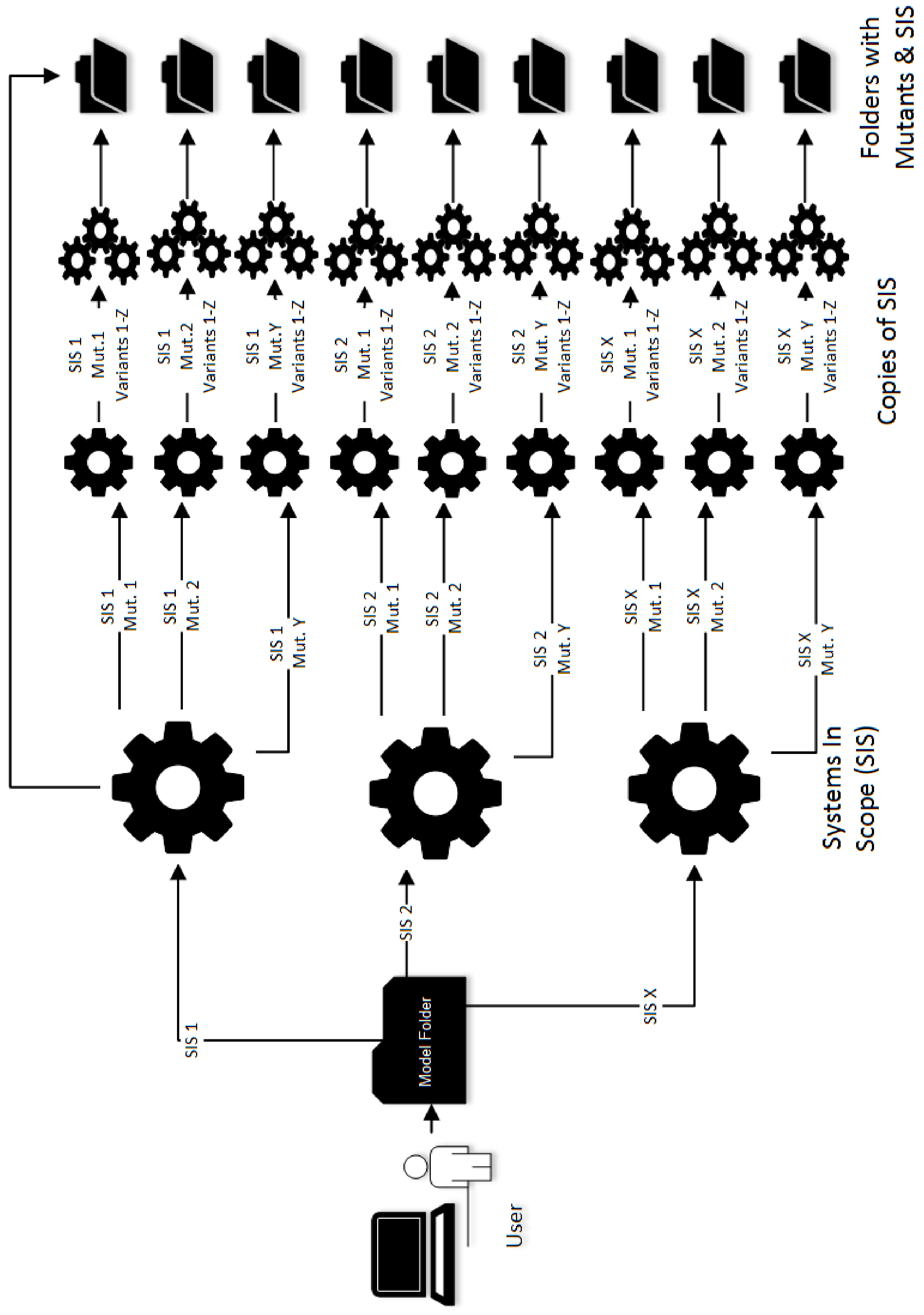


FIGURE 3 MuMonDE's Mutation Phase with X systems, Y mutations, and Z variants

TABLE 1 Simulink Mutation Taxonomy [19]

Mutation Class	Clone Type
Modification of Layout Attribute	Type 1
Reordering Underlying Elements	
Renaming a Block or Line	Type 2
Changing a Block's Value	
Add or Delete Block as Destination	Type 3
Add or Delete Block as Source	
Changing a Block's Type	
Changing a Subsystem's clone hierarchy	

systems within that single model to mutate. That is, our implementation employs a Simulink function that randomly selects X number of Simulink (sub)systems, often referred to simply as systems, where X can be configured as a parameter to the method. The impact of X on the run time of the model clone detectors with increasing numbers of systems depends on each detector's speed and scalability.

The X systems to mutate are represented on the left in Figure 3 , and we herein refer to these elements as systems in scope (SIS). A notable difference between MuMonDE and the code-based framework is rather than inject clones back into the original source and run the comparison [55], we 1) identify and duplicate the specific SIS, 2) mutate it, and 3) inject each mutated system as a standalone system in its own model file with a single mutation. There are two reasons for this. Firstly, copying the entire containing model and replacing the mutated system would fall victim to the nested clone problem [21]. That is, higher-level containing systems, or the model itself, would be identified as clone pairs, and it would be difficult to discern if the mutated SIS was detected. Secondly, injection back into a duplicated copy of the containing model but not in place of the original and pre-mutated system is unnatural from a modelling perspective. This is because all system components are connected to the rest of model, so it is unclear where a mutated system belongs. While there can be unconnected components, such as ones that we discovered in a small number of Simulink systems, they are typically syntactically and semantically trivial reference or annotation blocks. Duplicating the SIS satisfies our requirements for evaluating model clone detectors, while avoiding the nested clone problem. Additionally, it mimics the notion of a modeler "copy and pasting", and optionally modifying, a system from one place to another, which is a common scenario [13].

4.1.2 | Select Mutation Operations

After MuMonDE determines which systems to mutate, its next step involves determining which types of mutation operations to employ. The goal of this MuMonDE step is to have a set of operations producing variations of all three or four model clone types. For our Simulink MuMonDE implementation, we based our mutation operations on the taxonomy we defined empirically through an investigation of many successive versions of both industrial and open source systems [19]. We present that taxonomy in Table 1 . It includes a list of the mutation classes in the taxonomy and the resulting model clone type a mutation from that class will generate. The specific Simulink mutation operations we used in our implementation can be found in our previous work [22] and downloaded[2]. While they are Simulink specific, they provide a good basis for other modelling languages, especially data-flow languages. We now discuss the general requirements for selecting mutation operations for a MuMonDE implementation, organized by model clone type.

Type 1 Model Clones

Type 1 model clone mutation operations must generate systems that are structurally identical, but differ in visual presentation aspects. Ideally, there should be one mutation operation for each potential visual difference, allowing for variation. For example, if model elements have some notion of color, it should be possible to randomize the color that is selected. When devising

[2] http://www.users.miamioh.edu/stephamd/projects/mumonde/

our Simulink mutation operations, we ensured that we had mutation operations accounting for position, size, color, and other Simulink specific layout attributes [19].

Type 2 Model Clones

Type 2 model clone mutation operations will also produce structurally identical clones, but must rename elements and change elements' values. There are multiple renaming strategies in devising mutations including variations of Levenshtein distance [61], Hamming distance [62], or plain randomization. For mutating element values, the operations should change every value associated with an element both independently and in groups. For example, if an element has four values associated with it, the mutation operations should change each of those four values by themselves. Additionally, the operations should change combinations of values, such as the first and second, first and third, et cetera. When changing values, it is important to consider if the values are part of the identifiers used by a model clone detection tool.

Type 3 Model Clones

The mutation operations for creating near-miss clones are more complex because there are many ways that models can change. These mutation operations must encompass a mixture of adding, deleting, modifying, and replacing elements. Our suggestion to MuMonDE implementers is to ascertain what a sufficient and "realistic" set of Type 3 mutation operations entails by performing domain analysis focusing on model evolution. We recommend performing this as a first step. Additionally, another route is to formally determine a set of sufficient, and possibly complete, mutation operations using the meta model of the specific language. That is, if the modelling language allows for only a specific set of operations and that information can be explicated in the form of mutations, then that can be leveraged. This is similar to the notion of automatically generating model transformations from meta models [63].

For our Simulink implementation, we investigated a number of public and private Simulink projects and observed how models were modified across versions [19]. Specifically, every time a model element changed from one version to the next, we determined exactly what the change was, categorized it, and tallied it. This allowed us to better claim that the mutations used in our Simulink MuMonDE implementation were representative of, and consistent with, model clones in real life.

Type 4 Model Clones

There is very little research we found on semantic model clones. However, it is still possible for MuMonDE implementations to account for them. One approach is to consider all semantically-equivalent transformations as mutation operations. These transformations can be acquired through various derivation techniques. An example of this is Al-Batran's forty semantically preserving rules for Simulink derived using 1) mathematical, 2) logical, and 3) structural properties [39]. Rules such as these can be converted into mutation operations to create Type 4 model clone instances. The only reason we excluded Type 4 model clones in our Simulink MuMonDE implementation is because only Al-Batran's approach is capable of detecting Type 4 model clones.

4.1.3 | Inject Mutants

Once the mutation operations are determined by a MuMonDE implementer each selected SIS can undergo mutation. While MuMonDE can inject any combination or sequence of mutations, we suggest executing each operator on each selected and eligible SIS. In terms of eligibility, there may be examples where some models can not be mutated in a specific way, as we discussed in our previous work on Simulink model mutations [19]. The computational requirement in performing mutations is trivial in most cases, even in large numbers. For example, in our experience applying all the different mutation operations to a dozen or so systems in even the largest models we describe in our evaluation took less than 5 minutes. This included models containing upwards of 83 systems resulting in approximately 150 mutations. This is without any notable effort to optimize our mutation scripts.

Thus, we see no reason, other than eligibility, to not have each mutation operator applied to each SIS. That is, using X systems to be mutated (SIS), and Y mutation operators, then there will be at least X*Y mutations performed. Each mutation operator should be applied on the original SIS in isolation. This is because if multiple mutation operators are applied on the same mutant (1) analysts would lose the ability to evaluate a clone detector's ability to detect a specific type of clone, and (2) multiple modifications may cause a mutant to be so different that it is no longer a clone.

Some mutation operations can mutate in a random way depending on the type of mutation it is. For example, "deleting an element" or "adding an element" mutation operation can have a different element selected each time the mutation operation

Listing 1 Example Mutant Meta Data

```xml
<?xml version="1.0" encoding="utf-8"?>
<clones>
   <original subsystem="powerwindow/window_system">
      <block path="powerwindow/window_system/up"/>
      <block path="powerwindow/window_system/down"/>
      <block path="powerwindow/window_system/angular␣velocity"/>
      <block path="powerwindow/window_system/c0"/>
      <block path="powerwindow/window_system/c1"/>
       <!-- More blocks -->
   </original>
   <mutant subsystem="window_system_mRB/Subsystem">
      <block path="window_system_mRB/Subsystem/up"/>
      <block path="window_system_mRB/Subsystem/down"/>
      <block path="window_system_mRB/Subsystem/angular␣velocity"/>
      <block path="window_system_mRB/Subsystem/c0"/>
      <block path="window_system_mRB/Subsystem/c1"/>
       <!-- More blocks -->
   </mutant>
   <mutant subsystem="window_system_mRL/Subsystem">
          <!-- More blocks -->
   </mutant>
   <!-- More mutants -->
</clones>
```

is executed. The same goes for modifying a connection or changing system hierarchies. Thus, we recommend MuMonDE implementers provide analysts the ability to specify how many randomized mutations of each type they prefer. If an analyst specifies they would like Z executions, or variants, of the Y mutation operators, then there would be X*Y*Z mutations performed. This extra level of randomness and variation adds to MuMonDE's rigor and gives analysts more flexibility in how extensive they want their mutations to vary. This is represented in the right side of Figure 3 as we see X*Y*Z copies of the SIS and a corresponding folder containing the SIS, the mutated SIS, and the eventual model clone detection results.

This mutation phase of MuMonDE can be implemented in an automated way, as we did in our Simulink implementation. Our Simulink specific injection code is described in our previous work [22], and can be downloaded from our website for use by MuMonDE developers[3].

To facilitate recall calculation, it is necessary for MuMonDE to record information about the mutants during injection. This can be accomplished by storing data about the mutants in the form of mutant meta data. This meta data must have enough information such that each original SIS can be linked to any mutant, and the specific mutation operation can be identified. It must describe the mutant itself, with a link/relation to the original, and the operation applied. In our Simulink MuMonDE implementation, we represented this data in the form of one XML file per SIS, such as the example we present in Listing 1 to demonstrate a simple but sufficient data store. Specifically, this mutant meta data file has a root clones element that contains one child element called original, representing the original system, and one-to-many elements called mutant, representing the mutant systems that should be identified as clones. These children elements contain a subsystem attribute that identifies the full path to the original Simulink subsystem or mutated subsystem, respectively. The mutated subsystem path also includes the operation applied as indicated in last element of the path, for example, mRB for a renamed block. Because Simulink's fully qualified paths are unique, and there will be one copy of each system for each operator applied, we required a way to differentiate between the different systems and their elements. The original and mutant elements contain block elements, which have a path attribute. We believed it was necessary to store it in this way and include all blocks since every block is important in determining similarity and identifying inner nested clones during the evaluation phase. This data corresponds to the processed clone reports that will be available in the next phase. It is up to each MuMonDE implementation to determine what meta data it requires. The implementation will have to interpret that meta data to ascertain the operation performed, and the changes between an original and mutants.

4.1.4 | Application to UML

When determining mutation targets, UML MuMonDE implementations can employ the same randomization techniques for both hierarchical and non-hierarchical UML model types. For example, UML implementations can have a function that selects a user-configured, X, number of class diagrams to mutate. Our MuMonDE guidelines likely apply when it comes to duplicating

[3]See footnote 2

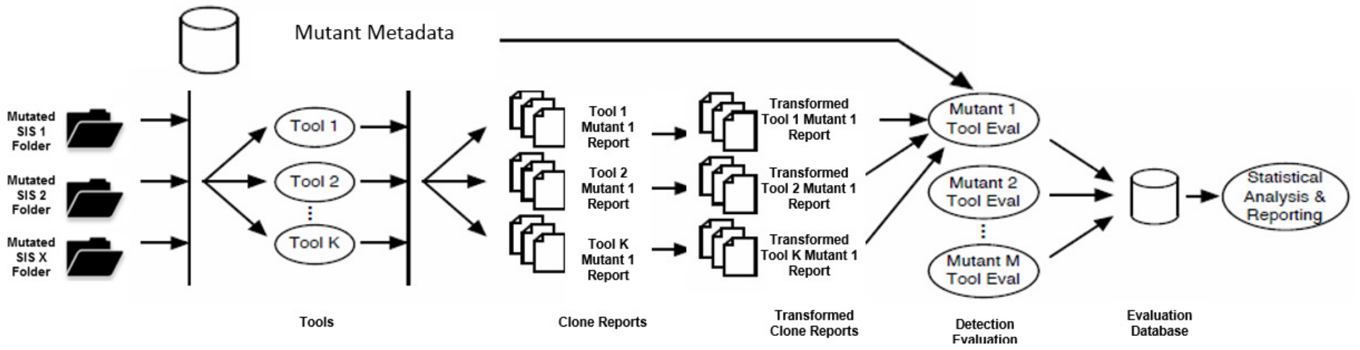


FIGURE 4 MuMonDE's Evaluation Phase

UML models instead of performing direct mutant injection, especially for hierarchical UML model types, such as hierarchically nested UML state diagrams.

For selecting model mutations, an implementer must select UML mutations that generate different clone types. In our experience, research on UML model mutations is sparser than Simulink model mutation. Krenn et al. [64] developed UML model mutations focusing on changing behavior in UML state diagrams to generate test cases. Examples include changing guards, effects, and entry/exit actions. Similarly, Granda et al. [65] developed mutation operators for conceptual schema specific UML class diagrams. While these operators are not intended to generate model clones, they may be a good starting point. Whatever operators are chosen by a MuMonDE implementer, our general process guidelines apply. An implementer must classify the operators by the clone type they generate, and can use our suggestions for devising categories. For Type 1 UML model clones, element positioning, line layout, annotations, and many other visual properties can be mutated. When it comes to UML Type 2 model clones, element names, like class names, have a large bearing on similarity measures and almost all UML elements have names [35]. So a UML MuMonDE implementation would have to rename UML element names accordingly. The same goes for value mutations in UML diagrams with values. For example, UML object diagrams have elements containing values indicating each object's state at a given time. When it comes to choosing operators for Type 3 UML model clones, our suggestions of performing UML model version history analysis and deducing operators from meta model interpretation apply. Type 4 UML semantic clone operators can be derived using semantically preserving rules, as outlined in our process.

We believe our described framework process for mutant injection can be applied to any type of model or modelling language, such as UML or data-flow languages. For UML, the model element injection eligibility depends on the specific operator being considered. For example, changing a UML element value applies only to UML elements with values. The notion of employing Z variants applies to UML models. The meta data for UML models would differ from our example. For the meta data indicating which UML mutation operator applied, using various naming conventions as we did in our example should suffice. The meta data required for identifying differences between the original and mutant UML models will differ from Simulink. A great resource for this is the wealth of work on UML model comparison [28, 26]. Various comparison approaches indicate that the meta data for a UML MuMonDE implementation would have to account for UML concepts such as subtype relationships, attribute assignments, and other language-specific aspects affected by the UML mutation operators. These are analogous to the steps we took in our Simulink MuMonDE implementation, which were not overly onerous.

4.2 | Second Phase: Evaluation

By the time of MuMonDE's second phase, the original SIS and their mutated versions are setup, organized, and ready for model clone detection. This phase is comprised of four steps that allow for tool execution and evaluation.

Figure 4 illustrates a general overview of the complete evaluation phase. The output from the mutation phase includes each SIS and its mutant variants stored in folders, and the mutant meta data. This phase entails running each model clone detection tool/configuration on those folders to generate reports, which are then processed for evaluation and comparison. The following sections go into detail on the steps in this phase, generalizing and fleshing out the details from our experiences implementing the framework in Simulink [18, 20, 22].

Listing 2 Example Normalized Form for Simulink Clone Class Results

```
<clones>
    <class classid="..." ... --(Additional/Optional Class Attributes)-->
    <!-- Each class element corresponds to a clone class -->
        <source ... --(Additional/Optional Source Attributes)-->
        <!-- Each source corresponds to a clone instance within a class -->
            <block path="..." ...--(Additional/Optional Block attributes)--/>
            <!-- ...More Blocks... -->
        </source>
        <!-- ...More Sources... -->
    </class>
    <!-- ...More Classes... -->
</clones>
```

4.2.1 | Perform Model Clone Detection

With the original and mutated SIS in place model clone detectors can be used on these systems. MuMonDE should have each detector, or configuration, attempt to detect the model clones created in phase one. For example, if there are three different tools, or three different configurations of the same tool, then the variable K in Figure 4 will equal three. There will be three tools/tool executions, and three model clone reports for each SIS. These executions can be realized in a MuMonDE implementation by having them either started manually by a framework user/analyst or automatically through a script. The output of this step is unmodified model clone reports. In our Simulink implementation, and with the model clone detectors we evaluated, model clone detection was applied in an each-with-each fashion. That is, the original SIS and the mutated SIS variants underwent clone detection together.

4.2.2 | Preprocess Model Clone Reports

Model clone reports from different tools likely differ from one tool to the next. In addition, there must be a mapping from the model clone reports to the mutant meta data describing injected model clones. Thus, this step in the framework involves transforming the output of the model clone detectors into a normalized format.

To address this in our Simulink MuMonDE implementation, we devised a consistent, target, format. We present our general form of this model clone report representation in Listing 2. We had to ensure that it had enough information so that, when analyzed automatically alongside the mutant meta data, it can identify if a mutant was killed. We aimed to make it simple as possible, while allowing for flexibility in element attributes. In our previous work [18], to address the problem of nested clones, we defined Fragment Containment for the modelling domain. It is a situation where all elements belonging to a model clone introduced by mutation are a subset of the elements belonging to a model clone detected by a specific tool and/or configuration. This corresponds well with the concept of "killed" mutants in mutation analysis [1], as it is an instance of the "non-overlap binary definition". Thus, the necessary information for calculating mutant death in our implementation had to include all blocks belonging to each clone instance. This helped us reduce bias by allowing us to check for both inner and outer clones.

Since Simulink blocks have unique fully-qualified paths, we used paths as block identifiers. As shown in Listing 2, our clone result representation begins with a clones element representing a listing of clones. It contains class elements as children with an attribute classid that identifies a model clone class. We decided on including model clone class as an element in our MuMonDE implementation because all the Simulink model clone detector tools we encountered detect classes but not all of them explicated clone pairs. Each class element contains model clone instances, which we represent using the source element. Each model clone instance contains all the blocks belonging to it in the form of children block elements. The block elements contain an attribute path that indicates the fully-qualified unique path to the specific block. This format is from our Simulink MuMonDE implementation. It 1) assumes all model clone detectors being evaluated identify model clone classes, and 2) uses the notion of "blocks", which is specific to Simulink. Future MuMonDE implementations can use this as a guideline, tailoring it to their specific modelling language if necessary. For example, we describe how a UML normalized form may differ from this in Section 4.2.5.

In order to have MuMonDE be as automatic as possible, a transformation from a tool's native clone report format into the normalized format is required. Manual creation of the transformation rules would need be done by a MuMonDE implementer seldomly. They would have to create one the first time a tool is being evaluated. They would then need to update it whenever an existing tool changes its output format, which may be dependent on meta-model changes. Once in place, execution of the transformation on all model clone reports can be automated and performed by MuMonDE on the clone reports as they are generated. Since model clone reports formats should be kept as simple as possible, these transformations should not be overly onerous nor computationally heavy. Especially when using a mature and robust transformation tool, like TXL [42], which we used for our Simulink MuMonDE implementation transformations. These are available for download in our repository [4].

4.2.3 | Analyze Model Clone Reports

Once the model clone reports are processed and normalized, MuMonDE implementations can facilitate automatic evaluation. This includes recall and precision calculation, which is now possible since the systems listed as model clones can be compared against those described in the mutant meta data. Using this information MuMonDE can begin calculating recall by identifying "killed" and "missed" mutants, and calculating precision by scrutinizing identified model clone pairs. Our Simulink MuMonDE implementation code for calculating recall and precision can be found in our repository.

Recall

MuMonDE recall calculation involves determining which injected model clones were detected by the respective tools/configurations, and which were not. The denominator in MuMonDE's recall function corresponds to the number of mutant systems injected into the projects by a MuMonDE implementation. The numerator is the number of those mutant systems detected by each tool/configuration as a model clone. We present a general form for MuMonDE recall calculations in Equation 1 and 2. These differ from our previous work by considering model clone pairs instead of clone classes only. Equation 1 indicates that if an original (non-mutated) system, OS, is either a clone pair or in the same clone class as mutated system, M, then that mutated system has been killed. In Equation 2, the complete recall for a model clone detector/configuration is calculated by adding all of the detected mutants found for each SIS using Equation 1 and dividing that number by the number of mutants created, MI. MuMonDE has this required information in the form of the mutant meta data files and normalized model clone detection reports.

$$Recall_{M,OS} = \begin{cases} 1, & \text{if M\&OS are clone pair OR in same clone class;} \\ 0, & \text{Otherwise.} \end{cases} \tag{1}$$

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

It is valuable for a MuMonDE implementation to indicate to analysts which mutants were missed. This information may be most important as it can identify weaknesses and areas for improvement. We accomplished this in our Simulink MuMonDE implementation, and provide an example of that in Figure 5 , which is an evaluation report of mutants from a system in Simulink's automotive demonstration set. This evaluation report contains the recall percentage of killed mutants for this system by ConQAT, 38.5%. It explicates missed model clone mutants that ConQAT was unable to kill and its precision for this system. The suffixes of each of the system names, like _mCBV and _mDBS, indicate which mutants were missed. While this was expected since ConQAT is not capable of detecting near-miss clones, it is helpful to have an explicit list as this can identify interesting cases as we discuss later.

Precision

Precision can be defined as the number of correctly identified mutant model clones divided by all the clone pairs identified by a specific model clone detection tool. This is a different challenge than recall because there may be lots of model clone pairs that were introduced during mutation injection beyond the expected ones. Some, or even all, of those are likely "correctly" identified as model clone pairs by tools. In order to accommodate this in MuMonDE, we modify the strategy employed in the code clone tool evaluation framework [55] of doing a line-by-line source code validation to make it work with models. Specifically, MuMonDE involves construction of a model clone pair validator. The validator investigates each of the model clone pairs detected, and performs a one-versus-one similarity check. This can be done by comparing each sub/contained element within each model clone pair, and coming up with similarity metrics. It is important to stress the validator is just that, and not a model clone detector. Specifically, analogous to the code-clone evaluation framework approach, MuMonDE validates clones by validating a single pair at a time using a priori knowledge of the specific mutations it injected. For example, to address this in our Simulink implementation, we verified that each system had less than or equal to two matching block differences,

[4]See footnote 2

```
Recall:
    0.385
Mutants Missed:
    detect_endstop_mCBV/Subsystem
    detect_endstop_mCBT/Subsystem
    detect_endstop_mABD/Subsystem
    detect_endstop_mDBS/Subsystem
    detect_endstop_mAIB/Subsystem
    detect_endstop_mDIB/Subsystem
    detect_endstop_mCS/Subsystem
    detect_endstop_mABS/Subsystem
Precision:
    1.0
```

FIGURE 5 Evaluation of ConQAT on Powerwindow/detect_endstop System

and had no more than 30% different elements. We use that percentage since we previously established it as a solid difference threshold [21]. To identify matching block differences, we leverage (1) the Simulink fully qualified path properties we discussed earlier, whereby each fully qualified path uniquely identifies a block in a Simulink model, and (2) that clones are being detected among the original systems and their duplicated mutants. Thus, we can tell if a block is present in a mutant, a connection has changed, or it has been modified. Another possible approach we could have employed would be to compare the entire underlying textual representations of the blocks after preprocessing the blocks. We discuss refinement of our Simulink-specific precision validator later as future work.

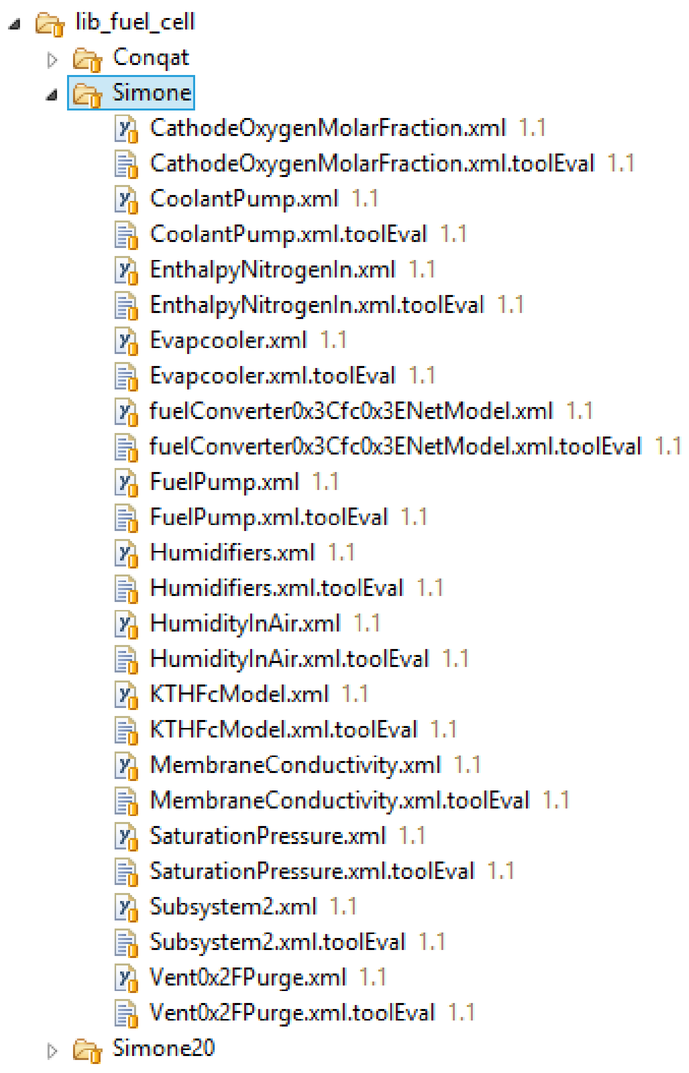
MuMonDE's precision calculation thus requires validation of each pair of detected model clone instances by doing an element-by-element comparison. It is important to remember that validation is done one system at a time only, looking at elements from an original system and its corresponding mutant systems. As we discuss in our evaluation, we experimented on industrial and industrial-sized systems and our analysis ran within minutes. However, we should acknowledge that this could be a potential bottleneck depending on how many elements are present in the SIS. The number of valid clone pairs is the numerator in the precision calculation. As demonstrated in Equation 3, which is the same equation employed in our Simulink MuMonDE implementation, total precision can be calculated as the summation of all valid clone pairs over all reported clone pairs across all SIS.

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

Similarly to what is done for recall, MuMonDE implementers should enumerate the invalid clone pairs to allow tool evaluators to investigate any invalid pairs. This can appear in a report output similar to the one we presented in Figure 5 .

4.2.4 | Visualize Results

Once a MuMonDE implementation calculates precision and recall its final step is to present results. It is important to provide both a report of the overall recall and precision of the tools/configurations and detailed lower-level reports. The MuMonDE implementation can employ a higher level layout similar to what we did in our Simulink MuMonDE implementation, such as the example in Figure 6 . Each mutated element can have a folder, such as the "lib_fuel_cell" model from the AVS Simulink project in our example. This folder should have sub folders for each tool execution. In our case, we had one for ConQAT, one for Simone using default configurations, and one with Simone configured with a 20% difference similarity threshold. The highlighted and expanded "Simone" folder contains both the normalized model clone report in *.xml* format and the corresponding tool analysis text file with a *.toolEval* extension as presented in Figure 5 . The benefits of this layout include separating each tool execution to allow for granular system analysis. A tool evaluator can identify specific strengths and areas of improvement by seeing specific recall and precision factors for each SIS. In our implementation, we had total recall and precision calculated during evaluation,



```
lib_fuel_cell
    Conqat
    Simone
        CathodeOxygenMolarFraction.xml 1.1
        CathodeOxygenMolarFraction.xml.toolEval 1.1
        CoolantPump.xml 1.1
        CoolantPump.xml.toolEval 1.1
        EnthalpyNitrogenIn.xml 1.1
        EnthalpyNitrogenIn.xml.toolEval 1.1
        Evapcooler.xml 1.1
        Evapcooler.xml.toolEval 1.1
        fuelConverter0x3Cfc0x3ENetModel.xml 1.1
        fuelConverter0x3Cfc0x3ENetModel.xml.toolEval 1.1
        FuelPump.xml 1.1
        FuelPump.xml.toolEval 1.1
        Humidifiers.xml 1.1
        Humidifiers.xml.toolEval 1.1
        HumidityInAir.xml 1.1
        HumidityInAir.xml.toolEval 1.1
        KTHFcModel.xml 1.1
        KTHFcModel.xml.toolEval 1.1
        MembraneConductivity.xml 1.1
        MembraneConductivity.xml.toolEval 1.1
        SaturationPressure.xml 1.1
        SaturationPressure.xml.toolEval 1.1
        Subsystem2.xml 1.1
        Subsystem2.xml.toolEval 1.1
        Vent0x2FPurge.xml 1.1
        Vent0x2FPurge.xml.toolEval 1.1
    Simone20
```

FIGURE 6 Example Presentation Layout

however, it is also possible to iterate through these .toolEval files and extract the necessary information. This is just an example. MuMonDE implementers can choose file types and hierarchies that work for their domain.

MuMonDE's final step is to present the results in a form conducive to analysis and sharing. This can be in the form of graphs, tables, et cetera. This step can be automated by having a MuMonDE implementation interpreting the .toolEval files, and generating a CSV or other file for use in graphing software. For our Simulink implementation, we created tables and charts manually, such as the ones we discuss in Section 5.

4.2.5 | Application to UML

Applying this phase's steps to UML and other modelling languages is simpler than the mutation phase, as it is more language agnostic. For example, performing model clone detection would be exactly as we laid out earlier, executing K tool executions generating K unmodified clone reports.

Preprocessing model clone reports depends on both the modelling language and model clone detectors being evaluated by MuMonDE. For UML modes, the target clone report format could include the fully qualified path to the diagram as the source. The diagram's elements can be listed with their UML universally unique identifier (UUID), along with any additional element attributes, as we did for Simulink blocks. The attributes would depend on the nature of the UML mutations injected in the first phase. This would work with the emerging UML model clone detection approaches [34, 35]. However, the approach by Störrle does not yet cluster or classify UML model clones into classes. Instead, they identify a list of model clone pairs based on similarity heuristics. So clone class information would have to be omitted by a MuMonDE implementer. As laid out in the MuMonDE framework, the transformation from original to target UML clone report would need be created only once per tool, updated accordingly, and executed automatically on the reports as they are generated.

The formulae and process we presented for calculating recall and precision should apply as is to any modelling language and model clone detector. Especially since we account for tools that identify either only clone pairs or only clone classes. The only language-specific aspect is the model clone validator. For UML models, UML model clone pairs can be compared and validated using existing UML model comparison approaches [28, 26] to calculate the differences in identified clones. Visualizing the recall and precision results is not language specific and can be accomplished following our framework guidelines.

5 | VALIDATION: RESULTS OF MUMONDE SIMULINK IMPLEMENTATION

In order to help validate the MuMonDE framework, we focus on the engineering value that MuMonDE provides in its ability to answer questions and provide insights. The questions we wanted to answer in our experiment involved evaluating Simone against the only other Simulink model clone detector available to us, ConQAT, and comparing two Simone configurations. They included,

1. How does the recall and precision of Simone with default settings compare against ConQAT?
2. How does lowering Simone's similarity threshold impact recall and precision?

Regarding the first question, it required one execution of each tool. ConQAT has no configuration when detecting system clones. For Simone, we were interested in Simone's default, 30% similarity threshold settings. In other words, models must be at least 70% similar to be considered a clone [21]. These settings were chosen based on empirical evaluation of similarity metrics for both code and model clone detectors [66]. Therefore, evaluating that setting was of interest to us. However, as demonstrated by Ragkhitwetsagul et al., clone detectors often can have their similarity settings adjusted and default settings are not always appropriate nor best [67]. Thus, it behooves analysts, and clone detector developers, to experiment and optimize their clone detectors for their specific domains and purposes. So, we posed the second question using Simone with a 20% similarity threshold, requiring clones be at least 80% similar to one another. Based on our experience with model clones, any lower setting resulted in trivial clones that had very little difference. Any clones found with a threshold higher than 30% were very different and uninteresting. We found that the model clone results in running Simone at 25% were not very different from the 30% default settings.

ConQAT is able to detect type 1 and 2 model clones only. Therefore, the result presentation we chose for our Simulink MuMonDE implementation is biased towards Simone. However, we believe it prudent to evaluate a model clone detector's ability to detect all types at once. Additionally, the results presentation process we describe still identifies type 1 and 2 model clones missed by ConQAT.

5.1 | Simulink Models Considered

Table 2 displays information about the models we mutated. The PowerWindow (PW) project is the automotive model set that comes with Simulink. It has one model only, but has many versions increasing in complexity. The Advanced Vehicle Simulator (AVS) system is a large scale open-source Simulink project[5]. We decided to go with two of the larger library models that contained rich systems. We also performed these experiments on our industrial partner's models, but we omit those results for proprietary reasons. In all cases, mutation injection through our MuMonDE implementation took a few minutes at most. Additionally, we contend that these open-source models are comparable size-wise [19], can be downloaded by readers, and act the same way as industrial models when being mutated. The "# of Systems Mutated" column in Table 2 indicates the number of randomly selected systems that were mutated. We had our program select at least a dozen systems from each model, if that many existed. We could have selected more, however the key is the amount of random of mutations injected, which is listed in the "# of Mutations Injected" column. We ended up with roughly one hundred fifty random mutations injections from fourteen different types of mutation operations for each model, totaling five hundred forty six mutations across all projects. For this

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

TABLE 2 Overview of Models and Injected Mutants

Project	Model	# of Systems	# Systems Mutated	# Mutations Injected
PW	PowerWindow(V1)	18	7	83
	PowerWindow(V3)	33	13	153
AVS	fc_KTH_lib	83	12	146
	lib_fuel_Cell	30	13	164

TABLE 3 Results of ConQAT Evaluation using MuMonDE

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

column, it may appear as if we can multiply the number of systems selected randomly by the number of Simulink mutations we implemented, fourteen. However, some of the mutations; such as deleting a block in the middle of a system, changing block values or types, and one or two other mutations; may not be executed on each system if there are no valid targets or locations in that system. In our case, there were more than enough systems that had all fourteen mutations injected successfully.

5.2 | Results

To help demonstrate MuMonDE's capabilities, we present the results of using our Simulink MuMonDE framework implementation to answer our two evaluation questions. For both our recall and precision results, we manually verified a small sample of the results on several systems we were very familiar with to make sure the automation proceeded as we expected. This manual validation was also helpful in understanding the nuances of the interesting examples we present in this section.

5.2.1 | Question 1: ConQAT versus default Simone

Running ConQAT on the model sets generated with MuMonDE allowed for the assessment demonstrated in Table 3 . While we knew beforehand ConQAT's implementation allows it to detect type 1 and some type 2 clones only, this is better quantified using MuMonDE. Since roughly two thirds of the types of mutation operations are type 3 [19], it is not surprising that ConQAT's recall was calculated as roughly 33%. One recurring type 2 missed mutant of interest was the changing of a block's value. ConQAT's labeling algorithm sometimes includes the block's value and sometimes does not [13]. For example our mutation operator mutated the *detect_endstop* system in the power window model by changing a constant block's numerical value. ConQAT viewed that block as a completely different block and, because they match only exact systems and not near-miss, it missed this mutant. While that is an implementation decision on their part, it is clear that these two systems should be reported at least as some type of clone pair. Simone reported this as a clone, while ConQAT did not at all.

We made other interesting observations about ConQAT and type 2 mutations they missed. For example, the rename block mutation was not killed on occasion. Through investigation, we learned this was due to ConQAT not considering nor including inner inport and outport blocks from systems in their evaluation. Specifically, during their flattening algorithm removing hierarchies, they ignore the inner "version" of the ports and use the outer versions that are connected to the subsystem being flattened. If the inner block is the one that is renamed in a potential type 2 clone it will be missed by ConQAT.

We present Simone's evaluation using MuMonDE with our model sets in Table 4 . Seeing as Simone is intended to detect all three types of clones, the expectation was a fairly high recall, which was confirmed by MuMonDE. The one 77% result was attributed to very small systems, which Simone is configured to ignore by default, but can be set to include. The important takeaway here is that by having MuMonDE in place, we were able to identify "missed" clone pairs, and determine the cause. It also lead us to ask our industrial partners questions related to "non-trivial" system model clone sizes, and allowed us to determine what were the minimum size of clones of interest to them. Another useful observation made possible by MuMonDE

TABLE 4 Results of Simone Evaluation using MuMonDE

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

TABLE 5 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%

was involving mutants created by subsystem extraction. In cases where three or less blocks were extracted into a subsystem, Simone was unable to identify a clone. This MuMonDE observation was useful to us as Simone developers since we realized we could try to reduce noise in Simone's textual normalization facility.

5.2.2 | Question 2: Lowering Simone's Similarity Threshold

The MuMonDE evaluation of running Simone with a 20% threshold is presented in Table 5 . We note a decrease in recall and an increase in precision compared to running Simone with its default, 30%, settings. This is expected when being more selective [68], however MuMonDE allows us to quantify the difference. Specifically, we note a total of eight recall percentage points dropped across all four models, and an increase of seven precision points. An interesting avenue of future research made possible by MuMonDE is to determine the rate of change in losses/gains when changing the similarity threshold.

One example of a recurring clone pair that was identified by Simone with default settings but not using 20% were those created by the "changing a subsystem's clone hierarchy" mutation. This mutation mimics the operation of refactoring model elements from a system into a subsystem, which can be done through a single Simulink command [19]. One example system was the *CathodeOverpotential* model in the AVS model set. It was mutated by moving a number of blocks, including a sum and product block, into a Simulink subsystem. Simone, allowing for at most 20% difference, failed to identify this clone pair (that is, kill this mutant). Simone with default settings successfully killed the mutant. While Simone at 20% was able to detect other mutants of this variety, in this instance the additional gluing of adding ports and connections in extracting the blocks into a subsystem caused too much of a difference. One could argue that a clone should still be identified here because the two systems differ by a single operation only. This information, made possible by our Simulink MuMonDE implementation, can be used by both analysts and Simone clone detector developers. The developers can interpret this to mean there might be too much textual noise in Simone's normalized form when accounting for embedded subsystems. They could try addressing this by attempting to reduce noise in Simone's normalization facility. For analysts, they may considering using Simone's default configuration instead of 20% if identifying all subsystem clones is important for their analysis.

5.3 | Result Visualization and Interpretation

The last step in MuMonDE involves presenting the results to analysts for interpretation. The tables in the previous section are examples of possible MuMonDE visualizations. Following the steps we described earlier for our Simulink MuMonDE implementation, another potential visualization could be to generate bar charts in Excel that contrast the recall and precision of each tool execution by feeding it MuMonDE data.

Simone's recall was typically in the high nineties. The exception was the lib_fuel_Cell case caused by models that were deemed too small by Simone's minimum size configuration. When we tuned Simone to detect model clones only with 20%

difference or less, the recall decreased. ConQAT detected roughly one third of the injected mutants because of its inability to detect near-miss clones, the interesting missed Type 2 examples we mentioned, and more.

Both ConQAT and Simone have very high precision. Using MuMonDE, we also identified the precision and recall trade off in reducing Simone's difference threshold from 30% to 20%.

5.4 | Threats to Validity

We acknowledge the following threats to validity, which we divide into external and internal threats.

5.4.1 | External Validity

Our validation was based on our ability to implement MuMonDE for Simulink models, and use that implementation to answer questions and provide insights. So, an important question to consider is the whether the selected models are representative of Simulink models in general. Beyond choosing them for their public availability, we contend that they are representative in both size and complexity. As noted in our earlier work [19], the AVS system actually has more model files and subsystems than our industrial model sets. We contend the mutation operations will behave uniformly on all sets of Simulink models since they all belong to the same Simulink mutation classes. Thus, this threat to validity applies more to the Simulink model mutation taxonomy, the refinement of which we consider future work.

An important external threat and area of future work for MuMonDE is its extension to other model types. Model clone detection tools are arising for other model types, such as UML. However, these are still relatively immature considering the only complete tool, MQClone, is not yet validated [35]. Implementing MuMonDE for UML model clones once that area has matured will not only further validate it as an evaluation tool, but will improve UML model clone detection. While this is a threat, the concepts around software modelling are roughly the same regardless of the modelling language. They involve connected graphs of elements describing structure and/or semantics of a software system. It is possible that there are difficult or impossible nuances to other languages that we were not able to foresee in devising MuMonDE. Of course, those can only be exposed by actually attempting to implement the framework in those languages. We just have yet to do so for anything but Simulink due to its model clone maturity and it being the language of interest to our industrial partners.

Because MuMonDE is primarily a general framework, designed to be model agnostic, implementers can tweak various aspects as they see fit. Also, from a planning perspective, it is helpful to have a completed implementation to ascertain the amount of work necessary. The most obvious shortcoming in following MuMonDE is that some of our examples may not be as applicable to other model types. For example, the meta data we employed and published for our Simulink implementation may not be as applicable to other modelling languages, thus requiring implementers to construct an entirely new meta data schema. While we included discussions on UML, other modelling languages may require very different mutations and/or clone report transformations. However, we contend learning from our experiences and using our existing approaches/code as a foundation would assist even analysts that do not follow MuMonDE.

5.4.2 | Internal Validity

It is prudent to consider whether implementing MuMonDE for a new model type is easier than implementing a new analysis from scratch. We did not have a control case to test this, so we can not make any such claim with certainty. The benefits in following MuMonDE include the ability to implement it in an example-driven manner. We described each step in detail, along with a working example that can be used as a starting point.

6 | FUTURE WORK

Future work includes, among other things, working on addressing the threats to validity described previously.

A future improvement is to refine MuMonDE's mechanism for calculating precision. It is roughly the same process in the analogous code-clone evaluation framework [55], however, it does not apply as directly because the validation process is not as simple. While the overall process for model clone validation will remain mostly similar, it potentially can be more formal for models. One approach may be to compare blocks from a clone pair by contrasting their underlying textual representation. However, this would likely require filtering out "noise" and sorting the lines, as we were forced to do when devloping Simone [21]. An interesting future work experiment would be comparing the performance of such a validator with the one we implemented.

Lastly, we are continuously working with our industrial partners and the research community on improving the presentation of results before we automate it in our Simulink implementation. It is our goal to have the information as easy to understand and as useful as possible. This includes discussions with both engineers and analysts from industry, and collaborations with our research colleagues.

7 | CONCLUSIONS

We presented a framework, MuMonDE, that applies model mutation analysis to assess model clone detectors by injecting model clones into projects. The idea is to inject variations of model clones created through mutation and determining if the model clone detectors can find them. MuMonDE is comprised of two main phases. The first phase is the mutation phase. It begins by having a framework implementer devise a way to select the targets for mutations, for example, random system selection within models. The specific categories and types of mutation operators must be selected so that many variations of model clones are injected into projects that are representative of real life model clones and cover the different model clone types. The last step in the first phase is to inject the mutation operations. The second phase involves observing the results of model clone detection on the mutated targets and evaluating those results to ascertain recall and precision. This phase involves running different model clone detection tools and/or configurations. The model clone reports that come from the tools must be normalized so that they can be easily analyzed and contrasted. Recall and precision are calculated from the normalized model clone reports, with the concluding information being presented from both a high-level big-picture view and through smaller more-detailed reports.

MuMonDE has been successfully implemented for Simulink model clone detectors. We reference our implementation, experiences, and results throughout the paper. While further validation by realizing MuMonDE for model types other than Simulink would be ideal, the immaturity of those research areas prevents that. We hope MuMonDE will be used to further research in both model clone detection and model mutation. We believe it will yield similar success as it did for us in evaluating Simulink model clone detectors. By presenting this research and giving way to better model clone detection techniques, our goal is to improve model driven engineering as a whole and demonstrate another novel application of mutation analysis.

ACKNOWLEDGMENT

This work was supported in part 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] Acree A, Budd T, DeMillo R, Lipton R, Sayward F. Mutation analysis. Technical Report, DTIC Document 1979.

[2] DeMillo RA. Mutation analysis as a tool for software quality assurance. Technical Report, DTIC Document 1980.

[3] Offutt AJ. A practical system for mutation testing: help for the common programmer. International Test Conference, 1994; 824–830.

[4] Kent S. Model driven engineering. Integrated formal methods, 2002; 286–298.

[5] Schmidt DC. Guest editor's introduction: Model-driven engineering. Computer 2006; **39**(2):25–31.

[6] Fabbri SCPF, Maldonado JC, Sugeta T, Masiero PC. Mutation testing applied to validate specifications based on statecharts. International Symposium on Software Reliability Engineering, 1999; 210–219.

[7] Mottu JM, Baudry B, Le Traon Y. Mutation analysis testing for model transformations. Model Driven Architecture–Foundations and Applications, 2006; 376–390.

[8] Fabbri SC, Delamaro ME, Maldonado JC, Masiero PC. Mutation analysis testing for finite state machines. International Symposium on Software Reliability Engineering, 1994; 220–229.

[9] Volter M, Stahl T, Bettin J, Haase A, Helsen S. Model-driven software development: technology, engineering, management. John Wiley & Sons, 2013.

[10] France R, Rumpe B. Model-driven development of complex software: A research roadmap. 2007 Future of Software Engineering, 2007; 37–54.

[11] Mohagheghi P, Aagedal J. Evaluating quality in model-driven engineering. International Workshop on Modeling in Software Engineering, 2007; 6 pp.

[12] Punter T, Voeten J. Quality in model driven engineering. Model-Driven Software Development: Integrating Quality Assurance. IGI Global, 2009; 37–56.

[13] Deissenboeck F, Hummel B, Juergens E, Schaetz B, Wagner S, Girard JF, Teuchart S. Clone detection in automotive model-based development. ICSE, 2009; 603–612.

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

[15] Stephan M, Cordy JR. Identifying instances of model design patterns and antipatterns using model clone detection. International Workshop on Modelling in Software Engineering, 2015; 48–53.

[16] Stephan M, Cordy JR. Identification of simulink model antipattern instances using model clone detection. International Conference on Model Driven Engineering Languages and Systems, 2015; 276–285.

[17] Stephan M, Cordy JR. Model-driven evaluation of software architecture quality using model clone detection. International Conference on Software Quality, Reliability and Security (QRS), 2016; 92–99.

[18] Stephan M, Alafi M, Stevenson A, Cordy J. Using mutation analysis for a model-clone detector comparison framework. ICSE, 2013; 1277–1280.

[19] Stephan M, Alalfi M, Cordy JR. Towards a taxonomy for simulink model mutations. International Conference on Software Testing, Verification, and Validation 2014 (ICST) – Mutation Workshop, 2014; 206–215.

[20] Stephan M. Model clone detector evaluation using mutation analysis. International Conference on Software Maintenance and Evolution: Doctoral Symposium, 2014; 633–638.

[21] Alalfi MH, Cordy JR, Dean TR, Stephan M, Stevenson A. Models are code too: Near-miss clone detection for Simulink models. ICSM, 2012; 295–304.

[22] Stephan M. A mutation analysis based model clone detector evaluation framework. PhD Thesis, Queen's University 2014.

[23] Koschke R. Survey of research on software clones. Duplication, Redundancy, and Similarity in Software 2006; :1–24.

[24] Kapser CJ, Godfrey MW. "cloning considered harmful" considered harmful: patterns of cloning in software. Empirical Software Engineering 2008; **13**(6):645.

[25] Roy C, Cordy J, Koschke R. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Science of Computer Programming 2009; **74**(7):470–495.

[26] Stephan M, Cordy JR. A survey of model comparison approaches and applications. International Conference on Model-Driven Engineering and Software Development, 2013; 265–277.

[27] Selonen P. A Review of UML Model Comparison Approaches. 5th Nordic Workshop on Model Driven Engineering, 2007; 37–51.

[28] Stephan M, Cordy JR. A survey of methods and applications of model comparison. Technical Report 2011-582 Rev. 3, Queen's University 2012.

[29] Garey M, Johnson D. Computers and intractability, vol. 174. Freeman San Francisco, CA, 1979.

[30] Gold N, Krinke J, Harman M, Binkley D. Issues in clone classification for dataflow languages. International Workshop on Software Clones, ACM, 2010; 83–84.

[31] Cordy JR. Submodel pattern extraction for simulink models. International Software Product Line Conference, 2013; 7–10.

[32] Pham NH, Nguyen HA, Nguyen TT, Al-Kofahi JM, Nguyen TN. Complete and accurate clone detection in graph-based models. ICSE, 2009; 276–286.

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

[34] Antony E, Alalfi MH, Cordy JR. An Approach to Clone Detection in Behavioural Models. International Working Conference in Reverse Engineering, 2013; 472–476.

[35] Storrle H. Towards clone detection in UML domain models. Software & Systems Modeling 2013; **12**(2):307–329.

[36] Dean TR, Chen J, Alalfi MH. Clone detection in Matlab Stateflow models. Electronic Communications of the EASST 2014; **63**.

[37] Kumar MA. Efficient weight assignment method for detection of clones in state flow diagrams. Journal of Software Engineering Research and Practices 2014; **4**(2):12–16.

[38] Gold N, Krinke J, Harman M, Binkley D. Cloning in max/msp patches. International Computer Music Conference, 2011.

[39] Al-Batran B, Schätz B, Hummel B. Semantic clone detection for model-based development of embedded systems. International Conference on Model Driven Engineering Languages and Systems, Springer, 2011; 258–272.

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

[41] Roy C, Cordy J. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. ICPC, 2008; 172–181.

[42] Cordy J. The TXL source transformation language. Science of Computer Programming 2006; **61**(3):190–210.

[43] Chen J, Dean T, Alalfi MH. Clone detection in matlab stateflow models. International Workshop of Software Clones, 2014; 1–10.

[44] Sen S, Baudry B. Mutation-based model synthesis in model driven engineering. Second Workshop on Mutation Analysis, 2006.

[45] Tran QM, Wilmes B, Dziobek C. Refactoring of simulink diagrams via composition of transformation steps. International Conference on Software Engineering Advances, 2013; 140–145.

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

[47] Adra SF, McMinn P. Mutation operators for agent-based models. International Conference on Software Testing, Verification, and Validation Workshops (ICSTW), 2010; 151–156.

[48] Bartel A, Baudry B, Munoz F, Klein J, Mouelhi T, Le Traon Y. Model driven mutation applied to adaptative systems testing. International Conference on Software Testing, Verification, and Validation Workshops (ICSTW), 2011; 408–413.

[49] Araujo RF, Vincenzi AMR, Delebecque F, Maldonado JC, Delamaro ME. Devising mutant operators for dynamic systems models by applying the HAZOP study. ICSEA 2011, 2011; 58–64.

[50] He N, Rümmer P, Kroening D. Test-case generation for embedded simulink via formal concept analysis. Design Automation Conference (DAC), 2011; 224–229.

[51] Zhan Y, Clark J. Search-based mutation testing for Simulink models. Genetic and Evolutionary Computation Conference, 2005; 1061–1068.

[52] Bellon S, Koschke R, Antoniol G, Krinke J, Merlo E. Comparison and evaluation of clone detection tools. Transactions on Software Engineering 2007; **33**(9):577–591.

[53] Schulze S, Meyer D. On the robustness of clone detection to code obfuscation. International Workshop on Software Clones, IEEE, 2013; 62–68.

[54] Ragkhitwetsagul C, Krinke J, Clark D. Similarity of source code in the presence of pervasive modifications. International Working Conference on Source Code Analysis and Manipulation, IEEE, 2016; 117–126.

[55] Roy CK, Cordy JR. A mutation/injection-based automatic framework for evaluating code clone detection tools. International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2009; 157–166.

[56] Svajlenko J, Roy CK, Cordy JR. A mutation analysis based benchmarking framework for clone detectors. International Workshop on Software Clones (IWSC), 2013; 8–9.

[57] Svajlenko J, Roy CK. Evaluating modern clone detection tools. International Conference on Software Maintenance and Evolution, IEEE, 2014; 321–330.

[58] Svajlenko J, Roy CK. Evaluating clone detection tools with bigclonebench. International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2015; 131–140.

[59] Uhrig S, Schwagerl F. Tool support for the evaluation of matching algorithms in the eclipse modeling framework. International Conference on Model-Driven Engineering and Software Development (MODELSWARD), 2013; 101–110.

[60] Jia Y, Harman M. An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering 2011; **37**(5):649–678.

[61] Yujian L, Bo L. A normalized levenshtein distance metric. IEEE Transactions on Pattern Analysis and Machine Intelligence 2007; **29**(6):1091–1095.

[62] Forney GD. Generalized minimum distance decoding. IEEE Transactions on Information Theory 1966; **12**(2):125–131.

[63] Falleri JR, Huchard M, Lafourcade M, Nebut C. Metamodel matching for automatic model transformation generation. Model Driven Engineering Languages and Systems. Springer, 2008; 326–340.

[64] Krenn W, Schlick R, Tiran S, Aichernig B, Jobstl E, Brandl H. Momut:: Uml model-based mutation testing for uml. International Conference on Software Testing, Verification and Validation, IEEE, 2015; 1–8.

[65] Granda MF, Condori-Fernández N, Vos TE, Pastor O. Mutation operators for uml class diagrams. International Conference on Advanced Information Systems Engineering, Springer, 2016; 325–341.

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

[67] Ragkhitwetsagul C, Paixao M, Adham M, Busari S, Krinke J, Drake JH. Searching for configurations in clone evaluation–a replication study. International Symposium on Search Based Software Engineering, Springer, 2016; 250–256.

[68] Buckland MK, Gey FC. The relationship between recall and precision. JASIS 1994; **45**(1):12–19.