Towards a Taxonomy for Simulink Model Mutations

Matthew Stephan, Manar H. Alalfi, and James R. Cordy
Queen's University
Kingston, Ontario, Canada
{stephan,alalfi,cordy}@cs.queensu.ca

***Abstract*—A relatively new and important branch of Mutation Analysis involves model mutations. In our attempts to realize model-clone detector testing, we found that there was little mutation research on Simulink, which is a fairly prevalent modeling language, especially in embedded domains. Because Simulink model mutations are the crux of our model-clone detector testing framework, we want to ensure that we are selecting the appropriate mutations. In this paper, we propose a taxonomy of Simulink model mutations, which is based on our experiences thus far with Simulink, that aims to inject model clones of various types and is fairly representative of realistic Simulink edit operations. We organize the mutations by categories based on the types of model clones they will inject, and further break them down into mutation classes. For each class, we define the characteristics of mutation operators belonging to that class and demonstrate an example operator. Lastly, in an attempt to validate our taxonomy, we perform a case study on multiple versions of three Simulink projects, including an industrial project, to ascertain if the actual subsystem edit operations observed across versions can be classified using our taxonomy and present any interesting cases. While we developed the taxonomy with the specific goal of facilitating and guiding the injection of mutants for model clones, we believe it is fairly general and a solid foundation for future Simulink model mutation work.**

I. Introduction

Mutation Analysis involves analyzing a software system using small modifications, or mutations, of elements within that system and observing how the system handles these changes [1]. The mutations, also termed mutation operators, either showcase an important property of a system or are representative of future modifications that may occur to a system's components. Up to this point, the majority of the work on Mutation Analysis in software testing is done on the source-code level. That is, mutation operators are devised to modify source code such that they test specific properties of a system. Most commonly, this involves the evaluation of the completeness of test suites by injecting potential errors and observing how the test suite covers them. Other examples of source-code level Mutation Analysis include mutating Java code to test concurrency [2], and C code to identify semantic misunderstandings [3].

In contrast to traditional source-code based development, Model-driven software development, or Model-driven Engineering (MDE), is a relatively recent and continually growing area within software engineering in which the primary artifacts are higher-level descriptive abstractions, or "models", that represent a system's elements and their behavior. It has seen much adoption in embedded areas, like telecommunications, and the automotive and aerospace domains. Compared to source-code based Mutation Analysis, Mutation Analysis for systems developed through MDE is still in its infancy.

One such MDE language that is of interest to us and our industrial partners is Simulink[1], which is a data-flow modelling language for embedded systems that allows for simulation, automatic code generation, and more. Specifically, we worked with our industrial partners and created a Simulink model-clone detector, Simone [4], that is able to identify identical or similar subsystems within Simulink projects. While we were able to perform a qualitative evaluation of our tool and comparison to other tools through manual testing [5] and analysis, we wanted to accomplish a more quantitative and automatic approach. To that end, we proposed, outlined, and are currently implementing a Model-Clone Detector Comparison Framework that utilizes Mutation Analysis [6].

A key part of this framework is identifying the appropriate Simulink model mutation operators that will allow for testing Simulink model-clone detectors. The contributions of this paper are proposing a taxonomy of classes for Simulink model mutations, which are based on our work with Simulink models thus far, and validating these classes through a case study that identifies and counts the mutation operators witnessed as evolutionary changes in multiple versions of three Simulink projects. While the mutation classes were devised with the goal in mind of testing Simulink model-clone detectors, we believe they are representative of Simulink model mutations in general and are a solid foundation to build on for future data-flow model Mutation Analysis.

This paper begins by providing background information on both Simulink and model-clone detection in Section II. We then propose our taxonomy, organized by categories that contain classes, in Section III. A case study we perform using three MDE projects to validate our taxonomy is presented in Section IV. We then discuss related work, future work, and conclude the paper in Sections V,VI, and VII, respectively.

II. Background

While we believe the Simulink model mutation classes are fairly general, we still make mention of their application to testing Simulink model-clone detectors, so we discuss model clone detection in Section II-B. To begin, we provide a very brief introduction to Simulink.

A. Simulink

Simulink models are data-flow models consisting of three levels of granularity: whole models, (sub) systems, and blocks.

[1]http://www.mathworks.com/products/simulink/

Models contain systems, and systems contain other (sub) systems and blocks. Blocks come from libraries, are connected by lines, and have their own semantics, allowing for parametrization and simulation. In addition to simulation, many blocks have corresponding C code that can be generated to embed into a target platform. Modellers edit Simulink models through the Matlab environment by navigating through systems and adding, modifying, and deleting blocks and lines. The underlying internal representation of Simulink models are stored as text either in Simulink MDL files or XML files, in newer versions of Simulink.

B. *Model Clone Detection*

Clone detection in software refers to the identification of similar or identical components within a set of projects. Model clone detection refers specifically to the identification of clones within model-based software systems and has been implemented in a variety of ways for different modelling languages [4], [7], [8]. It is generally accepted that there are three types of model clones [4]:

Type 1 - Exact model clones
These are identical fragments of models, regardless of variations in visual presentation, layout, and formatting.

Type 2 - Renamed model clones
Renamed model clones are those that are structurally the same, other than variations in labels, values, types, visual presentation, layout, and formatting.

Type 3 - Near-miss model clones
Near-miss model clones are those that are the same with additional modifications, such as position changes or connections with respect to other model fragments, small additions or removals of blocks or lines, and the variations found in Type 2 clones.

III. Simulink Model Mutation Classes

The mutation operators necessary for a model-clone detector comparison framework must test variations of all three model-clone types. That is to say, the Simulink mutation classes should contain instances of mutation operators that, once instantiated in the framework, will yield model clones representing all three types. In addition, we aim for mutation classes that are realistic edit scenarios as validated through our case study presented later.

Based on our experience with Simulink developing Simone, which included the creation of a Simulink grammar; and our Simulink clone class evolution study [9]; we devised our initial high-level categories of Simulink model mutations and outlined them [6]. In this paper, we further elucidate the categories, and define mutation classes belonging to each category. These categories and the classes themselves were based on our observations of model edit operations we encountered in a wide variety of Simulink models, including large public and private-industrial sets.

We present and organize the mutation classes organized by category. For each mutation class, we include 1) a description, 2) justification of its suitability, 3) and an example of a mutation (operator) belonging to the respective mutation class, including an image. For the examples, we manually mutate existing models to make the concepts as clear as possible. We present the models and their mutants in the native Simulink GUI because this is the form most familiar to Simulink modelers and it makes the mutants easily reproducible.

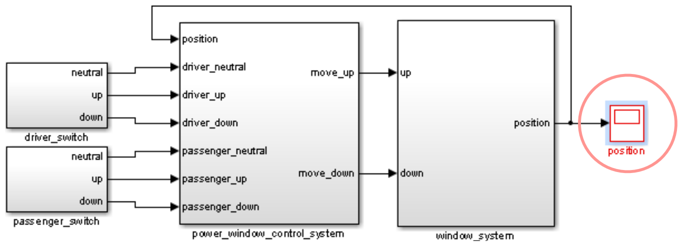


Fig. 1. Example of an mMLA Mutation Operator

Table I provides a listing of the mutation classes in our taxonomy, including the class' respective section in the paper, and the resulting type of model clone a mutation from the class will generate.

A. *Changing the Layout and Ordering of Elements*

This category of mutation classes contain mutation operators that are related to layout and presentation aspects of a model. Mutations belonging to these classes would enable detection of Type 1 model clones. When developing Simone, we found accounting for and filtering this information out improved recall for both exact and near-miss clones [4]. In addition, we also noticed that the ordering of elements in the textual representation of identical models may differ, so we have class of mutations accounting for that.

1) Modification of Layout Attribute (mMLA): Elements in Simulink contain different properties pertaining to presentation that should be filtered for clone detection as they do not impact system structure or behavior. As per the definition of Type 1 model clones, any systems that are identical regardless of these properties should be considered exact clones. As such we must inject mutants that are identical to an existing system but have some layout attribute differences. This includes differences in colour, position, size, and other layout attributes.

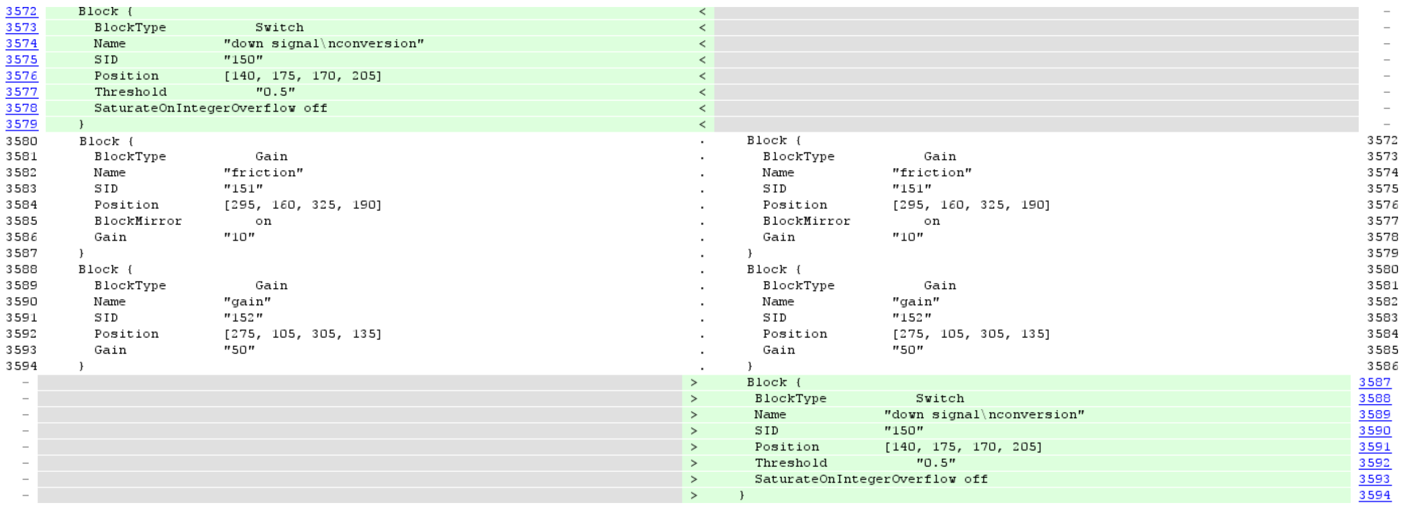
Layout edits are a reasonable edit/evolution operation as models may be refactored in this way to improve readability and comprehension, implement updates to company standards, and other related cases.

For our example mutation operator from this class, we choose to change the foreground colour of a block. We start with the original version of the Power Window model from the automotive demonstration set, which comes with Simulink. Specifically, we modify the root *powerwindow* system within that model by changing the foreground colour to red of a block of type Scope, entitled *Position*. The resulting mutant is pictured in Figure 1. In this case, the highlighted *Position* block was black until we mutated it to be red, as demonstrated in the figure.

2) Reordering Underlying Elements (mRUE): During our development of Simone, in which we looked at the underlying textual representations of the models, we noticed that in some

TABLE I. SIMULINK MUTATION CLASSES

Mutation Key	Title	Section	Clone Type
mMLA	Modification of Layout Attribute	III-A1	Type 1
mRUE	Reordering Underlying Elements	III-A2	
mRBL	Renaming a Block or Line	III-B1	Type 2
mCBV	Changing a Block's Value	III-B2	
mADBD	Add or Delete Block as Destination	III-C1	Type 3
mADBS	Add or Delete Block as Source	III-C2	
mCBT	Changing a Block's Type	III-C3	
mCSCH	Changing a Subsystem's clone hierarchy	III-C4	



```
Block {                                   <
  BlockType          Switch               <
  Name           "down signal\nconversion"<
  SID            "150"                    <
  Position       [140, 175, 170, 205]     <
  Threshold          "0.5"                <
  SaturateOnIntegerOverflow off           <
}                                         <
Block {                                   .   Block {
  BlockType          Gain                 .     BlockType          Gain
  Name           "friction"               .     Name           "friction"
  SID            "151"                    .     SID            "151"
  Position       [295, 160, 325, 190]     .     Position       [295, 160, 325, 190]
  BlockMirror        on                   .     BlockMirror        on
  Gain           "10"                     .     Gain           "10"
}                                         .   }
Block {                                   .   Block {
  BlockType          Gain                 .     BlockType          Gain
  Name           "gain"                   .     Name           "gain"
  SID            "152"                    .     SID            "152"
  Position       [275, 105, 305, 135]     .     Position       [275, 105, 305, 135]
  Gain           "50"                     .     Gain           "50"
}                                         .   }
                                          >   Block {
                                          >    BlockType          Switch
                                          >    Name           "down signal\nconversion"
                                          >    SID            "150"
                                          >    Position       [140, 175, 170, 205]
                                          >    Threshold          "0.5"
                                          >    SaturateOnIntegerOverflow off
                                          >   }
```

Fig. 2. Example of an mRUE Mutation Operator

cases the ordering of subsystems were not the same, even in identical systems. If this is the case, then a model-clone detector should be able to account for this and still identify such clones as Type 1. Thus, we need a class of mutants that are identical to existing systems but have their elements; such as, blocks, lines, ports, and branches; reordered textually.

We saw instances of this mutation class in multiple places, including the examples presented by us previously [4]. In terms of testing model-clone detectors, a mutant of this variety would only fail to be killed on text-based model-clone detectors, like Simone, as graph-based ones do not use the text representations. Thus, a mutant instance from this class would, correctly, be killed for all graph-based detectors and be a valid test for text-based ones. From a general perspective, this type of change in a system can sometimes occur when blocks are either added or deleted.

Continuing with the first version of the Power Window model, we this time choose to mutate the textual representation of the *window_system*. This example is demonstrated in Figure 2, which shows an excerpt of the text for the mutant on the right. This example mutation operator shifts the text representing the block named "down signal\n conversion" below 2 Gain blocks. The two models are structurally and semantically the same, however, any Simulink model-clone detection that uses the text, has to account for the variance represented by this mutant if they are to properly detect a Type 1 clone. In this case, we do not show a model image of this mutation as the mutant is visually identically to the original.

B. *Renaming and Value Modification of Elements*

This category of mutation classes are those that deal with variations in the names and values of the Simulink model elements. From a model cloning perspective, the model clones generated by mutations within this category are Type 2.

1) Renaming a Block or Line (mRBL): Each Simulink block has a name associated with it. In addition, "All block names in a model must be unique and must contain at least one character."[2] Although, used much less often and not necessary for a functional model, lines can also have names associated with them. These line names can be modified by changing the "Name" attribute of the line element itself, or changing the "PropagatedName" or "SignalName" attributes of an associated source or destination block.

Any model-clone detector that is capable of detecting Type 2 clones, should be able to identify clones that have blocks or lines having different names but sharing the same BlockType or LineType, respectively. A mutation that addresses this, must duplicate the system entirely and rename a single block or line.

To showcase this mutation class, we present a mutation operator that modifies a single block's name. Continuing with the *window_system* subsystem from our previous example, we this time rename an integrator block named "window position" to "window position\n RENAMED". The mutant is shown in Figure 3. All other elements within the model are the same; the mutated block is of the same type, and its connections remain intact. The only change is the block name.

[2]mathworks.com/help/simulink/ug/changing-a-blocks-appearance.html

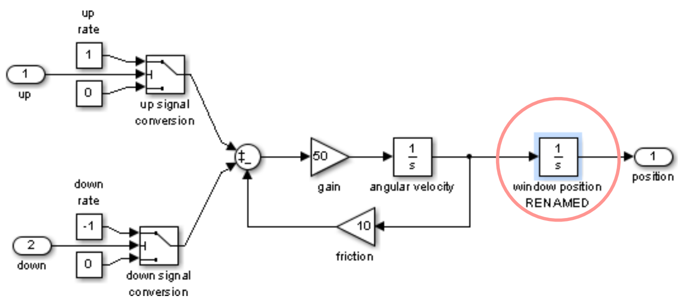


Fig. 3. Example of an mRBL Mutation Operator

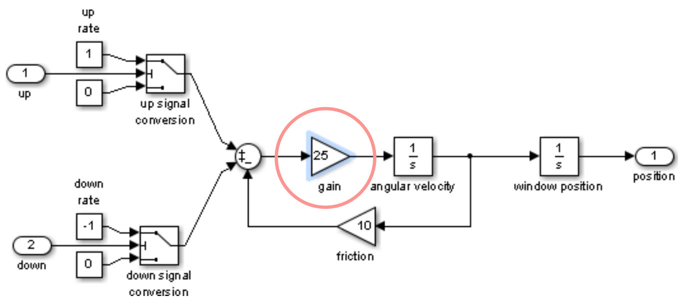


Fig. 4. Example of an mCBV Mutation Operator

2) Changing a Block's Value (mCBV): Simulink blocks can be configured through parameters, or values, that dictate specific aspects of the simulation. The simplest example is a "Constant" block, which outputs the constant specified by its "Constant Value" parameter. There are significantly more complex values that can be configured for more complex blocks including amplitude, relational operators, wave forms, signal delays, dialog parameters, and much more.

This class of mutations is important as value changes may not structurally modify the model, however, they likely, but not always, represent semantic changes in the model. It is analogous to the mutation of parameters in code, such as method and class parameter mutations [2]. From a model-clone perspective, mutation operators belonging to this class will introduce Type 2 clones as the systems will be identical except for a change in the values in a block.

A sample mutation operator from this mutation class is presented in Figure 4. Using the original version of the *window_system* subsystem, we introduce a mutant that has had a value modification occur to the highlighted Gain block, which is a block that multiples the input by the scalar, vector, or matrix parameter represented by the value[3]. Specifically, the example mutation operator demonstrated changes the value of the Gain block from 50 to 25.

C. Change Subsystems Structure

So far, the mutation categories presented have addressed non-structural changes to systems. This category looks at mutation classes that involve changing the structure of a Simulink system. Mutation operators belonging to these classes inject mutants that can test a model-clone detector's ability to discover Type 3 model clones. Mutations in this category should take into account the preservation of model connectivity, when necessary, to ensure a valid model.

1) Add or Delete Block as Destination (mADBD): This class of mutations involves adding or deleting a block as a destination block, with respect to an existing block in the system. This includes sink blocks[4], lines (signals), and required ports.

From a suitability perspective, changing a destination block within a subsystem is a likely case and can happen for a multitude of reasons, most of which are related to changes in the desired semantics of the simulation.

To demonstrate this mutation class, we provide two examples of mutations in Figure 5. Both of the examples, independently, mutate the "power_window_control_system" subsystem, which is also from the original version of the PowerWindow model we have been using thus far. In the left part of the figure, we illustrate a mutant that has the highlighted Scope (sink) block, *NEW DESTINATION BLOCK*, added to the system. Since every destination block must be connected to something in a valid model, we simply branched the output coming from the *validate_passenger* subsystem. This has much less impact on the model than creating a new output signal and port in a respective source block or source subsystem. The right side of the figure presents a mutant that has had its sink block *move_down* removed from the original subsystem from the highlighted part on the right side of the figure. In this case, it is not necessary to make any additional changes to the system to make it valid with respect to connectivity. Thus, we can have an unused port, as seen with the *moveDown* port in the figure. In some cases, we may have to check one level up to see if there is a higher level signal that used that outport and delete that signal only.

2) Add or Delete Block as Source (mADBS): Similar to mADBD, this mutation class includes both adding and deleting a block, but in this class, it involves operations on blocks that are a source block with respect to another block in the subsystem. Source blocks[5], required ports, and lines (signals) are included in mutations found in this class.

Changing the source blocks within a system are likely a common operation within a Simulink project as it is a key and relatively straightforward way of updating the simulation semantics, simply by updating the structure.

For this mutation class, we once again demonstrate it, in Figure 6, with two separate mutants of the original version of the "power_window_control_system" subsystem. In the example on the left, we simply add a new block called "NEW\n SOURCE BLOCK" of type Constant, which is highlighted in the figure and has a value of 18. Because it is being added as a source block, we connect it to the remainder of the model, including its corresponding lines, and add a new port in the subsystem "detect_obstacle_endstop" to connect this block to. For the deleting example, on the right, we illustrate a mutation that removes the "passenger_down" inport from the original version of the subsystem from the highlighted area in the figure. In this case, we removed the source block with the highest numbered port. Had we removed another one,

[3]mathworks.com/help/simulink/slref/gain.html

[4]mathworks.com/help/simulink/sinks.html

[5]mathworks.com/help/simulink/sources.html

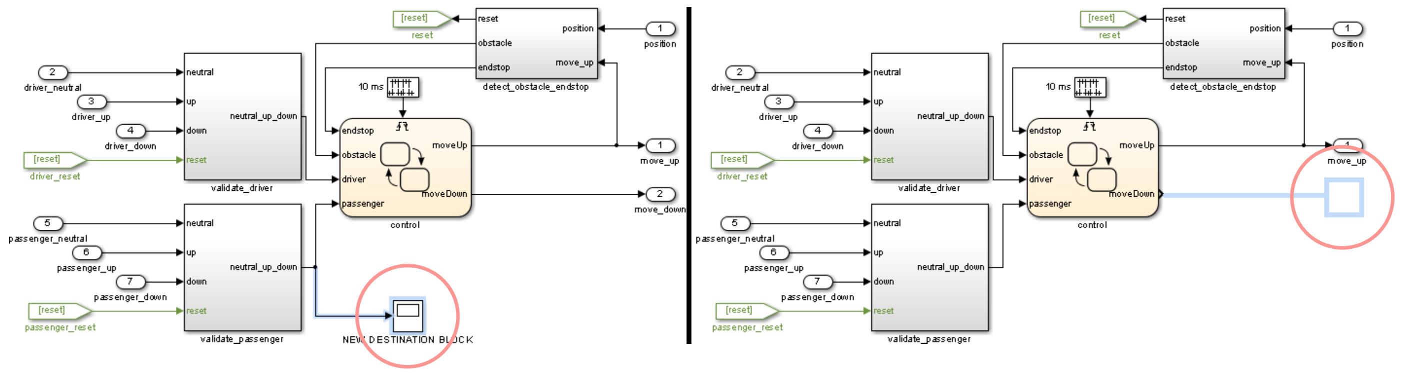


Fig. 5. Examples of Adding (left) or Deleting(right) Block as Destination (mADBD) Mutation Operators

depending on how the mutation operator was implemented, the other inports may have automatically had their port numbers adjusted to be sequential, which would have a more significant impact on model clone detection.

3) Changing a Block's Type (mCBT): Each Simulink block has a type, which has properties and actions associated with it. Changing a type is relatively significant and essential to a block's identity. For example, model-clone detectors, including Simone and CloneDetective [7], consider blocks with different types to be non-equivalent. As such, it is clear that there is a need for a class of mutations that account for this. In contrast to the mADBD and mADBS mutation classes, a mutation operator belonging to this class would modify only a block's type and leave the name intact. If both were changed, it would belong to either the mADBD or mADBS mutation classes, as it would essentially be deleting and adding a (different) block.

Changing a block's type is a realistic edit scenario as it is a quick way to change or tweak a system's functionality. Examples of this can include updated library blocks containing new and improved types, behavior correction, and other related refactoring tasks. In many of these cases, it is possible that, rather than changing a subsystem's structure and layout, it is just easier to change a block's type.

In order to demonstrate a sample mutation belonging to this mutation class, we once again modify the original version of the *window_system* subsystem in Figure 7. Specifically, we change the type of the highlighted block named *friction* from a Gain block, as pictured on the left side of the figure, to a Sqrt (square root) block in order to have friction simulated using the square root of the input rather than a constant multiplier. As explained, previously, this mutation modifies only the type of the block.

4) Changing a subsystem's clone hierarchy (mCSCH): This class of mutations involves mutation instances that mimic a batch of edit operations that are done to refactor model elements from a system into a subsystem. Although it is technically a batch of operations, it is such a common process that there is even an option to do it automatically, in one step, in the Simulink UI simply by selecting a group of blocks[6]. The key defining characteristic of this mutation operator, is that all of the model elements being refactored into a subsystem are completely unchanged.

This is a suitable mutation class in that refactoring groups of blocks into reusable subsystems is one of the more useful aspects of Simulink. Through the use and creation of block libraries and sublibraries [7], extracting and using subsystems in this manner is very plausible. From a model-clone detection perspective, this mutation class is important, because it will help test whether or not a model clone detector can properly account for subsystem boundaries.

We present an example mutation from this class in Figure 8. In this case, we mutate the system by taking four blocks from the middle of the original version of the *window_system* subsystem, and extract them into the highlighted subsystem *"Subsystem"* in the right part of the diagram. The four blocks, which are highlighted in the left part of the figure, include a Sum block, two Gain blocks, and an Integrator block. In order to do this, however, the newly created subsystem must have the proper amount of inports and outports created and connected to the upper-level system. The extracted subsystem, not shown in this diagram, contains the four blocks and includes the appropriate connections. The first Sum block, "window input" takes input from two inports, and the "angular velocity" integrator block is connected to the newly created outport.

IV. Applying the Taxonomy to Simulink Projects

Our taxonomy was developed with the goal of having mutation classes that result in variations of the different model-clone types. However, having mutation classes that represent realistic edit scenarios that occur in actual MDE projects is important to both the testing of model-clone detectors and the generality of our Simulink model mutation taxonomy. So, in order to validate our choice of mutation classes, we consider both publicly available models and private models from our industrial partners to perform a case study. That is, we will be looking at these particular systems and analyzing them by seeing how the witnessed edit operations across versions correspond to the mutation classes in our Simulink model mutation taxonomy. As we noted in our model clone evolution study [9], model clones are fairly representative of an MDE project in general, as is the evolution of such clones. So; rather than exhaustively, and rather unfeasibly, consider every system and subsystem within a project; we look for mutation instances that occur from one version to the next in any subsystems that

[6]mathworks.com/help/simulink/ug/creating-subsystems.html#f4-7371

[7]mathworks.com/help/simulink/ug/creating-block-libraries.html

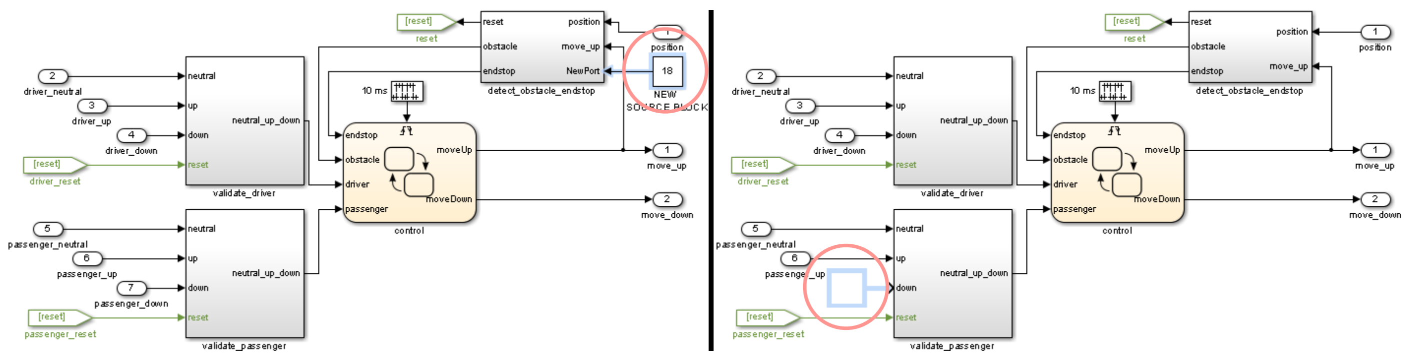


Fig. 6. Examples of Adding (left) or Deleting(right) Block as Source (mADBS) Mutation Operators

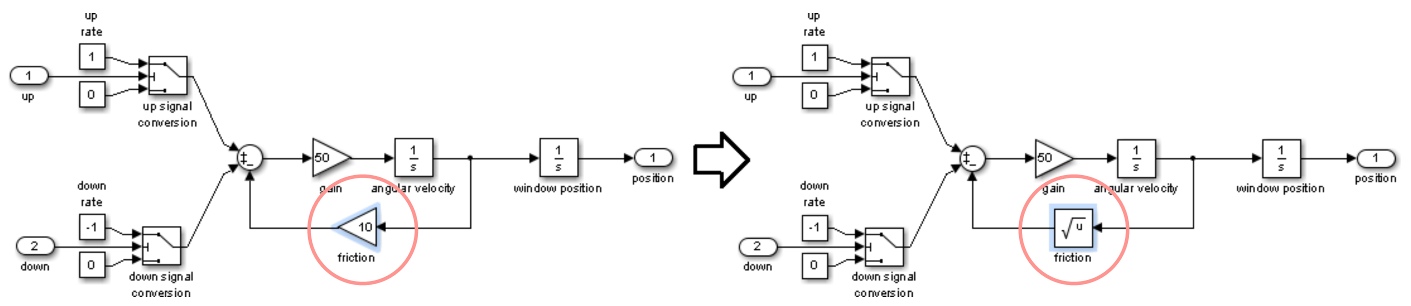


Fig. 7. Example of an mCBT Mutation Operator

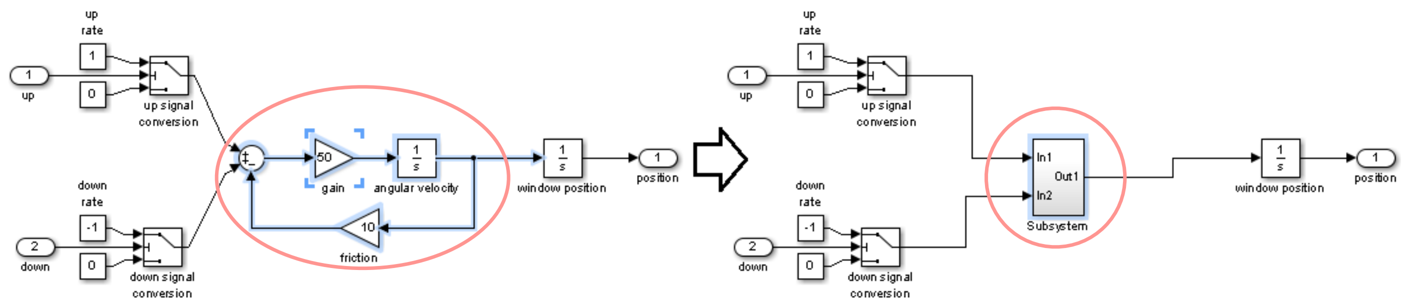


Fig. 8. Example of an mCSCH Mutation Operator

have been identified as belonging to a clone set in the original version. This is sufficient as we are simply trying to exhibit that these mutations do, in fact, exist within projects and can be classified into one of our proposed mutation classes. Thus, we will also note if there are any mutations that occur that do not belong to our categories and elaborate on them.

In this case study, we consider both publicly available models and private models from our industrial partners. The public models include the **Automotive Power Window** (PW) System that comes with the Simulink example set and a large open-source **Advanced Vehicle Simulator**(AVS)[8]. Table II displays statistics about the projects. The last column, Model Clone Classes (MCC), demonstrates the number of model clone classes discovered in each of the three projects using Simone with our best-fit [4] settings of 70% similarity and blind-renaming. As shown, the PW system is a smaller, compact, and simple system. AVS is quite large and complex, and has more MCCs than our industrial system set. Thus, we believe it is a fairly representative and rich system.

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

TABLE II. PROJECTS UNDER STUDY. MODIFIED FROM [9]

Project	Version #	Model Files	SubSystems	MCCs
PW	1	1	18	5
	2	1	29	5
	3	1	33	6
	4	1	25	4
	5	1	45	6
AVS	r0000	69	861	18
	r0080	69	1621	35
	r0116	72	1714	38
Industrial Set	55	9	977	20
	56	9	977	21
	57	9	986	23
	58	9	1091	30

In order to look for evidence of mutation operators in these projects, we use Simulink's model XML comparison tool[9], which is part of the Simulink Report Generator package. While performing model comparison using XML comparison has its drawbacks [10], [11], this tool was very useful for our purposes when combined with our domain knowledge and some manual

[9]mathworks.com/help/rptgenext/ug/how-to-compare-xml-files-exported-from-simulink-models.html

interaction/tweaking. For example, things that clearly should have matched with one another, did not because of innate XML hierarchy issues. Fortunately, albeit it non-trivial, we could manually enumerate and classify the instances of mutations witnessed within the models as we went through it.

Using the following filtration settings, we were able to trace through the different project versions, and tally and classify the mutation instances as we encountered them[9]:

Do not filter - Nonfunctional changes
By default, the comparison tool ignores all tags that are considered non-functional, including positions, fonts, colors, and more. We want this comparison to be included because this will allow us to detect layout mutations.

Do not filter - Changes in lines
Changes in lines are often indicative of changes in source or destination blocks, which is information we want.

Filter - Changes in the graphical interface
This information is a summary of inports and outports at the top level of the model. While we are interested in ports, as values, this information is reported at the higher level as a block‘s value, so we do not need this information in this form for the comparison.

Filter - Changes in block parameter defaults
Because changes in blocks are exhibited as functional changes and default usage will be represented uniformly across versions, we do not need to know if the default values have changed.

A helpful feature of the Simulink Report Generator tool is each element has all of its sub elements contain a property called *ZOrder* that indicates the sub elements' location within the element's textual representation. So, if a matching element is out of order with respect to its containing element, we would see that as a difference within the comparison tool. This was perfect for identifying instances of mutations where an underlying textual ordering change had occurred.

We apply this approach to the three projects individually and discuss them as such below. So, for each subsystem that was discovered within a clone class, we traced that subsystem across all versions of the project, and counted and classified all mutations. There were a few clone classes that contained a significant amount of systems, for example there were some classes that contained thirty seven, seventy one, and even 222 systems. Because this process was mostly manual and we needed only evidence of existence of the mutations as edit operations, we decided to consider only a random sampling of a quarter of the systems belonging to clone classes larger than twenty systems.

We present a table summarizing our findings for each project in each of the respective project sections; IV-A,IV-B, and IV-C; and present any interesting cases we encountered. In each table, the column "# Of Times a System Changed" represents each time that at least one change was witnessed from one version of a subsystem to the next. For the subsequent columns, each value in the "Total Count" row represents the number of times a specific instance of a mutation corresponding to the column's mutation class was witnessed from one version of a subsystem to the next. The subsequent row presents that information as a ratio of the number of times the specific mutation class was observed with respect to the total number of times a system change was observed. The purpose of the "Other" column is to illustrate if there were any model edits that were unclassifiable with respect to our taxonomy.

A. Case 1: Application to the PW Project

The original version of the PW project contained eleven subsystems that were spread across five clone classes. We traced those subsystems across five versions and classified and counted the instances of mutations, yielding the results presented in Table III. As noted in the table, each mutation class was represented in some form among the system edits. The layout mutation class (mMLA) and source block mutation (mADBS) were present in more than half of the cases when a system was changed from one version to the next. All of mutation classes were observed in at least one fifth of the system traces that involved changes. In addition, no edit operations were observed in this project that could not be classified using our taxonomy.

In this project, we observed a number of edit operations that could be classified as mCSCH instances. One such example of this was the modification from the third version of the "window_system" into the fourth version. Each version is illustrated in Figure 9. What we specifically witnessed was all the highlight blocks and lines in the left part of the diagram from version three were extracted and placed into a subsystem entitled *"process"*. This subsystem was then used in the fourth version, as highlighted in the right part of the figure.

B. Case 2: Application to the AVS Project

Our model-clone analysis of the first version of the AVS project discovered 281 subsystems that belonged to eighteen different model clone classes. After taking a random quarter of the subsystems belonging to larger clone classes, as discussed before, we considered 132 subsystems and how they changed across three versions of this project. The summarized results are in Table IV. As we soon discovered with this open-source system, the majority of changes across versions were not model-based changes but changes to Matlab simulation code. However, there were some model-based changes witnessed. The vast majority of times a subsystem was changed, a value/parameter change was observed. Also, more than half of the subsystems that changed across versions included a change in layout.

An example of one of these documented mCBV mutation instances in the AVS project is demonstrated in Figure 10, where the system "lib_controls|<vc>par auto s/a", from the model "models/library/lib_controls.mdl" is presented in the Simulink Report Generator tool after it has been configured as we discussed earlier. In this case, the block *key_on* has had its "Mask Initialization"[10] value changed.

C. Case 3: Application to the Industrial Project

The first iteration of our industrial project contained 217 subsystems that were contained in twenty clone classes. After

[10]www.mathworks.com/help/simulink/ug/initialize-mask.html

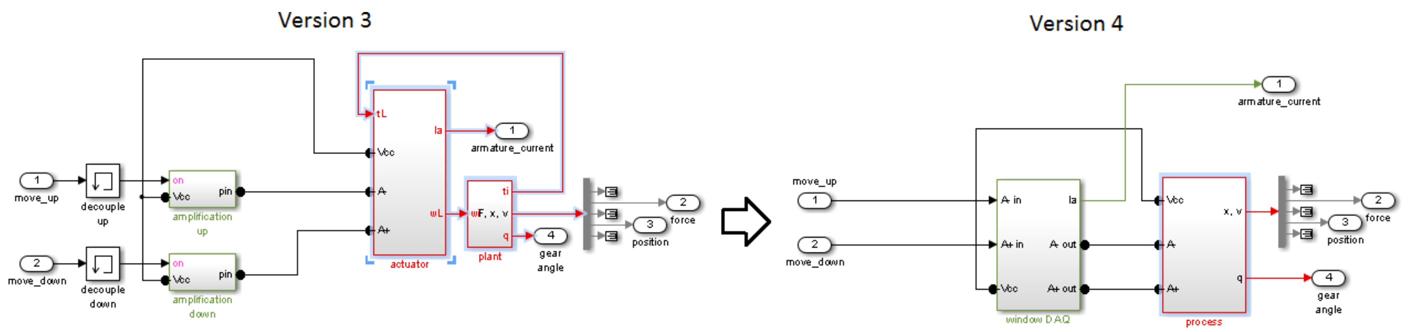


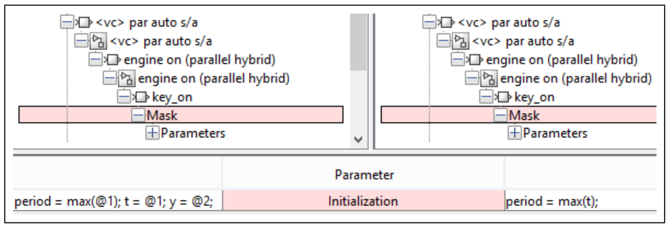
Fig. 9. Version Three and Four of the PW Project “window_system” Subsystem

TABLE III. SUMMARY OF MUTATION INSTANCES WITNESSED IN THE PW PROJECT

	# Of Times a System Changed	mMLA	mRUE	mRBL	mCBV	mADBD	mADBS	mCBT	mCSCH	Other
Total Count	15	10	7	4	4	6	8	3	3	0
% of Total # of System Changes	-	66.67%	46.67%	26.67%	26.67%	40.00%	53.33%	20.00%	20.00%	0.00%

TABLE IV. SUMMARY OF MUTATION INSTANCES WITNESSED IN THE AVS PROJECT

	# Of Times a System Changed	mMLA	mRUE	mRBL	mCBV	mADBD	mADBS	mCBT	mCSCH	Other
Total Count	50	28	0	0	48	1	0	0	0	0
% of Total # of System Changes	-	56.00%	0.00%	0.00%	96.00%	2.00%	0.00%	0.00%	0.00%	0.00%


<vc> par auto s/a
<vc> par auto s/a
engine on (parallel hybrid)
engine on (parallel hybrid)
key_on
Mask
Parameters

<vc> par auto s/a
<vc> par auto s/a
engine on (parallel hybrid)
engine on (parallel hybrid)
key_on
Mask
Parameters

period = max(@1); t = @1; y = @2;	Parameter Initialization	period = max(t);

Fig. 10. mCBV Mutation Operator Observed in AVS Project

considering only a random quarter of the subsystems from larger clone classes, we focused on 102 subsystems. Looking at how these subsystems evolved from version fifty five to version fifty eight led to the results shown in Table V. This project, in contrast to the AVS project, had many model edit operations that occurred from one version to the next. Similarly to the AVS project, the modification of layout attributes and the changing of values were witnessed in more than half of the instances where a subsystem was updated. Other edit operations that corresponded to mutations that were witnessed in roughly a third or more of subsystems that were changed included reordering the textual representations; renaming blocks or lines; and adding and deleting blocks, both as sources and destinations. There was one instance where we saw an edit operation that corresponded to a subsystem hierarchy change. Lastly, there were a small number of subsystem changes that we were unable to classify using our taxonomy. We discuss these in Section IV-D.

D. Discussion

Overall, the significant majority of edit operations we found in these projects could be sorted into the classes proposed in our taxonomy. In terms of each mutation class’ prevalence with respect to the total number of times a subsystem was changed, most of the classes were well represented. However, changing a block’s type (mCBT) and changing a subsystem’s hierarchy (mCSCH) were underwhelmingly present in the projects in our case study. We still believe these two are suitable classes for the reasons provided in our original definitions of them and hope it was just a function of the projects we were investigating.

One instance of an edit operation that could not be classified in our taxonomy was the addition, modification, or removal of a standalone and unconnected *Annotation* block. In regards to both testing model-clone detectors and the generality of our taxonomy, this operation is not significant nor semantic, as it is essentially documentation[11]. A graph-based model-clone detector would likely disregard it and a text-based detector would include it, but since it is only a single block, it would affect the similarity only slightly. The remaining occurrences of edit operations that were unclassifiable according to our taxonomy were a small number of additions or deletions of a standalone and unconnected block of type *Reference* that was added or removed. This type of block is essentially a maintenance hyperlink in that it is something for engineers to look at when using or changing the system containing this reference block and is used by Simulink to refer to other libraries as a place holder. Again, the impact to testing model-clone detectors would be quite trivial and, from a general perspective, the block is just an unconnected library block linking to other models.

The obvious threat to validity in this case study is that this process was performed semi-automatically, rather than fully automatically. It would be ideal if this process could be automated, as discussed in Section VI. However, because we are looking only for evidence of the existence of edit operations that can be classified using our taxonomy, it is acceptable if we missed some. The only issue is if we missed unclassifiable edit operations belonging to the “Other” class. We made an effort to investigate every single edit operation for each subsystem, so we believe that none were missed.

Another threat to validity is our choice and availability of Simulink models available to us. The open-source projects

[11]http://www.mathworks.com/help/simulink/ug/annotating-diagrams.html

TABLE V. SUMMARY OF MUTATION INSTANCES WITNESSED IN THE INDUSTRIAL PROJECT

	# Of Times a System Changed	mMLA	mRUE	mRBL	mCBV	mADBD	mADBS	mCBT	mCSCH	Other
Total Count	83	67	38	27	50	32	32	0	1	3
% of Total # of System Changes	-	80.72%	45.78%	32.53%	60.24%	38.55%	38.55%	0.00%	1.20%	3.61%

were ideal because they both had multiple versions and could be shared publicly. The industrial project is a rather rich and real-life MDE project in use, so we believe that is a very strong example. However, we claim only that this is a starting point for a taxonomy of Simulink model mutations. It would, of course, be great to see the applicability of our taxonomy to further industrial projects as mentioned in Section VI

V. RELATED WORK

Model transformations, which involve going from a source model to specific target model, are generally related to the idea of model mutations and the corresponding edit operations we were looking for in our case study. A key difference between what we did in our case study and what is done in the majority of model transformation work, is we were retroactively looking at model evolution to see what operations had occurred, whereas the model transformation area is more focused on forward engineering and prescriptive. An example of this is the work done by Sen and Baudry [12], where they use graph grammers on meta models to develop model transformations. Simulink model transformations are discussed by Tran et al. [13], who attempt to employ them for the purposes of Simulink refactoring. They have operations that include adding, copying, replacing, and deleting blocks and use these to devise composite operations. The relation between their work and our work, is that once their transformations have been applied, we would be able to classify them according to our taxonomy. Also worth mentioning is the mutation work done on the ATL model transformation language by Khan and Hassine [14]. They devise mutation operators for that language in order to detect inadequacies in programmed model transformations. The key difference here is their work is more related to code-based mutation work than it is model-based mutations.

While research on model-based mutations is still relatively newer than its source-code counterpart, there is still some work of note. Trakhtenbrot [15] introduces model mutations for state charts. This work is not directly applicable to Simulink models, however, there are Simulink Stateflow[12] blocks, which are state charts. This work may be applicable to those blocks, in isolation. Adra and McMinn [16] developed mutations intended for agent-based models, while Bartel et al. [17] develop a model-mutation based framework for testing adaptive systems. In both of these cases, the mutations themselves are derived at in a model-driven fashion and later transformed to text for test suites. Other than our mRUE mutation, all of our proposed mutations in our taxonomy are purely model-driven and can be tested as such, as we outlined in our original presentation of our framework [6].

The notion of Simulink model mutations have been addressed previously by Zhan and Clark [18], He et al. [19], and Araujo et al. [20]. In these examples, they describe mutations that explicitly try to mutate a model's run-time properties. That is, their mutation operators are concerned with modifying the signal carried on wires between blocks only. In contrast, our proposed mutation taxonomy considers both design-time properties, which are necessary for model-clone detection testing, in addition to some run-time properties, like value changes. Neither of them purpose a taxonomy, per se, however, Zhan and Clark identify three categories of signal mutations: Add, Multiply, and Assign, representing signal addition, multiplication, or specific value assignment, respectively. The mutation operators they purpose to accomplish those signal mutations can be classified using our taxonomy.

Lastly, our model-clone detection framework is a direct extension of the work done by Roy and Cordy [21], [22] in which they proposed a framework for testing code-clone detectors that was based on mutations. Our work is contrasted with theirs in our initial framework outline [6], with key differences including the representation and implementation of the mutation operators, the different nature of the model-clone detectors, the output format of the resulting model clones, and other differences that are specific to the modeling domain. The Roy and Cordy framework used much existing research on code mutations and "realistic" programmer edit scenarios to explicate and utilize its mutation taxonomy, whereas, we had no relevant existing studies pertaining to Simulink model mutation or edit operations, and, thus, carried out the work presented in this paper.

VI. FUTURE WORK

The first area of future work involves completely automating the process that was performed semi-automatically in our case study. Because there are issues with the Simulink Report Generator XML comparison tool, we would have to use the provided API, because the internal workings are proprietary, and perform quite a bit of tweaking on the output of the tool to try to account for the errors. We would then need to programmatically represent the mutation classes and create detection algorithms for each of them, which would include definitions of a more formal nature than we have provided above. As mentioned in our discussion, we held off on doing that for now as we needed only to see if our taxonomy was applicable in the various MDE projects we had access to for the purpose of our mutation-based model-clone detection evaluation framework.

The second area of future work, which we are currently conducting, involves implementing the mutation operators so that they can be utilized in our evaluation framework. This step involves properly choosing where to inject the model mutants; and how this will be done, which includes the complete and precise rules and form the mutation operators will have. So far, we plan on representing the mutations as Matlab model functions[13], possibly in conjunction with TXL [23] for the reordering text mutation (mRUE). By using this proposed taxonomy, we can ensure that we have chosen the right

[12]mathworks.com/products/stateflow/

[13]mathworks.com/help/simulink/functionlist.html

selection and quantity of mutation operators for our evaluation framework.

Lastly, it would be beneficial to have access to more multi-version Simulink projects, especially from industry. This would allow us to further validate our proposed taxonomy and would help improve its generality.

VII. Conclusion

In this paper, we contributed a Simulink model mutation taxonomy that is primarily intended to facilitate and guide creation of mutation operators for a model-clone detection comparison framework. The original taxonomy was based on our experiences developing our Simulink model-comparison tool, creation of a Simulink grammar, and a Simulink clone-class evolution study. It included mutation classes that were constructed in such a way that mutation operators belonging to each class would facilitate testing various model-clone types. We then attempted to validate our taxonomy by means of a case study that used the taxonomy to classify edit operations witness in multiple versions of three Simulink projects. Specifically, we selected subsystems within a project that were discovered in clone classes, and contrasted those subsystems from one version to the next. For each project, we presented a table summarizing the number of times an edit operation corresponded to one of our proposed mutation classes and also compared these tallies to the total number of times a subsystem was modified.

Overall, each of the mutation classes were represented among the edit operations observed in the three projects, albeit some more than others. Layout (mMLA) and value changes (mCBV) were the most prevalent, while changing a block's type (mCBT) and a subsystem's hierarchy (mSCS) were the least common. There was a very small number of edit operations observed that could not be classified using our taxonomy, however, these were relatively trivial model elements that were unconnected to the systems and mostly related to documentation and maintenance.

In the future, we will be looking into automating the process executed in our case study, although there are a few non-trivial obstacles to overcome in doing so. Currently, we are working on implementing the mutation operators within our model-clone detection evaluation framework based on this proposed taxonomy by representing them in both Matlab model modification functions and text transformations. It would also be ideal if we could further validate this taxonomy by having access to more industrial Simulink model sets that have multiple versions.

While our taxonomy was developed with a specific goal in mind, we hope that it is fairly general to Simulink model mutations and that our taxonomy provides a solid starting foundation for future Simulink, and data-flow, model mutation research.

Acknowledgment

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

[2] J. Bradbury, J. Cordy, and J. Dingel, "Mutation operators for concurrent Java (J2SE 5.0)," in *International Workshop on Mutation Analysis*, 2006, pp. 57–62.

[3] H. Dan and R. M. Hierons, "SMT-C: A semantic mutation testing tools for C," in *International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)*, 2012, pp. 654–663.

[4] 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.

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

[6] ——, "Using mutation analysis for a model-clone detector comparison framework," in *NEIR track - ICSE*, 2013, pp. 1261–1264.

[7] 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.

[8] H. Storrle, "Towards clone detection in UML domain models," in *ECSA: Companion Volume*, 2010, pp. 285–293.

[9] 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.

[10] M. Stephan and J. R. Cordy, "A survey of methods and applications of model comparison," Queen's University, Tech. Rep. 2011-582 Rev. 3, 2012.

[11] ——, "A survey of model comparison approaches and applications," in *Modelsward 2013*, 2013, pp. 265–277.

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

[13] Q. M. Tran, B. Wilmes, and C. Dziobek, "Refactoring of Simulink diagrams via composition of transformation steps," in *International Conference on Software Engineering Advances*, 2013, pp. 140–145.

[14] Y. Khan and J. Hassine, "Mutation operators for the Atlas Transformation Language," in *International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)*, 2013, pp. 43–52.

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

[16] 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.

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

[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)*. IEEE, 2009, pp. 157–166.

[22] C. Roy, J. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.

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