A Mutation Analysis Based Model Clone Detector Evaluation Framework

by

MATTHEW STEPHAN

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen’s University
Kingston, Ontario, Canada
August 2014

Copyright © Matthew Stephan, 2014
Model-Driven Engineering is becoming increasingly prevalent and mature. As software projects developed through this methodology age, the need for analysis of Model-Driven projects becomes imperative. One form of analysis is Model Clone Detection, which involves finding similar or identical model fragments in a given context. There are a number of techniques intended for Model Clone Detection and for different types of models. One hindrance to the growth of this field is the ability to objectively and quantitatively compare different model clone detectors and settings of the same detector.

In this thesis, our original contribution to knowledge includes a framework utilizing Mutation Analysis to evaluate and compare model clone detectors. It is our proposition that, through distinguishing edit operations on models as mutations, we can create such a framework. In order to demonstrate the plausibility of our framework, we develop a Simulink implementation of the framework.

We begin by outlining our initial, qualitative, attempts evaluating our Simulink model clone detector. This includes challenges encountered that are addressed by our framework. We outline the framework and describe each step in its process in an example-driven manner through creation of a framework prototype that works on Simulink model clone detectors. We choose Simulink because it is the most
mature form of Model Clone Detection, it is of interest to our industrial partners, and we previously created a Simulink model clone detector. An additional contribution is a taxonomy of Simulink model mutations intended to inject the various types of model clones, while still being representative of realistic Simulink model evolution, which we verify through a case study. We run our Simulink framework prototype on leading Simulink clone detectors to ascertain their recall and precision. We observe high recall for Simone, lower recall for ConQAT because it is intended for only a subset of clone types, and high precision for both tools.

It is our hope that having such a framework in place will help facilitate gains in Model Clone Detection research as engineers in this area can now refine their own tools and new detectors can be compared against existing ones.
Acknowledgments

I would like to begin by acknowledging my family for all their patience and support during these past few years. You realized pretty early that it is a faux pas to ask how the research or thesis is going. To my father, Paul, thank you for being a wonderful example of professionalism and for showing me the importance of hard work. I do not think this is what you had in mind, but at least one of your children became a Doctor just like you. To my mother, Frankie, thank you for being a constant presence in my life and always being there for me. Your wisdom and caring made my journey possible. Also, thank you for proofreading this thesis. Thanks to my brothers Dave and Rick for building the deck for the new cottage while I was in my lab working on this thesis. Sorry I could not help.

Thank you to my fraternity of friends for making sure that I take time to enjoy life and relax by maintaining a good balance of work and leisure.

To my lab mates, Eric, Doug, Karolina, Paul, Gehan, and sometimes Scott. I appreciate you not making fun of me too much for my standing desk and when I do my exercises. It’s been a pleasure spending my afternoons in the lab with you. I want to thank my Ph.D. Committee, Dr. Dingel and Dr. Dean, for the help and guidance. To my examining committee, thanks for reading my thesis and agreeing to serve on my committee. I especially appreciate you taking time out of your busy schedules
and doing this during the summer.

Lastly, and most importantly, I would like to thank my Supervisor, Dr. James R. Cordy. As one of the creators of the Turing programming language, your academic influence on me began even before we met. Turing was the first programming language I learned and cultivated my interest in Computing. That, in conjunction with how enthusiastic and welcoming you were when I was choosing Ph.D. programs, led me to one of the best decisions of my life in joining your lab here at Queen’s. You really were the ideal mentor. You challenged me, yet gave me freedom. You had a literal open-door policy and I never felt uncomfortable or nervous coming to you with anything. It is a shame you will be retiring in the not-too-distant future as incoming generations of students will not have the privilege and honour of being supervised by you. I am excited to call you a colleague and for our collaborations in the future.

Thanks, Jim.

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.
Statement of Originality

These statements certify that, to the best of my knowledge, the content of this thesis is my own work. This thesis has not been submitted for any degree or other purposes. I certify that the intellectual content of this thesis is the product of my own original work and that all the assistance received in preparing this thesis and sources have been acknowledged in the thesis.

Part of Chapter 2 was published as the best student paper at the 2013 International Conference on Model-Driven Engineering and Software Development [40], and as a technical report [37]. Various parts of Chapter 4 appeared in the 2012 International Conference on Software Maintenance [4], the 2012 International Workshop on Software Clones [35], and the NIER track of the 2013 International Conference on Software Engineering [36]. Much of the material from Chapter 5, and some of Chapter 3, was published in the 2014 International Workshop on Mutation Analysis 2014 [38], and parts of Chapters 6 and 7 have been accepted in the Doctoral Symposium of the 2014 International Conference on Software Maintenance and Evolution [41].
## Contents

Abstract i  
Acknowledgments iii  
Statement of Originality v  
Contents vi  
List of Tables ix  
List of Figures x  

### Chapter 1: Introduction 1  
1.1 Motivation and Problem 2  
1.1.1 Thesis Statement 2  
1.2 Research Approach 3  
1.2.1 Scope 3  
1.3 Contributions 4  
1.4 Organization of Thesis 5  

### Chapter 2: Background 7  
2.1 Simulink 7  
2.2 Model Clone Detection 8  
2.2.1 Model Clone Types 11  
2.2.2 Graph-Matching Algorithms 14  
2.2.3 Text-based Approaches 16  
2.3 Mutation Analysis 18  
2.4 Model Evolution 20  
2.4.1 Model Variants and Software Model Product Lines 21  
2.5 Summary 22  

### Chapter 3: Related Work 23
Chapter 4: Framework Overview  
4.1 Preliminary Evaluation .......................... 31
4.2 Framework Design Challenges .................. 34
4.3 Framework Details ................................. 37
4.3.1 Mutation Phase ................................. 37
4.3.2 Execution and Evaluation Phase ............. 40
4.4 Summary ........................................... 41

Chapter 5: Mutators for Simulink Model Clone Injection  
5.1 A Taxonomy of Mutators for Simulink Models ...... 46
5.1.1 Changing the Layout and Ordering of Elements ... 47
5.1.2 Renaming and Value Modification of Elements .. 53
5.1.3 Change Subsystems Structure .................. 58
5.2 Validating the Taxonomy ............................ 80
5.2.1 Case 1: Application to the PW Project ........ 84
5.2.2 Case 2: Application to the AVS Project ........ 87
5.2.3 Case 3: Application to the Industrial Project ... 87
5.2.4 Discussion ....................................... 89
5.3 Summary ........................................... 92

Chapter 6: Framework Evaluation Phase  
6.1 Desired Clone Result Format ..................... 93
6.2 Persisting Mutant Metadata ...................... 96
6.3 Recall Calculation ................................. 98
6.4 Precision Calculation .............................. 99
6.5 Summary ........................................... 101

Chapter 7: Simulink Framework Implementation Results  
7.1 Specific Process for Prototype ................... 102
    7.1.1 Simone Report Transformation ............... 103
    7.1.2 ConQAT Report Transformation ................ 104
7.2 Experiment Models and Mutations ................ 106
7.3 ConQAT Results ................................... 107
7.4 Simone Results ................................... 111
List of Tables

5.1 Simulink Mutation Classes ........................................ 46
5.2 Projects Under Study ............................................. 81
5.3 Summary of Mutation Instances Witnessed in the PW Project .... 86
5.4 Summary of Mutation Instances Witnessed in the AVS Project .... 88
5.5 Summary of Mutation Instances Witnessed in the Industrial Project . 90

7.1 Models Mutated by Prototype for Tool Evaluation ................ 107
7.2 Results of ConQAT Evaluation .................................... 108
7.3 Results of Simone Evaluation ..................................... 112
7.4 Results of Simone Evaluation at 20% Difference .................. 115
List of Figures

2.1 Detect_Obstacle_Endstop Simulink System ........................................ 8
2.2 Model Clone Example ................................................................. 10
2.3 Type 1 Model Clone ................................................................. 11
2.4 Type 2 Model Clone ................................................................. 13
2.5 Type 3 Model Clone - New Block ................................................ 13
2.6 Type 3 Model Clone - Structural Change ........................................ 14
2.7 Normalised ConQAT Model Graph of Example Model Clone .................. 15
2.8 Textual Representation of a Simulink System .................................... 18
2.9 Code Mutation for Java Concurrency ............................................. 19
3.1 Example of a Semantic Clone ....................................................... 25
3.2 Simulink Model Signal Mutation ................................................. 27
3.3 Mutations for Code Clone Framework ............................................ 29
4.1 System Clone Classes Detected by SIMONE and ConQat ....................... 33
4.2 Type 3 Subsystem Clone Pair Discovered by Simone but not ConQAT ...... 34
4.3 Nested Clone Example .............................................................. 36
4.4 Mutation Phase with X systems, Y mutations, and Z variants ............... 39
4.5 Model-Clone Detection Comparison Framework Evaluation Process ...... 42
5.1 Example of an mMLA Mutation Operator .............................................. 48
5.2 Example of an mRUE Mutation Operator .............................................. 52
5.3 Example of an mRBL Mutation Operator .............................................. 54
5.4 Example of an mCBV Mutation Operator .............................................. 57
5.5 Adding or Deleting Block as Destination (mADBD) Mutation Operators 61
5.6 Adding or Deleting Block as Source (mADBS) Mutation Operators .... 65
5.7 Example of Adding Block in the Middle of a System (mADBS/mADBD) Mutation Operator ......................................................... 69
5.8 Example of Deleting Block in the Middle of a System (mADBS/-mADBD) Mutation Operator ......................................................... 71
5.9 Example of an mCBT Mutation Operator .............................................. 75
5.10 Example of an mCSCH Mutation Operator ............................................. 79
5.11 Version Three and Four of the PW Project window system Subsystem 85
5.12 mCBV Mutation Operator Observed in AVS Project ......................... 89

7.1 Prototype File Organization of Clone Reports and Mutant Metadata 104
7.2 Evaluation of ConQAT on Powerwindow/detect_endstop System ....... 109
7.3 Mutated CathodeOverpotential System from fc_KTH.lib model ......... 110
7.4 Evaluation of Simone on Powerwindow/detect_endstop System ....... 112
7.5 The SaturationTemperature System ...................................................... 114
7.6 Recall (%) of Each Tool Run ............................................................... 117
7.7 Precision (%) of Each Tool Run ........................................................... 118
Chapter 1

Introduction

Model-Driven Engineering (MDE) is becoming increasingly common in Software Engineering and is starting to experience substantial adoption. Communications, automotive, and other embedded software areas are all starting to employ MDE in various forms. As projects created through MDE begin to age and continually grow, the necessity of analysis on these systems becomes more important. One type of analysis, model clone detection, a relatively new research area, involves finding sets of software models that are identical or similar to one another with respect to some measure of similarity. The results from model clone detection can be used to reduce redundancy, extract patterns, aid in system understanding and refactoring, and identify similar system components after error detection [17]. Thus, it is of great interest to many in the Software Engineering community including our partners in industry.

There are small but growing number of approaches in existence that perform model clone detection [37, 40]. However it is not clear which approach is best suited to what situations, and how to compare tools or individual tools with different configurations. As such, there is a large need for a standard way of comparing different model clone detectors or the same detector using different tuning parameters [35], especially if the
research in this area is to evolve. Having a facility to do this will help fuel growth in
model clone detection and analysis, as it will provide researchers a means with which
they can try out and validate new techniques as well as compare their techniques to
existing ones.

1.1 Motivation and Problem

In previous work, we presented initial ideas for performing a qualitative evaluation
of Simulink model clone detection approaches [4, 35], motivated by our desire to
compare our new model clone detection approach to existing tools as well as optimize
our own tool. While we are able to perform some basic qualitative analysis and find
differences in the model clones detected by different tools and configurations, we still
are faced with a number of problems: Recall computation, the different nature of the
clones reported, and the different representations of the resulting clone classes and
pairs provided by each of the tools. We elaborate on these problems in detail later
in Chapter 4. If model clone detection is to flourish as an area of research and help
advance model driven software engineering, evaluation of these tools must be more
quantitative and automatic. In addition, by analyzing top candidate tools in the field,
one could identify areas for improvement and suggest future research directions for
model clone detection in general.

1.1.1 Thesis Statement

We propose that, by empirically characterizing model modifications as formal muta-
tions for mutation analysis, we can design a framework for evaluating model clone
detectors that is both objective and quantitative.
1.2 Research Approach

At a high-level we approach this problem and test our hypothesis by means of the following steps that will result in a prototype demonstrating the plausibility of the framework

1. Identify a corpus of Simulink mutation operators based on 1) tailoring mutation operators to cover each model clone class and variations in those classes and 2) a Simulink model evolution study.

2. Codify these mutations such that they can be injected into our Simulink model sets.

3. Develop a normalization process to transform the output of clone tool results into a common representation that can be used for tool evaluation.

4. Finalize and package the entire framework process as a whole by example of the prototype. This includes providing a well-formed definition of recall and precision, and discussing how one can evaluate them using the framework.

1.2.1 Scope

In order to demonstrate the framework and its feasibility, we develop a prototype that implements the framework for Simulink data-flow models as a proof-of-concept. This decision is based upon the quantity of models available to us from both public and industrial sources. In addition, this model type is of specific interest to both us and our industrial partners and we have already developed a clone detection tool able to work with Simulink and data flow models. Lastly, model clone detection for Simulink
is, by far and away, the most mature and prevalent form of model clone detection as we discuss later in this thesis.

We focus on Simulink clones at the system granularity, as this is the level that all tools report and is the level deemed to be the most relevant to the engineers and Simulink researchers we spoke with. We attempt to keep the framework itself fairly general and, while we build a prototype for Simulink model clone tool evaluation, we discuss briefly how it can be extended to other model types, such as structural models.

The operations we will consider for mutation operators will be based on model evolution and domain analysis. Specifically, this includes operations that 1) yield valid models AND 2) Are consistent with the way models evolve, that is, model evolution; OR 3) Cover all classes, also referred to as clone types, of model clones. As will be discussed in the thesis, we established 2) through surveying existing data-flow model evolution literature as well as performing domain analysis during our model clone detector tool development and doing an evolution study. In addition, we published the evolution study and mutation operators themselves in mutation [38] and evolution [39] conferences so as to receive feedback and acceptance from the model evolution community.

1.3 Contributions

Currently, there is no way to adequately evaluate model clone detectors. This includes both self evaluation and comparison with other tools. Thus, as new detection techniques, configurations, and tools arise, there is no way to easily measure quality. The contributions of this research are
1.4. ORGANIZATION OF THESIS

1. An evaluation framework that uses model mutations to estimate recall and precision for model clone detectors.

2. A catalogue and implementation of Simulink mutations that encompass the model clone types and are representative of model evolution in practice.

3. A prototype evaluation workbench that works on Simulink model clone detectors, uses the above Simulink mutations, and illustrates the framework in practice.

1.4 Organization of Thesis

We begin this thesis in Chapter 2 by providing background information on Simulink models, which are the kind of models at which our framework prototype is aimed. We then introduce model clone detection by outlining the various model clone types, and describing the two main types of model clone detection approaches: Graph-Matching Algorithms and Text-Based approaches. Since our framework is based on mutation analysis and has mutations that are validated through model evolution, the chapter also contains descriptions of those two concepts. Chapter 3 discusses work related to this thesis, including model transformations and, specifically, Simulink model transformations; and model mutations including existing Simulink model mutation work.

We then begin the meat of the thesis by providing an overview of our proposed framework and its process in Chapter 4, which starts off with preliminary attempts at evaluating model clone detectors and the challenges specific to doing so. We then split the framework into two phases, Mutation and Evaluation, and discuss them in that chapter.
1.4. ORGANIZATION OF THESIS

Chapter 5 presents our Simulink mutation taxonomy intended to cover all types of Simulink model clones. This is split up into three main categories, with each category containing mutation classes, and each class containing mutations along with their specific implementations we use for our prototype. This chapter also contains a model evolution study we employed to validate our taxonomy and choice of Simulink mutation classes. Chapter 6 outlines the automatic evaluation process for the framework by demonstrating how we accomplish it in our Simulink prototype. This includes a presentation of our desired clone report format, how we persist the injected mutant information, and how recall and precision are calculated in our prototype.

The thesis continues in Chapter 7 by using our prototype to actually evaluate multiple tool runs. We start by demonstrating how we transformed the various clone reports for the different tools. We then introduce the models we use in our evaluation experiments and the systems under study to be mutated along with the mutations. This is continued by a presentation of evaluation results for two model clone detection tools and different settings of one of those tools. We then discuss research questions and future work in Chapter 8 and provide a summary of our thesis conclusions and contributions.
Chapter 2

Background

The following chapter provides background information related to the material presented in thesis. In it, we discuss Simulink data-flow models, model clone detection and model clones, mutation analysis, and model evolution. Some of it is taken from our previously published work, including survey papers [37, 40].

2.1 Simulink

Simulink models are data-flow models consisting of three levels of granularity: whole models, (sub) systems, and blocks. Models contain systems, and systems contain other (sub) systems and blocks. They are quite prevalent in the embedded domains, especially the automotive and aerospace domains [17]. Figure 2.1 presents the detect_obstacle_endstop system from the automotive demonstration set that comes with Simulink. This is a system found in the Powerwindow model. It contains two subsystems: detect_obstacle and detect_endstop. It has an OR block and five port 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
2.2. MODEL CLONE DETECTION

Figure 2.1: Detect_Obstacle_Endstop Simulink System

have corresponding 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.

2.2 Model Clone Detection

Model-driven engineering is an ever-growing area in Software Engineering. As model-driven projects begin to age they can begin to exhibit similar properties to 3rd generation-programming language projects. One such property is the emergence of clones, which are similar or identical fragments situated throughout a project.

A code clone refers to fragments of code that are deemed similar to one another through some measure of similarity [25]. One common reason that code clones exist in software projects includes the implementation of a similar concept throughout the same system. A problem with code clones is that a change in this one concept means that the system may have to be updated in multiple places, which, sometimes,
is forgotten or difficult to do. Despite the fact that clones are generally seen as a negative, they are still introduced because of factors like poor reuse practices, time constraints, lack of knowledge about cloning, and others [25]. The research area devoted to the study, identification, evolution, and refactoring of code clones is very mature and there are many techniques and tools that are in existence to deal with them [33].

The analogous problem of model clones refers to models or model elements that are similar according to some definition of similarity. Due to the graphical nature of models, techniques intended to deal with code clones are not well suited to model clones. In comparison to code clone detection, the research related to model clone detection is quite novel and more limited. Model clone detection can be viewed as a more specific version of techniques intended to accomplish model comparison, of which there are numerous approaches [37, 40]. An important consideration about model clone detection is that it is an NP-complete problem because it is a special case of the largest common sub graph problem [19]. This problem entails looking for common sub graphs in a single graph. An example of a clone in data-flow modeling is displayed in Figure 2.2 [16]. The clone set in the two models shown through colorization come as a result of the same sub graph being present in both models after being normalized, or abstracted, in some fashion through a model clone detection approach.

While not as mature as its source-code counterpart, there are a number of different model clone detection approaches that have been developed and for different types of models including Simulink [16] and UML [42]. We now briefly go over the basics of some of these approaches. Much of this is based on our published survey on the area of Model Comparison [40].
2.2. MODEL CLONE DETECTION

Figure 2.2: Model Clone Example [16]
2.2. MODEL CLONE DETECTION

Figure 2.3: Type 1 Model Clone - Friction Mode subsystem found in Sldemo_Clutch and Sldemo_Clutch_if systems

2.2.1 Model Clone Types

We first include our definitions for model clones, as we laid out in [4], that are consistent with model clone detectors thus far. These examples are the same ones we described in earlier work [4] that are from the Simulink demonstration set 1.

Type 1 (Exact) Model Clones

Type 1, or exact, model clones are identical model fragments except for variations in visual presentation, layout and formatting. Figure 2.3 shows a single example system that is found in two different Simulink models in the demonstration set: Sldemo_Clutch and Sldemo_Clutch_if. Both of these models contain this duplicate subsystem, entitled Friction Mode. While this is a straight forward, duplicate, example, if one of the copies was laid out differently or had blocks or lines of a different colour, it would also be an example of a Type 1 model clone. We will see examples of this later in the thesis.

1http://www.mathworks.com/help/techdoc/ref/demo.html
Type 2 (Renamed) Model Clones

Type 2, or renamed, model clones are those that are structurally identical model fragments except for variations in labels, values, types, visual presentation, layout and formatting. Figure 2.4 displays an example of a Type 2 clone of two different subsystems: *Required Friction for Lockup*, on the right, and *Break Apart Detection*, on the left. Both of these subsystems are found in the *Sldemo_Clutch* example model of the Simulink demonstration set. The top part of the figure illustrates the context of the two subsystems while the bottom half illustrates the subsystems themselves. The first takeaway from this example is that, similar to code clones, model clones can cross structural and hierarchical levels. We see in the example the subsystem clones actually come from two different levels of abstraction in the same model of the Simulink automotive set.

The second takeaway is the differences between the two subsystems that make them Type 2 clones of each other. We see here that, while structurally identical, there are differences in block labels, such as Friction Torque versus TF and Max Friction Torque versus Tfmaxs. In addition, the value parameter for the comparator (rectangle) block differs from greater-than-equal-to in one subsystem and less-than-equal-to in the other.

Type 3 (Near-Miss) Model Clones

Type 3, or near-miss, model clones include clones that have modifications in addition to those in Type 1 or Type 2, such as changes in connection with respect to other model fragments and small additions or removals of blocks or lines.

Figure 2.5 illustrates a Type 3 clone pair of the *Throttle.throttle_estimate* and the
2.2. MODEL CLONE DETECTION

Figure 2.4: Type 2 Model Clone - Break Apart Detection and Required Friction for Lockup Subsystems

Figure 2.5: Type 3 Model Clone - \texttt{Throttle.throttle\_estimate} and \texttt{Speed.speed\_estimate} Subsystems

\texttt{Speed.speed\_estimate} subsystems, on the left and right, respectively. Both of these were found in the \texttt{sldemo_fuelsys} model of the Simulink demonstration set. In this example, a new block and line have been added to the subsystem on the right. In addition, there are naming and attribute changes to other blocks and lines.

Figure 2.6 shows an additional example of a Type 3 clone. In this case, the structure of the subsystem on the left, the \textit{Low Mode} system, differs from the subsystem
2.2. MODEL CLONE DETECTION

on the right, the Rich Mode system, in that a block is in another structural position in relation to other blocks.

2.2.2 Graph-Matching Algorithms

CloneDetective’s ConQAT [17] is the most mature approach used to perform clone detection in models. It draws from ideas in graph theory and the technique itself is applicable to any model that is represented as a data-flow graph. It is comprised of three steps. The first step taken by ConQAT includes preprocessing and normalisation. Preprocessing involves flattening, or inlining, all of the models and removing any unconnected lines. Normalisation takes all of the blocks and lines found in the models and assigns them a label that consists of information that is considered important for comparing them. The information described in the label changes according to the types of blocks being searched for by the tool. Figure 2.7 displays a sample of a model after it has gone through this initial step. This includes the labels that come from normalisation, such as UnitDelay and RelOp:. The grey portions in the graph represent a clone. These clone portions are identified by ConQAT in its second phase: clone pair extraction, whereby they iterate through node pairings using a breadth-first search. Lastly, they cluster their clones based on the set of nodes identified in the clone pairs.
2.2. MODEL CLONE DETECTION

Figure 2.7: Normalised Model Graph of Model Clone Example [17]
2.2. MODEL CLONE DETECTION

*eScan* and *aScan* are also graph-matching algorithms that attempt to detect exact-matched and approximate clones, respectively [29]. Like ConQAT, exact-matched clones are groups of model elements having the same size and aggregated labels, which contain topology information, and edge and node label information. Approximate clones are those that are not exactly matching but fit some similarity criteria. aScan uses vector-based representations of graphs that account for a subset of structural features in the graph. The main difference between these algorithms and ConQAT is that these algorithms group their clones first and from smallest to largest. They claim that this will help detect clones that ConQAT can not. This is later refuted, however, by the authors of ConQAT [16]. aScan is able to detect approximate clones while ConQAT is not. Neither *eScan* nor *aScan* are available for use or supported anymore. Thus we are not able to test them in our framework prototype.

Peterson [28] has developed the *Naive Clone Detector* to detect exact Simulink clones. Like ConQAT, it uses graph-based modeling and Simulink information, but by contrast, it employs a top-down approach.

2.2.3 Text-based Approaches

We recently developed *Simone*, which detects near-miss clones in Simulink models [4]. This is done by modifying existing code-clone techniques to work with the textual representations of the Simulink models while still being model sensitive. Specifically, seeing as Simone was an extension to NICAD [32], which is a parser-based clone detector, we began by creating a *TXL* [13] grammar for Simulink based on the large sets of models we had access to. We had to ensure that our grammar accounted for all Simulink constructs, including models, systems, blocks, lines, ports, branches
and other fine-grained model components. The next step involved us creating an
Simulink extractor that takes out the units of interest, for example, models, systems,
or blocks. After experiments and working with industrial partners, we discovered that
extracting systems yielded the most structurally meaningful clones of parts of models.

We then filter out text pertaining to color, font, spacing, orientation, printing and
other attributes that dominated Simulink models in their text representation. Even
after filtering, we were noticing that Simone was still missing some clones that it
should be detecting. We shortly realized that in some cases, the textual elements
comprising structurally identical systems may not be in the same order. Thus we
added a sorting component. Lastly, in order to account for Type 2 model clones,
we implemented “blind renaming”. After all these steps, Simone identifies “System”
clones that match up to a user-provided similarity percentage threshold by using their
underlying textual representation, an example of which is presented in Figure 2.8. So,
the system representation presented in Figure 2.8 is the Simone view of a Simulink
system before Simone has normalized and sorted the text.

Storrle [42] developed the MQlone tool to experiment with the idea of detecting
UML model clones. They convert XMI files from UML CASE models and turn them
into Prolog ². Once in Prolog, they attempt to discover clones using static identity
matching combined with similarity metrics such as size, containment relationships,
and name similarity. Very recently, model clone detection tools have been made for

²www.swi-prolog.org
2.3 Mutation Analysis

Mutation Analysis involves analyzing a software system using small modifications, or mutations, of elements in 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

```plaintext
System {
    Name             "onoff"
    Location         [168, 385, 668, 686]
    Open             on
    ModelBrowserVisibility  off
    ModelBrowserWidth   200
    ScreenColor        "automatic"
    PaperOrientation   "landscape"
    PaperPositionMode  "auto"
    PaperType          "usletter"
    PaperUnits         "inches"
    ZoomFactor         "100"
    AutoZoom           on
    ReportName         "simulink-default.rpt"
}
```

```plaintext
Block {
    BlockType       DiscretePulseGenerator
    Name             "Discrete Pulse\nGenerator"
    Position         [45, 25, 75, 55]
    Amplitude        "1"
    Period           "2"
    PulseWidth       "1"
    PhaseDelay       "0"
    SampleTime       "1"
}
```

```plaintext
Block {
    BlockType       Product
    Name             "Product"
    Ports            [2, 1, 0, 0, 0]
    Position         [145, 67, 175, 98]
    Inputs           "2"
    SaturateOnIntegerOverflow  on
}
```

Figure 2.8: Textual Representation of a Simulink System
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 [10], and C code to identify semantic misunderstandings [15]. Code mutations can be as simple as changing variable values, or more complex and tailored to a specific context, like Java Concurrency [10], as shown in Figure 2.9. The purpose of this mutation is to modify a method’s timeout and can be applied to any method call that has an optional timeout parameter. Specifically, in this case, we see that the mutational operator involves multiplying the supplied timeout by two. This allows evaluating how the system under test responds and reacts to different timeouts.

Model mutation analysis is essentially the same as traditional mutation analysis except that it involves making small-stepwise modifications to a model. We elaborate on related model mutations in Chapter 3.
2.4 Model Evolution

More people are employing model-driven paradigms, which have the models act as the main elements in the software development and management process. As software projects created using this paradigm begin to age, these models must evolve, be it through refactoring, bug fixes, feature additions and more. The notion of software evolution, in itself, is not a new idea [9], however, the “introduction of model-driven engineering . . . requires a new style of evolution.” [47]. Model evolution can refer to evolution of the models themselves, the meta models, platforms, and the abstractions (modeling languages) used [47]. These can be tied together through the notion of co-evolution, which describes how these artifacts evolve simultaneously. For the purpose of our research, we are focused mostly on the evolution of the models themselves because that is currently the level that clone detection is being applied to. So, specifically, we are concerned with how models evolve over time: the different operations, the frequency of these operations, and the motivation behind them. Evolution of Simulink models is a fairly unexplored area. This is in contrast to UML models, for which there exists a number of articles [26, 23, 18]. As such, this thesis includes an evolutionary study on Simulink models in Chapter 5.

There are some language-agnostic model and metamodel evolution approaches [21, 30] that can track both evolution and co-evolution. However, in order to use techniques like this for Simulink evolution, we would have to create a meta model conforming to those techniques specific requirements. As alluded to previously, there are some model comparison approaches [37, 40] that can find similarities and differences among models for versioning and other purposes, but there are no attempts to explicate the structural evolution of Simulink models. That is, to define what are the
potential structural changes that can occur to a Simulink model and their prevalence. Model evolution is strongly related to model comparison and versioning, but can be viewed as a longer-term analysis over multiple versions with a focus on how a specific artifact or clone has changed. None of the model comparison techniques we surveyed previously were ideal for tracking Simulink evolution. The only work that deals with any form of Simulink evolution is from Tran and Kreuz, who focus on refactoring Simulink [27]. Specifically, they look at forms of antipatterns in Simulink and discuss tool support for correcting them.

2.4.1 Model Variants and Software Model Product Lines

Related to model evolution are the notions of model variants and software model product lines. Both of these refer to instances where a model may have branching points of variability that represent differences in functionality or implementation.

For example, Simulink has built in functionality to allow for variants through a model variants block \(^3\). In this context, a variant represents a possible run-time “mode” that a block can operate in. Each variant references only a single model and only one variant may be active during a simulation.

Software product lines, in general, are ways of efficiently designing, creating, and managing a group software systems that are very similar but differ slightly. One such way of representing these systems is through the use of feature modeling [22]. Software model product lines refer to techniques intended to work with model-driven projects. An example technique that attempts to unify software product lines and model-driven engineering is the use of feature-based model templates [14]. These templates represent model variants for different types of models and contain a feature

\(^3\)http://www.mathworks.com/help/simulink/examples/model-reference-variants.html
model that represents the various branching points.

2.5 Summary

In this chapter we provided a review of the basic concepts of Simulink, model clone detection, mutation analysis, and model evolution in order to provide sufficient background for the remainder of the thesis. This included the three different types of model clones and the main two techniques employed for model clone detection. In the next chapter we will be discussing work related to the research completed in this thesis.
Chapter 3

Related Work

In this chapter we review related work in the areas of model transformations, model mutations, and other clone detector evaluation frameworks. In each instance, we discuss how it is related to our thesis work and what, if anything, can be leveraged. Some of this material has been published in a workshop on Mutation Analysis [38].

3.1 Model Transformation

Model transformations, which involve going from a source model to a specific target model, are generally related to the idea of model mutations and the corresponding edit operations we are looking for in our framework. A key difference between the mutations we discuss in this thesis and what is done in the majority of model transformation work, is we are retroactively looking at model evolution to see what operations have 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 [34], where they use graph grammars on meta models to develop model transformations. Also worth mentioning is the mutation work done on the ATL model transformation language by Khan and Hassine [24]. 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.

### 3.1.1 Simulink Model Transformations

Simulink model transformations are discussed by Tran et al. [45], 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.

Al-Batran et al. [3] note that existing model clone detection approaches deal with syntactic clones only, that is they can detect syntactically/structural similar copies only. Using normalization techniques that utilize graph transformations, they outline how to extend existing approaches to cover semantic, or Type 4, clones that may have similar behavior but different structure, for example those displayed in Figure 3.1 [3]. So, a clone in this context is now defined as two (sub)sets of models that have “equivalent unique normal forms” of models. These unique normal forms are acquired by having existing model clone detectors first perform forty semantic-preserving transformations that are structural modifications on Simulink models. The model fragments in Figure 3.1 are an example of clones that can be identified by a ’Joining Consecutive Sum/Product Blocks’ transformation: In system A, there are two product blocks that correspond to the joining of one product block in system B. While this is not an approach on its own, it is a way of extending a clone detection strategy to yield
more clones than simple syntactic comparison. As such, it can not be evaluated using our framework as is, however, the transformations could be incorporated as mutation/transformation operators as we discuss later on in Chapter 8.1. In addition, approaches that have incorporated this extra step can still have their syntactic clone detection abilities evaluated by the framework presented in this thesis.

### 3.2 Model Mutation

While research on model-based mutations is relatively newer than its source-code counterpart, there is still some work of note. Trakhtenbrot [44] introduces model mutations for state charts. This work is not directly applicable to Simulink models, however, there are Simulink Stateflow blocks, which are state charts.\(^1\) This work

\(^1\)mathworks.com/products/stateflow/
3.2. MODEL MUTATION

may be applicable to those blocks, in isolation. Adra and McMinn [2] developed mutations intended for agent-based models, while Bartel et al. [7] 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 the text-based mutation we will be discussing, the mRUE mutation in Section 5.1.1, 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 [36].

3.2.1 Simulink Model Mutation

The notion of Simulink model mutation frameworks have been addressed previously by Zhan and Clark [48], He et al. [20], and Araujo et al. [6]. In these works, 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. For example, the model in Figure 3.2, shows an example of a mutated Simulink model. Specifically, the thick block, AddMut, has been added to this model, in between a sum and a product block, in order to increase the signal.

In contrast, our proposed mutation taxonomy in this framework considers both design-time properties, which are necessary for model-clone detection testing, in addition to some run-time properties, like value changes. None of these works purpose a taxonomy, per se, however, Zhan and Clark identify three categories of signal mutations: Add, Multiply, and Assign, representing signal addition, multiplication, and specific value assignment, respectively. The mutation operators they purpose to accomplish those signal mutations can be classified using our taxonomy. We are
interested more in general mutations that modify the architectural structure of the model itself. In essence, our proposed framework attempts to mutate a Simulink model’s design-time properties while, in contrast, the frameworks proposed by Zhan and Clark and He et al. mutate a model’s run-time properties. That being said, sometimes a change in signal goes hand-in-hand with a change in structure, and vice versa. So, it may be interesting for us to determine if their mutators apply to our work.

3.3 Existing Clone Detector Evaluation Approaches

Early experiments on evaluating clone detectors was completed by Bellon et al. [8]. In their work, they evaluated six clone detectors on eight programs by using a human
oracle to analyze the clone candidates that were submitted from each tool. Similar to our work, they inject some code clone pairs to be sought after by the tools being evaluated.

After qualitatively evaluating code clone detectors [33], Roy and Cordy [31] proposed a mutation-based approach for comparing and evaluating source code clone detectors, which they recently implemented [43]. While the motivation of that is similar and we adapt the same general idea for this thesis, the mutation operators proposed for source code clones, presented in Figure 3.3, were validated using realistic programmer edit scenarios that cannot be carried over to a model mutation framework as they inapplicable to models. For example, things like white space, comment changes, changes in program lines, and others do not have a direct translation to the modeling, nor Simulink, domain. As such, we will be generating different, albeit potentially analogous, mutation operators of a different nature. In addition, they were not faced with the model-/domain- specific challenges that we were, as discussed in Section 4.2 including differing nature of the reported model clones and formats of the output files.

Somewhat related is an Eclipse Tool developed by Uhrig and Schwagerl [46] that is intended to evaluate matching algorithms intended for Eclipse Modeling Framework (EMF) models. They combine both user involvement and automated testing. This includes a match model that formalizes intended matches. While there are key differences between model matching, or comparison, and clones [40], it is possible that some of their work could be leveraged in the context of extending our framework to structural models by determining if their benchmarking matching algorithms are applicable.
3.4. Summary

In this chapter, we outlined work related to the research in this thesis. Model transformations are analogous to model mutations but are more related to forward engineering and require a specific source model. Refactoring and semantic model clones are examples where Simulink model transformations are employed, but can not assist us in injecting model clones. Model mutations and Simulink mutations are two growing areas, but neither can be used to inject model clones, as required by our framework. Lastly, we use an existing code-clone detection framework as the starting point in developing our framework for model clone detectors. In the next chapter, we introduce and provide an overview of our framework, including challenges that must be addressed that are unique to the modelling domain.

<table>
<thead>
<tr>
<th>Name</th>
<th>Mutation Description</th>
<th>Clone Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>mCW A</td>
<td>Change in whitespace (addition)</td>
<td>1</td>
</tr>
<tr>
<td>mCW R</td>
<td>Change in whitespace (removal)</td>
<td>1</td>
</tr>
<tr>
<td>mCC BT</td>
<td>Change in between token (/* */) comments</td>
<td>1</td>
</tr>
<tr>
<td>mCC EOL</td>
<td>Change in end of line (//) comments</td>
<td>1</td>
</tr>
<tr>
<td>mCF A</td>
<td>Change in formatting (addition of newlines)</td>
<td>1</td>
</tr>
<tr>
<td>mCF R</td>
<td>Change in formatting (removal of newlines)</td>
<td>1</td>
</tr>
<tr>
<td>mSRI</td>
<td>Systematic renaming of an identifier</td>
<td>2</td>
</tr>
<tr>
<td>mARI</td>
<td>Arbitrary renaming of a single identifier</td>
<td>2</td>
</tr>
<tr>
<td>mRL N</td>
<td>Change in value of a single numeric literal</td>
<td>2</td>
</tr>
<tr>
<td>mRL S</td>
<td>Change in value of a single string literal</td>
<td>2</td>
</tr>
<tr>
<td>mSIL</td>
<td>Small insertion within a line</td>
<td>3</td>
</tr>
<tr>
<td>mSDL</td>
<td>Small deletion within a line</td>
<td>3</td>
</tr>
<tr>
<td>mILs</td>
<td>Insertion of a line</td>
<td>3</td>
</tr>
<tr>
<td>mDLs</td>
<td>Deletion of a line</td>
<td>3</td>
</tr>
<tr>
<td>mMLs</td>
<td>Modification of a whole line</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 3.3: Code Mutations for Code Clone Framework [43]
Chapter 4

Framework Overview

Quantitative evaluation of model clone detectors is a key milestone for the field of model clone detection. As discussed in previous chapters, this evaluation has not yet been realized despite a number of emerging model clone detection approaches, tools, and configuration options. Our framework addresses this by using mutation analysis with mutations that are based on injection of different types of model clones and model evolution. We present and elaborate on our proposed framework through the means of an implementation of it intended to test Simulink model clone detectors.

In this chapter, we begin by discussing the preliminary steps we took in evaluating model clone detectors [4] that led us to realize that existing approaches for model clone detection evaluation were inadequate. This will help provide context for our research and proposed framework. We enumerate the specific challenges we encountered that motivated us to devise the framework in the manner we did. Lastly, we present and outline the framework, including each step in the general process. We published some of this material previously [4, 35, 36] and has been included and updated in this chapter.
4.1 Preliminary Evaluation

During our development and preliminary evaluation of Simone [4, 35], we were rather limited in terms of assessing Simone and comparing it to other tools. For the most part, the evaluation was qualitative. The only quantitative aspect involved us running different Simulink model clone detectors on the same set of models and manually checking the discovered system clones to see which tools discovered which clones.

Specifically, we compared Simone against ConQAT. Using the definition of model clone types in Chapter 2, ConQAT detects and identifies only type 2 (renamed) model clones. ConQAT detects but does not distinguish type 1 clones because of its renaming strategy.

To compare and contrast with ConQAT, we set Simone to use a NICAD [32] 30% near-miss difference threshold. To mimic ConQAT’s configuration, we chose to use blind renaming to ignore differences in names and values while including each block’s type. This set-up allowed us to detect and compare the set of type 2 clones discovered by Simone with those detected by ConQAT, and also demonstrate Simone’s detection of type 3 clones that ConQAT may not be able to find. In order to compare the two approaches, we use the same publicly available models from Matlab Central\(^1\) that were used by the ConQAT authors previously [16].

We could not use the Simulink demo model set because it contained Stateflow extensions, which are not handled by ConQAT. Thus our direct comparison was restricted to three systems from Matlab Central (MPC, MUL, and SIM). In these systems, we discovered Simone found all the clones detected by ConQAT. We present the results for one system, SIM, in Figure 4.1. We do this by means of a Venn diagram

\(^1\)http://www.mathworks.com/matlabcentral/
of the Simulink subsystem level clone classes discovered by running both tools on the communications system project, SIM, and manually going through the output and extracting the information. Circles outlined with dashes represent Simone near-miss clone classes discovered using filtering, sorting, and blind renaming, while the circles with full lines represent ConQAT clone classes discovered using the default ConQAT configuration. The percentages next to the circles represent the NICAD minimum pairwise similarity among all the subsystems in the clone class.

In this specific project, we can see that Simone was able to find all of the clones that ConQAT did, although in some instances, the corresponding ConQAT classes were embedded in larger near-miss SIMONE classes. Also of note are the near-miss (70%) clone classes identified by SIMONE only. Figure 4.2 shows an example of a near-miss clone pair discovered by Simone in the SIM system. The two systems are relatively similar. They differ only by the summation block, represented by the circle with two “+” signs, which splits the subsystem into two halves in the bottom, pcmwithnoise, subsystem. ConQAT and other graph based approaches will not detect this clone because the summation block creates a cut vertex that partitions the graph into two smaller sub-clones. These sub-clones may, individually, be beneath the size threshold for detection even if the near-miss clone as a whole is above the threshold, as is the case for this clone, which is not reported at all using ConQAT. By comparison, SIMONE reports these subsystems as near-miss clones with 72% similarity.

The key takeaway from this preliminary evaluation, however, was that the comparison process was non-trivially completed manually, was fairly qualitative, and was subject to the specific systems being used for comparison. Ideally, and what we attempt to accomplish in this thesis, is a more automatic, quantitative, and objective
Figure 4.1: System Clone Classes Detected by SIMONE and ConQat in the SIM Project [4]
Figure 4.2: Type 3 Subsystem Clone Pair Discovered by Simone but not ConQAT [4] way of comparing Simulink model-clone detectors.

In the next section we elucidate the challenges we faced during our manual comparison that were addressed in designing our mutation-analysis model-based framework.

4.2 Framework Design Challenges

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

As we outlined in previous work [4] and demonstrate in Figure 4.3, the second obstacle we found was coping with nested clones. This issue arises because Simone reports only the outer most (sub-)system satisfying the difference threshold whereas ConQAT reports identical clone groups and can cross subsystem boundaries, if desired by the user. As shown in the figure, the outer circles, which represent a clone pair that is 70% similar, contain an inner clone pair with 90% similarity. If ConQAT reported the 90% similar (according to Simone) clone, we would not be able to find the equivalent clone in Simone’s result set as Simone reports only the 70% outer clone pair as it is the largest subsystem meeting its threshold. Neither result is undesirable, it just makes it difficult to find corresponding matches between the two tools for the purpose of comparing recall. This issue of nested clones can be mitigated through mutation operators as well. Firstly, when comparing threshold-configurable tools, mutations being executed can be tailored to a specific level at a time and the tools could be set to look for only clones at that level. So, using the example in Figure 4.3, we start with lower level clones, for example the 90% clone class in the figure, and ensure that the tools that are configurable are set to that threshold. For comparing a
configurable tool and a tool that identifies only exact clone matches, we can adapt and extend the notion of *Fragment Containment* used for the source-code clone detector evaluation framework [31]. Specifically, they note that if a detected clone contains the clone introduced through mutation, then it is acceptable to say that the mutant clone has been detected. This aligns with the definition of “killed” mutants traditionally used in mutation analysis as it is an example of the “non-overlap binary definition” of detection. They define *Fragment Containment* as it applies to code, but we must define it as it applies to models: *Fragment Containment* in models is the case where all blocks and belonging to the clone introduced through mutation are a subset of the blocks of the detected clone instance.

Lastly, tools may provide different representations of their clone results. In our experiments we found ConQAT represents clone classes and clone instances by grouping individual blocks into “findings” of “finding groups” in XML format. SIMONE also reports its clone classes and instances into XML, however, the schema of the XML has different elements including the raw textual source of the model, which comprises the majority of data in the file. We elaborate further on this challenge.
when we describe how we eventually addressed it for our framework implementation in Section 7.1. Early on though, for our qualitative evaluation, we had to essentially do a manual comparison for completeness. The framework addresses this challenge by ensuring there is a normalization facility that can transform clone result output into a common form, if necessary. For purposes of comparison, an XML format that lists the system’s blocks sorted by both clone class and respective clone instances should suffice. For each new tool being compared, a transformation will have to be written only once, possibly with TXL [13], that takes output from the new tool and transforms it into this format.

4.3 Framework Details

While the general layout is similar to any mutation framework used for test case generation and tool evaluation for source-code; such as the one designed by Roy and Cordy [31, 43], which we used as a guideline for our framework; our framework differs from all other techniques as it addresses the model-specific challenges presented earlier. Similar to their framework, we also employ two phases, which we discuss below: the mutation phase, and the execution and evaluation phase.

4.3.1 Mutation Phase

The first phase involves mutating models and is illustrated in Figure 4.4 using systems as an example unit of granularity. Our framework process begins by allowing a user to select specific elements to use as the original source model elements. This can be done at different granularities. For example, in Simulink, by selecting a higher-level containing model file, or a specific system to use as the base. In that
user-selected scope, the framework then involves randomly selecting a configurable amount of (sub)systems to mutate. We herein refer to these elements as systems in scope (SIS). In our Simulink implementation of the framework, we copy the SIS from their corresponding model files and store them separately so they can be dealt with independently from the original source.

Once separated, each of the SIS can be duplicated and undergo mutation via the mutation operators, such as the ones we present in Chapter 5. While any combination or sequence of mutations can be selected for the purpose of this framework, we chose to implement our framework prototype by having each operator executed on each SIS. That is, say there are X systems to be mutated (SIS) and Y mutational operators, then there will be X*Y mutations performed. In addition, each operator randomly mutates the system based on the type of mutation it is and the user of the framework can specify how many mutations of each type they would like executed. So, if the user specifies they would like Z executions of the mutational operators, then there would be X*Y*Z mutations performed. We inject only a single mutant to each mutated SIS in isolation as this allows for explicit identification of which mutant was killed or missed. In addition, if we were to do multiple injections at once, it is possible that the mutated model would be changed to the point where it is no longer similar enough to be considered a clone. This phase of the framework can be implemented in such a way that it is automatic, as we show later in this thesis.

A notable difference between this model-clone framework and the code-based one is that, rather than inject clones back into the original source and run the comparison [31], we simply identify and duplicate the specific SIS, mutate it, and inject each mutated system as a standalone system in its own model file with a single mutation.
Figure 4.4: Mutation Phase with X systems, Y mutations, and Z variants
There are two reasons for this. Firstly, including the entire containing model and replacing the mutated system is not an option as it would then fall victim to the nested clone problem we discussed earlier. That is, higher-level containing systems, or the model itself, would be identified as clones 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, pre-mutated, system, at least from a Simulink modeling perspective, is unnatural. This is because all system components are connected to the rest of model, so it is unclear where a mutated system would belong. While there are some unconnected components that we discovered in our evolution study and discuss in Chapter 5, they are syntactically and semantically trivial reference or annotation blocks. As such, for our prototype implementation of the framework, we resort to running model-clone detectors on folders containing the original SIS and each stand-alone mutated copy of those SIS. Specifically, there will be one folder for each SIS and desired mutant variants, or $X^*Z$ folders using our variables from earlier. This satisfies our requirements for evaluating Simulink clone detectors while avoiding the nested clone problem. In addition, it mimics the notion of a modeler “copy and pasting”, and optionally modifying, a system from one place to another.

### 4.3.2 Execution and Evaluation Phase

As shown in Figure 4.5, the phase begins with each mutant-injected folder, which includes the original SIS and mutated copies of the SIS, being given as input to each of the respective model-clone detectors. The tools are then executed on each folder. As mentioned previously, any tool that has configurable similarity parameters
can have them set to match the specific clones being injected. The output of this step are clone reports. New to this model-based framework is the transformation of clone reports into the form we alluded to previously and delve deeper into in Chapter 6. This is necessary in a model version of a framework as both the nature and representations of the model clones may differ and impact the evaluation. The manual creation of a transformation will have to be done only once when a new tool is being used in the framework or an existing tool changes its output format. Otherwise, the transformation, like the rest of this phase, is automatic in that it can be executed on each of the resulting clone reports as they are generated. The transformed clone reports can then be subject to automatic model clone detection evaluation, including recall and precision calculation, which should now be much more straightforward given that the systems involved in the clone instances can be compared against those in an injection database/file set. This file set contains all the mutated clones data in a format conducive to comparison. Similar to what is done in the source-code framework, the evaluation database/file set stores the results of the detection evaluation that are then fed into a statistical analysis and reporting program, which we elaborate on in Chapter 6.

4.4 Summary

In this chapter, we began by providing an overview of our manual and qualitative comparison of two Simulink clone detectors. We highlighted the challenges we faced during that process including 1) recall calculation, 2) nested clones, and 3) differing clone report formats. We then introduce our framework that uses Mutation Analysis in order to address these challenges and provide a more automatic and quantitative
Figure 4.5: Model-Clone Detection Comparison Framework Evaluation Process
means of model clone detector evaluation. We breakup the framework into two phases. The mutation phase involves identifying targets for mutation and injecting mutants. The second phase involves executing the clone detectors to be tested on the mutated systems, transforming the resulting clone reports, and analyzing and comparing those reports. In the chapter that follows, we will be introducing Simulink mutations we create that are intended to inject model clones for our Simulink implementation of the framework.
Chapter 5

Mutators for Simulink Model Clone Injection

As discussed in the previous chapter, an important step of our framework involves conceiving mutations that cover and inject all types of model clones relevant to the specific type of model being considered. As such, we demonstrate an example of this process by walking through our creation, and validation, of Simulink model mutations that do just that in this chapter. We present and organize Simulink mutation classes and validate them through an evolution study. The classes are organized by higher-level categories. For each mutation class, we include 1) a description; 2) justification of its suitability; 3) an example of a mutation (operator) belonging to the respective mutation class, including an image; 4) and the programmatic representation of the mutation in our framework prototype. For the examples, we manually mutate existing models to make the concepts as clear as possible. We present the example 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. Much of this content is taken directly and updated from our paper in the workshop on Mutation Analysis [38], as we submitted there to help validate our taxonomy and mutation classes. There are a number of additions and more detail in this thesis, including
implementations and descriptions for the mutation operators.

The mutation operators, or instances, necessary for a model-clone detector comparison framework must test variations of all three model clone types. That is to say, we designed all of the Simulink mutation classes in such a way that, once instantiated in the prototype, the mutation instances will yield model clones representing all three types. In addition, we aim for mutation classes that are realistic edit scenarios as validated through our Simulink model evolution study [38]. We accomplish this by documenting each time each of our proposed mutation classes are witnessed as an evolutionary step in any clones discovered in 3 systems. These systems are made up of both open source and industrial examples.

For the actual programmatic representation and implementation of the mutations, we employ Matlab model functions\(^1\) in the Matlab Simulink programming language. The reason for this is these functions are equivalent to executing the same step-wise mutations through the Simulink GUI, which is the standard way engineers work on their models. Thus, the collateral impact of the mutations are minimal and they mimic plausible editing operations. The only mutation we do not implement in this fashion is the reordering text mutation (mRUE), discussed later, because this is an operation that can not be intentional executed in Matlab, rather it happens organically when Simulink is creating and modifying model elements and systems. For that, we use TXL [13].

We first begin this chapter by listing out and describing our proposed mutations classes, organized by higher-level categories. We then delve into our evolution study that attempts to demonstrate the existence of the proposed mutation classes in actual Simulink projects.

\(^1\)mathworks.com/help/simulink/functionlist.html
5.1. A TAXONOMY OF MUTATORS FOR SIMULINK MODELS

Table 5.1: Simulink Mutation Classes

<table>
<thead>
<tr>
<th>Mutation Key</th>
<th>Title</th>
<th>Section</th>
<th>Clone Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>mMLA</td>
<td>Modification of Layout Attribute</td>
<td>5.1.1</td>
<td>Type 1</td>
</tr>
<tr>
<td>mRUE</td>
<td>Reordering Underlying Elements</td>
<td>5.1.1</td>
<td>Type 1</td>
</tr>
<tr>
<td>mRBL</td>
<td>Renaming a Block or Line</td>
<td>5.1.2</td>
<td>Type 2</td>
</tr>
<tr>
<td>mCBV</td>
<td>Changing a Block's Value</td>
<td>5.1.2</td>
<td>Type 2</td>
</tr>
<tr>
<td>mADBD</td>
<td>Add or Delete Block as Destination</td>
<td>5.1.3</td>
<td>Type 3</td>
</tr>
<tr>
<td>mADBS</td>
<td>Add or Delete Block as Source</td>
<td>5.1.3</td>
<td>Type 3</td>
</tr>
<tr>
<td>mCBT</td>
<td>Changing a Block's Type</td>
<td>5.1.3</td>
<td>Type 3</td>
</tr>
<tr>
<td>mCSCH</td>
<td>Changing a Subsystem's Clone Hierarchy</td>
<td>5.1.3</td>
<td>Type 3</td>
</tr>
</tbody>
</table>

5.1 A Taxonomy of Mutators for Simulink Models

Before going into the details of the mutations, we summarize the mutation classes in Table 5.1. The Mutation Key column contains the short form that we will use to refer to each mutation class, the Title column indicates the title of the mutation class, Section refers to where in the thesis the mutation class can be found, and Clone Type refers to which of the three clone types the mutation class covers.

Based on our work with Simulink developing Simone and our Simulink clone evolution study [39], we categorize the mutations in the following way. This categorization and the mutation classes themselves are based on our observations of model edit operations encountered in both the large set of publicly available Simulink models available to us from the Mathworks Simulink demos, Matlab Central, open-source models, and our private industrial models. Much of this has been discussed and presented previously [36, 39].
5.1. A TAXONOMY OF MUTATORS FOR SIMULINK MODELS

5.1.1 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.

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 in that model by changing the foreground colour to red of a block of type Scope, entitled `Position`. The resulting mutant is pictured in Figure 5.1. In this case, the highlighted `Position` block was black until we mutated it to be red, as demonstrated in the figure.

Here we present two implemented mutation instances of the mMLA class. Listing 5.1 contains the code to change a random block’s foreground colour. In Simulink, colors are represented as a three number array with real numbers from zero to one. So, after selecting a random block in a provided system, three random numbers are generated and used to assign that randomly selected block, what is extremely likely to be, a new color. There is a mathematical possibility that the color could be the same as the previous ones, however the odds of that are negligible and most Simulink users simply specify their block colors using preset color strings. \(^2\)

\(^2\)http://www.mathworks.com/help/simulink/slref/common-block-parameters.html
5.1. A TAXONOMY OF MUTATORS FOR SIMULINK MODELS

Listing 5.1: Mutation Function for Changing a Random Block’s Color

```matlab
function f = mutChangeBlockColour(modelName, subSystemName)
    % Works on a model containing the subsystem, subSystemName.
    load_system(subSystemName);
    % find all blocks within the specified system
    blocks = find_system(subSystemName,'SearchDepth',1,'type', 'Block');
    % pick a random block
    blockToColour = blocks(randi([2 numel(blocks)]));
    % get a random colour and set the block to that.
    randomNumber1 = rand;
    randomNumber2 = rand;
    randomNumber3 = rand;
    newColor = strcat('[',num2str(randomNumber1),',',num2str(randomNumber2),','
                     ,num2str(randomNumber3),']');
    set_param(char(blockToColour), 'ForegroundColor', newColor);
    save_system(modelName);
end
```

The second implemented mutation instance of this class in our prototype changes a block’s position. This is presented in Listing 5.2 and also begins by selecting a random block from a provided subsystem. It then randomly generates a location to place the block. In Simulink, a block’s position involves both the origin point of the block and its size. So, after generating a random origin point, in the range of zero to one hundred in both the x and y dimensions to keep it in the upper left corner, we make the block arbitrarily 30x30. Again, there is a remote chance that the randomly selected position is equal to the block’s original position and size, but it is very unlikely.
Listing 5.2: Mutation Function for Changing a Random Block’s Position

```matlab
function f = mutChangeBlockPosition(modelName, subSystemName)
% Works on an unopened system indicated by the systemName variable.
load_system(subSystemName);
% find the names of all blocks within the specified system
blocks = find_system(subSystemName, 'SearchDepth', 1, 'type', 'Block');
% pick a random block
blockToMove = blocks(randi([2 numel(blocks)]));
% pick a random start position from (0..100 , 0..100)
randomNumberTop = randi([1 100]);
randomNumberLeft = randi([1 100]);
% make a block 30x30 wide
set_param(char(blockToMove), 'Position', [randomNumberLeft randomNumberTop 
randomNumberLeft+30 randomNumberTop+30]);
save_system(modelName);
end
```

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 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 we presented 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 5.2, 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.

As alluded to earlier, this was the one mutation that we could not implement through the Matlab GUI or programmatically through Matlab. Instead, it seemed like a task perfectly suited to the TXL transformation language [13]. Thus, we created a TXL transformation that randomly selects two elements in a system and switches their location in the textual representation.\(^3\) We provide an excerpt of the transformation in Listing 5.3 with the full listing in Listing A.1 in Appendix A. As demonstrated in the excerpt, and in more detail in the Appendix, the transformation takes the provided path of the system, *SystemPath*, and splits it up into two parts, *SystemPathSegments*. After finding the occurrence of the first part of the split, it makes a recursive call on the latter part of the split. When the transformation has only one component left in the chain, we pick two random numbers in order to randomly select the two elements we will be swapping. Swapping is done in the standard ‘swap-with-temporary-variable’ technique.

\(^3\)Transformation written by Andrew Stevenson
Figure 5.2: Example of an mRUE Mutation Operator
5.1. A TAXONOMY OF MUTATORS FOR SIMULINK MODELS

Listing 5.3: Excerpt of TXL Transformation for Reordering Textual Elements

```plaintext
... rule mutateSystem PathSegments [repeat stringlit]
  deconstruct PathSegments
  SystemName [stringlit] RestOfPath [repeat stringlit]
  replace $ [system_list]
  System {
    Name SystemName
    SingleElements [repeat default_single_element]
    ListElements [repeat compound_element]
  }
  by
  System {
    Name SystemName
    SingleElements
    ListElements [swapBaseCase PathSegments]
    [mutateSystem RestOfPath] % strip off the path head
    and call recursively with tail
  }
end rule
...
```

5.1.2 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 in this category are Type 2.

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.” 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 5.3. All other elements in 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.

The first implemented mutation for this mutation class involves mutating a random block by renaming it and is presented in Listing 5.4. It is a relatively straightforward mutation operation in that it simply selects a random block in the provided subsystem and gives it the name “mutatedName”, which we do not anticipate will be
the name of any existing blocks. If we wanted to, we could also randomly generate a
string name and inject it or even modify the existing name, but the result would be
the same.

Listing 5.4: Mutation Function for Renaming Random Block

```matlab
function f = mutRenameBlock(modelName,subSystemName)
    % Works on an unopened system indicated by the systemName variable.
    load_system(subSystemName);
    % find the names of all blocks within the specified system
    blocks = find_system(subSystemName,'SearchDepth',1,'type', 'Block');
    % pick a random block
    blockToRename = blocks(randi([2,numel(blocks)]));
    % give that random block our mutated name.
    set_param(char(blockToRename), 'Name', 'mutatedName');
    save_system(modelName);
end
```

The other mutational operator for this mutation class we have implemented in-
volves mutating a random line by changing/giving it a name and can be found in
Listing 5.5. This operation is very similar to renaming a block and exploits Simulink’s
ability to retrieve all lines in a subsystem. We simply select one of those and rename
it to “mutatedLineName”. In this case, we add the additional check of identifying a
system with only unconnected blocks and disregarding the mutation in this case.

Listing 5.5: Mutation Function for Renaming Random Line

```matlab
function lineWasFound = mutRenameLine(modelName,resolvedSystemName, subSystemName)
    % Works on an unopened system indicated by the systemName variable.
    load_system(subSystemName);
    % pick a random line
    lines = find_system(subSystemName,'SearchDepth',1,'FindAll', 'on', 'type', 'line');
    if (~isempty(lines))
        lineToMutate = lines(randi(numel(lines)));
        set_param(lineToMutate, 'Name', 'mutatedLineName');
        lineWasFound = 1;
    else
        % We have the unlikely case where there’s a model with no lines (all
disjoint blocks).
        lineWasFound = 0;
    end
end
```
5.1. A TAXONOMY OF MUTATORS FOR SIMULINK MODELS

5.1.1 close_system(subSystemName);
5.1.2 newModelFileName = strcat(resolvedSystemName,'/',modelName,'.mdl');
5.1.3 delete(newModelFileName);
5.1.4 end
5.1.5 save_system(modelName);
5.1.6 end

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 [10]. 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 5.4. 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. Specifically, the example mutation operator demonstrated changes the value of the Gain block from 50 to 25.

\[^5\text{mathworks.com/help/simulink/slref/gain.html}\]
Listing 5.6 provides our initial implementation for mutating a block’s value in our prototype. At this point, we stick to three common blocks and change their primary values: The Gain block’s “Gain” value, the Constant block’s “Value” value, and the Integrator block’s “InitialCondition” value. While this is by no means exhaustive, these are commonly used Simulink blocks, and we could easily add any blocks and values we like to this mutation operator. In terms of execution, we begin by gathering all the blocks of the specific type we are looking for by employing the Simulink find_system functionality. After making sure we have at least one eligible block, we then pick randomly from those blocks and mutate the appropriate value. In this case, we pick a specific number rather than modifying an existing value because in some cases a value may actually be a dynamic variable that we would not be able to increment statically.

Listing 5.6: Mutation Function for Changing a Block’s Value

\[\text{http://ctms.engin.umich.edu/CTMS/index.php?aux=Extras_Blocklib#2}\]
5.1. A TAXONOMY OF MUTATORS FOR SIMULINK MODELS

5.1.3 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.

**Add or Delete Block as Destination (mA\text{ADBD})**

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,\textsuperscript{7} lines (signals), and required ports.

From a suitability perspective, changing a destination block in 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 5.5. 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, \textit{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 \textit{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 \textit{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 \textit{move\_Down} 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

\textsuperscript{7}\url{mathworks.com/help/simulink/sinks.html}
that outport and delete that signal only.

Adding a block as a destination is a relatively simple mutation operator that we implement in our prototype using the Matlab code in Listing 5.7. We begin by adding a new block, in this case a Scope block, to the subsystem. We could have chosen any sink block other than an outport block, as an outport block would involve connections at a higher-level. After we add the block to an arbitrary position in the subsystem, we then pick a random line to branch and connect to this new block.
Figure 5.5: Examples of Adding (left) or Deleting (right) Block as Destination (mADBD) Mutation Operators
Listed 5.7: Mutation Function for Adding a Block as Destination

```
function f = mutAddBlockAsDest(modelName, subSystemName)

  load_system(subSystemName);
  lines = find_system(subSystemName,'SearchDepth',1, 'FindAll', 'on', 'type', 'line');
  newBlock = strcat(subSystemName,'/mutNewDestinationBlock');
  newBlockHandle = add_block('simulink/Sinks/Scope', newBlock);
  set_param(newBlock, 'position', [400 400 430 430]);
  lineToConnectTo = lines(randi(numel(lines)));
  temp = get_param(newBlockHandle,'PortHandles');
  add_line(subSystemName,get_param(lineToConnectTo,'SrcPortHandle'),temp.Inport);
  save_system(modelName);
end
```

For our implementation of deleting a block as a pure destination, or sink, we target only sinks that are not outports, as deleting outports would involve changing systems at multiple levels. Thus, as shown in Listing 5.8, we delete a source block only if it is not an outport. We begin by gathering all blocks and then searching for a block that has zero outgoing ports and is not an outport itself. If we discover a suitable block, we then retrieve all the lines connected to it and delete them and the block itself.
5.1. A TAXONOMY OF MUTATORS FOR SIMULINK MODELS

Listing 5.8: Mutation Function for Deleting a Block as a Destination

```matlab
function blockWasDeleted = mutDeleteBlockAsDest(modelName, resolvedSystemName, subSystemName)
    % Works on an unopened system indicated by the subSystemName variable in the model, modelName. The resolvedSystemName provides the containing folder of the file holding the model(file).
    load_system(subSystemName);
    blocks = find_system(subSystemName,'SearchDepth',1,'type','Block');
    blockWasDeleted = 0;
    % Go through the blocks until we find a non-outport dest block.
    for blockIterator = 2:numel(blocks)
        blockToDelete = blocks(blockIterator);
        ports = get_param(blockToDelete,'Ports');
        type = get_param(blockToDelete,'BlockType');
        % The following will be 0 in the case of a destination block.
        if (~ports{1}(2)) && ~strcmp(type,'Outport')
            blockWasDeleted = 1;
            break;
        else
            blockWasDeleted = 0;
        end
    end
    if (~blockWasDeleted)
        close_system(subSystemName);
        newModelFileName = strcat(resolvedSystemName,'/',modelName,'.mdl');
        delete(newModelFileName);
        return;
    end
    % Note all lines to be deleted. Do not delete here because it'll crash on % branches.
    blockToDeleteHandle = get_param(blockToDelete,'Handle');
    lines = find_system(subSystemName,'SearchDepth',1,'FindAll', 'on', 'type', 'line');
    linesToDelete = double.empty;
    for currentLineIndex = 1:numel(lines)
        currentLine = lines(currentLineIndex);
        if (blockToDeleteHandle{1} == get_param(currentLine,'DstBlockHandle'))
            linesToDelete(end+1) = currentLine;
        end
    end
    % delete the lines seperately so we can handle branches.
    for currentLineToDelete = 1:numel(linesToDelete)
        delete_line(linesToDelete(currentLineToDelete));
    end
    delete_block(blockToDelete);
    save_system(modelName);
end
```
5.1. A TAXONOMY OF MUTATORS FOR SIMULINK MODELS

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. At the system level, this would involved adding a source blocks.\(^8\)

Changing the source blocks in a system are likely a common operation in 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 5.6, 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\nSOURCE 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, 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.

In terms of the basic case, of adding an unconnected block as a source, we implement the mutation operator of adding an unconnected source block, a Step block,

\(^8\)mathworks.com/help/simulink/sources.html
5.1. A TAXONOMY OF MUTATORS FOR SIMULINK MODELS

Figure 5.6: Examples of Adding (left) or Deleting (right) Block as Source (mADBS) Mutation Operators
5.1. A TAXONOMY OF MUTATORS FOR SIMULINK MODELS

To a random system, via the Matlab code presented in Listing 5.9. We are unable to connect it to anything at this point because branches can be created on only outgoing, and already connected, lines. This is still a valid and executable model. The more complicated case of adding a block as a source with respect to another block is covered in the next section.

Listing 5.9: Matlab Code for Adding an Unconnected Source Block

```matlab
function f = mutAddBlockAsSource(modelName, subSystemName)

% Works on an unopened system indicated by the systemName variable.
load_system(subSystemName);
% create the new block
newBlock = strcat(subSystemName,’/mutNewSourceBlock’);
simulink;
newBlockHandle = add_block(’simulink/Sources/Step’, newBlock, ’Position’,
[400 400 430 430]);
simulink(’close’);
save_system(modelName);
end
```

Deleting a source block without modifying other systems is possible for all non-inports (interface) source blocks. As such, our mutation function, displayed in Listing 5.10, goes through all blocks in a specified subsystem looking for a block with no incoming ports that is not an inport type block. After it finds the appropriate block, it has to connect the soon-to-be open port to an existing block. Because lines are created with respect to blocks, we need to find another block that has at least one connected outgoing port and is not the block we already selected. Once that occurs, we can then delete the source block and its incoming lines, while reconnecting the now open port to another random, but suitable, block in the model.

Listing 5.10: Matlab Code for Deleting a Non-Inport Source Block

```matlab
function blockWasDeleted = mutDeleteBlockAsSource(modelName, resolvedSystemName, subSystemName)

% Works on an unopened system indicated by the subSystemName variable in
% the
```
% model, modelName. The resolvedSystemName provides the containing folder
% of the file holding the model(file).
load_system(subSystemName);
bloods = find_system(subSystemName,'SearchDepth',1,'type','Block');
blockWasDeleted = 0;
% Go through the blocks until we find a non-inport source block.
for blockIterator = 2:numel(bloods)
    blockToDelete = bloods(blockIterator);
    ports = get_param(blockToDelete,'Ports');
    type = get_param(blockToDelete,'BlockType');
    % The following will be 0 in the case of a source block.
    if (~ports{1}(1)) && ~strcmp(type,'Inport')
        blockWasDeleted = 1;
        break;
    else
        blockWasDeleted = 0;
    end
end
if (~blockWasDeleted)
    close_system(subSystemName);
    newModelFileName = strcat(resolvedSystemName,'/','modelName','mdl');
    delete(newModelFileName);
    return;
end
% Note all lines to be deleted. Do not delete here because it’ll crash on
% branches.
blockToDeleteHandle = get_param(blockToDelete,'Handle');
lines = find_system(subSystemName,'SearchDepth',1,'FindAll', 'on', 'type',
    'line');
linesToDelete = double.empty;
for currentLineIndex = 1:numel(lines)
    currentLine = lines(currentLineIndex);
    if (blockToDeleteHandle(1) == get_param(currentLine,'SrcBlockHandle'))
        linesToDelete[end+1]= currentLine;
    end
end
% Connect the open port to an existing block. First find existing block.
blockToConnect = cell.empty;
counter = 0;
while counter <= numel(bloods)*2
    blockToConnect = bloods(randi([2 numel(bloods)]));
    ports = get_param(blockToConnect,'Ports');
    % if the block has 1 or more outports and isn’t the blockToDelete
    if ports{1}(2) && ~strcmp(blockToConnect{1},blockToDelete{1})
        break;
    else
        blockToConnect = cell.empty;
    end
    counter = counter+1;
end
% delete the lines seperately so we can handle branches.
Add or Delete Block in Middle of a System

Both mADBS and mADBD involve block modification as both sources and destinations with respect to other blocks. A specific, but important, instance of this class of mutations to consider are when a block is added or deleted from the middle of a system. We present these specific cases in this section, which we were unable to elaborate on in our previous presentation of the taxonomy [38] due to a lack of space.

Adding a block in between other blocks in a system is a relatively straight-forward mutation that involves selecting a random location in a system and injecting a new mutant block there. Since this is a specific example of mADBS and mADBD, it is clear that this mutation injects a type 3 clone. This is a structural change and would definitely be of the near-miss variety, likely with a very high similarity value because of the small nature of the change.

Adding a block in this manner is a very plausible operation by Simulink modelers as there may be different operations that need to be added to a Simulink signal along its path that was not considered before.

We present an example of this in Figure 5.7. In this case, we continue using the
window_system subsystem. Specifically, we add the highlighted gain block mutInBet-
tweenBlock, randomly, in between the friction block and the sum block. Seeing as all
the previous connections are still accounted for, this is still a valid Simulink model
and will run in Matlab.

In code listing 5.11, we present our implementation of the mutation function to
add a block to the middle of a system. We make this function fairly simple by
selecting a random line to intersect and adding a new Gain block of an arbitrary size
and position. We then delete the original line that we will be intersect and replace it
with two lines going into and out of the block, respectively.

Listing 5.11: Mutation Function for Adding an Intersecting Block

```matlab
function f = mutAddBlockInbetween(modelName,subSystemName)
% Works on the indicated subSystem given the provided modelName.
load_system(subSystemName);
% find the names of all the lines within the specified system
lines = find_system(subSystemName,'SearchDepth',1,'FindAll','on','type','line');
% pick a random line to intersect. Any line will do.
lineToConnectTo = lines(randi(numel(lines)));
```
Deleting a block in a system that is located in the middle can be a more complicated mutation than adding a block. In this case, we need to consider and preserve all incoming and outgoing connections into the blocks. In order to simplify things, we limit this mutation to randomly select deleting blocks that have only a single incoming and a single outgoing port. The reason for this is because it is unclear how one would handle a case where a block to be deleted had a different number of inputs and outputs. Specifically, consider a case where a block to be deleted had two inputs and one output, such as the Sum block in the window system we have been considering thus far. If this block was to be deleted, only one of the two incoming connections could be connected to the subsequent block that was originally connected to the block to be deleted, which would be the gain block from the window system subsystem example. Additional, blocks containing only a single incoming and outgoing port are quite common.\(^9\) As such, at this point, we have this mutation operator find and delete a randomly-selected block from the middle of a system by selecting

\(^9\)mathworks.com/help/simulink/blocklist.html
blocks that have only a single incoming and outgoing port only.

In regards to this edit operation from an engineering perspective, there are many reasons why a modeller would delete an element. Our example presented in Figure 5.8, where we have deleted the friction block from the window_system subsystem, actually demonstrates such a case. In this instance, the engineers could remove this element if they no longer wanted to account for friction in this simulation. This could be to estimate the impact of friction on the rest the subsystem and model, or even for illustrative purposes.

The example in Figure 5.8 involved a mutational operator that randomly selected and deleted that single friction block. In this case, the source was previously a branched line, so this mutation had to account for that and connected that same branch to the output of the no-longer present friction block as demonstrated by the highlighted line in the figure.

Listing 5.12 provides our mutation function for deleting a block from the middle of
the system. As discussed, we start by selecting a block that has exactly one incoming
and one outgoing port only. Once selected, we then get all the associated lines that
are coming into and leaving from that block. For the outgoing lines we also have to
account for the case that the outgoing lines are part of a line branch. We then are
able to delete those lines, and connect the appropriate lines to complete the system.

Listing 5.12: Mutation Function for Deleting an Intersecting Block

```matlab
function blockWasFoundAndDeleted = mutDeleteBlockInbetween(modelName,
resolvedSystemName, subSystemName)
    % Works on an unopened system indicated by the subSystemName variable in
    % the
    % model, modelName. The resolvedSystemName provides the containing folder
    % of the file holding the model(file).
    load_system(subSystemName);
    blockWasFoundAndDeleted = 0;
    blockToDelete = 0;
    counter = 0;
    blocks = find_system(subSystemName,'SearchDepth',1,'type', 'Block');
    while counter <= numel(blocks)*2
        % pick a random block
        blockToDelete = blocks(randi([2,numel(blocks)]));
        ports = get_param(blockToDelete,'Ports');
        % select only blocks with a single incoming and outgoing port.
        if (ports{1}(1) == 1 && ports{1}(2) == 1)
            blockWasFoundAndDeleted = 1;
            break;
        end
        counter=counter+1;
    end
    if blockWasFoundAndDeleted
        % this means we found a block
        blockToDeleteHandle = get_param(blockToDelete,'Handle');
        blockToDeleteHandle = blockToDeleteHandle{1};
        incomingLineToBlockToDelete = find_system(subSystemName, 'FindAll', 'on'
            , 'type', 'line','DstBlockHandle',blockToDeleteHandle);
        outgoingLinesFromBlockToDelete = find_system(subSystemName, 'FindAll', '
            on', 'type', 'line','SrcBlockHandle',blockToDeleteHandle);
        sourcePortHandleForNewLine = get_param( incomingLineToBlockToDelete, '
            SrcPortHandle');
        branchHandle = 0; % onlyUsedIfWeHaveABranch
        % getDestinationPortHandlesFirst
        destinationPortHandlesForNewLine = double.empty;
        for currentLineIndex = 1:numel(outgoingLinesFromBlockToDelete)
            % ignore branches
```
5.1. A TAXONOMY OF MUTATORS FOR SIMULINK MODELS

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 [17], 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

if numel(get_param(outgoingLinesFromBlockToDelete(currentLineIndex), 'DstPortHandle')) == 1
    destinationPortHandlesForNewLine(end+1)= get_param(
        outgoingLinesFromBlockToDelete(currentLineIndex), 'DstPortHandle');
    elseif numel(get_param(outgoingLinesFromBlockToDelete(currentLineIndex), 'DstPortHandle')) >1
        % we have a branch. Record that line handle for deletion
        branchHandle = outgoingLinesFromBlockToDelete(currentLineIndex);
        end
end
%f delete the originally selected line and any incoming lines to the %block
if branchHandle
    delete_line(branchHandle);
else
    delete_line(outgoingLinesFromBlockToDelete);
end
delete_line(incomingLineToBlockToDelete);
for currentDestinationPortHandle = 1:numel(destinationPortHandlesForNewLine)
    add_line(get_param(blockToDelete, 'Parent'),
        sourcePortHandleForNewLine,destinationPortHandlesForNewLine(
            currentDestinationPortHandle));
end
delete_block(blockToDelete);
save_system(modelName);
else
    % we did not find a block to delete
    close_system(subSystemName);
    fullPathToFile = strcat(resolvedSystemName,'/','modelName','.mdl');
    delete(fullPathToFile);
end
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 5.9. 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.

We present our implementation for mutating the block type of a random block in Listing 5.13. We begin by selecting suitable block candidates to mutate. Once again, we limit ourselves to blocks with a single incoming and outgoing port as this simplifies the mutation process significantly and blocks of this nature are quite common. It is possible to consider blocks with various numbers of incoming and outgoing ports, however, it would require exhaustive manual “matching” of block types and many new (if) cases in the mutation function. Once we have our candidates, we select a random block from that list. We then replace the selected block with a default integrator block, unless the random block selected is already an integrator block, in which case
5.1. A TAXONOMY OF MUTATORS FOR SIMULINK MODELS

Figure 5.9: Example of an mCBT Mutation Operator
we replace that block with a Gain block. We then account for, delete, and replace the appropriate lines in multiple steps, as lines are read only when working with them programmatically. In this process, we once again have to consider branches.

Listing 5.13: Mutation Function for Changing a Block’s Type

```matlab
function blockFoundAndChanged = mutChangeBlockType(modelName, resolvedSystemName, subSystemName)

% Works on an unopened system indicated by the subSystemName variable in the
% model, modelName. The resolvedSystemName provides the containing folder
% of the file holding the model(file).
load_system(subSystemName);
allBlocks = find_system(subSystemName,'SearchDepth',1,'type', 'Block');
blockCandidates = [];
% check every block and add it as a candidate if it has 1 input and 1 output.
for iterator = 2:numel(allBlocks)
    currentBlock = allBlocks(iterator);
    ports = get_param(currentBlock,'Ports');
    if (ports{1}(1) == 1 && ports{1}(2) == 1)
        % if it's a block with 1 input and 1 output, we can replace it.
        blockCandidates = [blockCandidates; currentBlock];
    end
end
if isempty(blockCandidates)
    % we did not find a block to delete
    close_system(subSystemName);
    newModelFileName = strcat(resolvedSystemName,'/',modelName,'.mdl');
    delete(newModelFileName);
    blockFoundAndChanged = 0;
    return;
end
% pick a random block from our candidates.
blockToReplace = blockCandidates(randi(numel(blockCandidates)));
incomingLineToBlockToDelete = find_system(subSystemName, 'FindAll', 'on', 'type', 'line','DstBlockHandle',get_param(char(blockToReplace),'Handle'))
;
outgoingLineFromBlockToDelete = find_system(subSystemName, 'FindAll', 'on', 'type', 'line','SrcBlockHandle',get_param(char(blockToReplace),'Handle'))
;
nameOfBlockToReplace = get_param(char(blockToReplace),'Name');
positionOfBlockToReplace = get_param(char(blockToReplace),'Position');
typeOfBlockToReplace = get_param(char(blockToReplace),'BlockType');
delete_block(blockToReplace);
newBlock = 0;
if strcmp(typeOfBlockToReplace,'Gain')==1
```

5.1. A TAXONOMY OF MUTATORS FOR SIMULINK MODELS 76
5.1. A TAXONOMY OF MUTATORS FOR SIMULINK MODELS

newBlock = add_block('built-in/Integrator', strcat(subSystemName,'/',
nameOfBlockToReplace), 'Position', positionOfBlockToReplace);
else
newBlock = add_block('built-in/Gain', strcat(subSystemName,'/','
nameOfBlockToReplace), 'Gain', '1', 'Position',
positionOfBlockToReplace);
end
sourcePortHandleForNewLine = get_param( incomingLineToBlockToDelete, '
SrcPortHandle');
destinationPortHandleForNewLine = get_param( outgoingLineFromBlockToDelete,
'DstPortHandle');
delete_line(incomingLineToBlockToDelete);
delete_line(outgoingLineFromBlockToDelete);
newBlocksPortHandles = get_param(newBlock,'PortHandles');
add_line(subSystemName, sourcePortHandleForNewLine,newBlocksPortHandles.
Inport);
if (numel(destinationPortHandleForNewLine) > 1)
for iterator = 1 : (numel(destinationPortHandleForNewLine)-1)
    add_line(subSystemName,newBlocksPortHandles.Outport ,
    destinationPortHandleForNewLine(iterator));
end
else
    add_line(subSystemName,newBlocksPortHandles.Outport ,
    destinationPortHandleForNewLine);
end
blockFoundAndChanged = 1;
save_system(modelName);

Changing a subsystem’s clone hierarchy (mCSCH)

This class of mutations involves mutation instances that mimic a batch of edit op-
erations 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.\textsuperscript{10} 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

\textsuperscript{10}mathworks.com/help/simulink/ug/creating-subsystems.html#f4-7371
subsystems is one of the more useful aspects of Simulink. Through the use and creation of block libraries and sublibraries, 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 5.10. 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.

Fortunately, the implementation for creating a subsystem in a provided subsystem, as shown in Listing 5.14, is significantly simplified by the fact that we can employ the, already discussed, built-in Simulink function to create a subsystem from a set of blocks. Specifically, we select a range of blocks in the system, add them to a set, and use that set in our call to the “createSubSystem” function in Simulink. The block set need not take blocks directly connected to one another since the built-in function adds all the appropriate inports and outports to retain model cohesion and validity.\footnote{mathworks.com/help/simulink/ug/creating-block-libraries.html}
Figure 5.10: Example of an mCSCH Mutation Operator
Listing 5.14: Mutation Function for Changing the System Hierarchy

```matlab
function f = mutCreateSubsystem( modelName, subSystemName )
    % This function creates a randomly sized subsystem out of
    % randomly selected blocks in the provided subsystem.
    load_system(subSystemName);
    blocks = find_system(subSystemName, 'SearchDepth', 1);
    bh = [];
    startingBlock = randi([2 numel(blocks)],1,1);
    finishingBlock = randi([startingBlock numel(blocks)], 1,1);
    for i = startingBlock:finishingBlock
        bh = [bh get_param(blocks{i}, 'handle')];
    end
    Simulink.BlockDiagram.createSubSystem(bh);
    save_system(modelName);
end
```

5.2 Validating the Taxonomy: Evolutionary Case Studies

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 [39], 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 in a project, we look for mutation instances that occur from one version to the next in any subsystems that have been identified as belonging to a clone set in the original
5.2. VALIDATING THE TAXONOMY

Table 5.2: Projects Under Study. Modified from [39]

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

version. This is sufficient as we are simply trying to exhibit that these mutations do, in fact, exist in 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).\(^\text{12}\) Table 5.2 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.

\(^\text{12}\)http://sourceforge.net/projects/adv-vehicle-sim/?source=dlp
In order to look for evidence of mutation operators in these projects, we use Simulink’s model XML comparison tool,\textsuperscript{13} which is part of the Simulink Report Generator package. While performing model comparison using XML comparison has its drawbacks [37, 40], this tool was very useful for our purposes when combined with our domain knowledge and some manual 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 in the models as we went through it.

Using the following filtration settings,\textsuperscript{13} we were able to trace through the different project versions, and tally and classify the mutation instances as we encountered them:

\textbf{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.

\textbf{Do not filter - Changes in lines}

Changes in lines are often indicative of changes in source or destination blocks, which is information we want.

\textbf{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

\textsuperscript{13}mathworks.com/help/rptgenext/ug/
how-to-compare-xml-files-exported-from-simulink-models.html
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 in 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 in 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 in 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; 5.2.1, 5.2.2, and 5.2.3; 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.

### 5.2.1 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 5.3. 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 mCSCCH 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 5.11. 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
5.2. VALIDATING THE TAXONOMY

Figure 5.11: Version Three and Four of the PW Project window system Subsystem
Table 5.3: Summary of Mutation Instances Witnessed in the PW Project

<table>
<thead>
<tr>
<th># Of Times a System Changed</th>
<th>mMLA</th>
<th>mRUE</th>
<th>mRBL</th>
<th>mCBV</th>
<th>mADBD</th>
<th>mADBS</th>
<th>mCBT</th>
<th>mCSCH</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Count</td>
<td>15</td>
<td>10</td>
<td>7</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>% of Total # of System Changes</td>
<td>-</td>
<td>66.67%</td>
<td>46.67%</td>
<td>26.67%</td>
<td>26.67%</td>
<td>40.00%</td>
<td>53.33%</td>
<td>20.00%</td>
<td>20.00%</td>
</tr>
</tbody>
</table>
highlighted in the right part of the figure.

5.2.2 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 5.4. 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 5.12, 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”\(^{14}\) value changed.

5.2.3 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 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

\(^{14}\text{www.mathworks.com/help/simulink/ug/initialize-mask.html}\)
Table 5.4: Summary of Mutation Instances Witnessed in the AVS Project

<table>
<thead>
<tr>
<th></th>
<th>mMLA</th>
<th>mRUE</th>
<th>mRBL</th>
<th>mCBV</th>
<th>mADBD</th>
<th>mADBS</th>
<th>mCBT</th>
<th>mCSCH</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Total Count</strong></td>
<td>50</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>% of Total # of System Changes</td>
<td>-</td>
<td>56.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>96.00%</td>
<td>2.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
</tbody>
</table>
5.2. VALIDATING THE TAXONOMY

Figure 5.12: mCBV Mutation Operator Observed in AVS Project

shown in Table 5.5. 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 5.2.4.

5.2.4 Discussion

Overall, the significant majority of edit operations we found in the 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
Table 5.5: Summary of Mutation Instances Witnessed in the Industrial Project

<table>
<thead>
<tr>
<th></th>
<th># Of Times a System Changed</th>
<th>mMLA</th>
<th>mRUE</th>
<th>mRBL</th>
<th>mCBV</th>
<th>mADBD</th>
<th>mADBS</th>
<th>mCBT</th>
<th>mCSCH</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Count</td>
<td>83</td>
<td>67</td>
<td>38</td>
<td>27</td>
<td>50</td>
<td>32</td>
<td>32</td>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>% of Total # of System Changes</td>
<td>-</td>
<td>80.72%</td>
<td>45.78%</td>
<td>32.53%</td>
<td>60.24%</td>
<td>38.55%</td>
<td>38.55%</td>
<td>0.00%</td>
<td>1.20%</td>
<td>3.61%</td>
</tr>
</tbody>
</table>
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.\(^\text{15}\) 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 placeholder. 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 8.1. However, because we are looking only for evidence of the existence of edit operations that can be classified using our

\(^\text{15}\)mathworks.com/help/simulink/ug/annotating-diagrams.html
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 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 8.1.

5.3 Summary

In this chapter, we presented our taxonomy for Simulink model mutations intended to inject model clones. We outlined the mutation classes, provided descriptions and examples, and also presented our implementation of mutation operators that both select a random mutation target and mutate in a random way each time. We also wanted to ensure our Simulink mutations represented realistic edit operations. Thus, we performed a model evolution study whereby we observed three systems over three or more versions to see if the actual model changes from one version to the next could be classified using our taxonomy. Almost all of them could be, with the only exception being unconnected annotation or reference blocks. With our Simulink mutation creation complete, the next chapter involves the evaluation phase and how we achieve it for our Simulink implementation of the framework.
Chapter 6

Framework Evaluation Phase

In the previous chapter, we stepped through the process of devising mutations for our framework by creating and validating Simulink mutations that would allow us to adequately inject various forms of model clones into systems for our Simulink implementation. In this chapter we discuss how such information can be used in the framework by means of walking through the evaluation process, presented earlier in Section 4.3.2, as we implement it in our prototype process for Simulink model clone detectors. We begin by presenting the format that the Simulink model clone reports must be transformed to in order to facilitate evaluation. We continue by delving into the details about how we store the injected Simulink mutant information for our framework implementation and how it is used, in conjunction with each transformed clone report, to calculate recall and precision. Some of this material has been submitted to a conference on software maintenance and evolution [41].

6.1 Desired Clone Result Format for Tool Evaluation

As discussed previously in our overview of the framework in Chapter 4, once the mutation operators have been executed on the various systems and clone detection tools
are employed on these mutated systems, there will be different result sets, potentially from different tools. One of the challenges we mentioned in Chapter 4 for which there is no direct analogy in the code clone domain, is the different schemas or formats that these results may have. This will be further illustrated when we enable our framework implementation to work with different Simulink model clone tools in Chapter 7. As such, in order to calculate and compare precision and recall for tools with different output formats, we need a uniform target representation for clone results that can be interpreted by a tool report analyzer.

While conducting research for this thesis, we also engaged in corollary work that came as a result of looking into model evolution and model clone detection. Specifically, we devised a model-clone evolution tool that is able to track and present the evolution of Simulink model clones over time [39]. One contribution of this work was a representation we used to identify and track clone instances over time. For our desired Simulink clone report format in this work, we use that representation as a basis to achieve the model clone class result representation we desire. We present it, in its general form, in Listing 6.1. It contains all the necessary information to identify if a mutant was killed, yet is as simple as possible and allows for flexibility in element attributes.

In Chapter 4, we defined the notion of Fragment Containment in the modeling domain at the system level to mean having all blocks belonging to a model clone introduced by mutation be a subset of the blocks belonging to a model clone instance detected by a specific tool and/or configuration. As described previously, this aligns well with the concept of “killed” mutants in Mutation Analysis. Thus, the necessary information for calculating mutant death must include all blocks belonging to each
clone instance. Since Simulink blocks all have unique fully-qualified paths, we can gather and use paths as block identifiers in our prototype. As shown in Listing 6.1, our clone result representation begins with a `clones` element to indicate that what follows is a listing of clones. It then contains `class` elements as children with an attribute `classid` that identifies the class. We decided on including class as an element because all the Simulink model clone detector tools we encountered detect classes but not all of them explicate clone pairs. Each class element contains model clone instances, which we represent using the `source` element. Each model clone instance then must contain all the blocks belonging to it in the form of children `block` elements that contain an attribute `path` that contains the fully-qualified path to the specific block.

Listing 6.1: General Form for Simulink Clone Class Results

```xml
<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>
```

In order to have this framework be as automatic as possible, we indicated in Chapter 4 that a transformation from a tool’s native clone results format into our format would be required. Creating the transformation would need be done only once per tool and execution of the transformation on all clone results can be automated in the implementation of the specific instance of the framework. Seeing as we kept our clone result format as simple as possible, these transformation should not be overly onerous nor computational heavy. We provide details and examples of the
transformations we create for our Simulink implementation in Chapter 7.

6.2 Persisting Mutant Metadata

As noted in Figure 4.5, the location and details of the injected mutant systems must be stored by the framework in some form of persisted data for referencing later on. For our framework prototype that evaluates Simulink model clone detectors, we persist this data in the form of one XML file per SIS, such as the example in Listing 6.2, in order to realize a simple but sufficient data store. Specifically, our Mutant Metadata files have a root clones element that contains one child element called original and one-to-many elements called mutants. Both of these children elements contain a subsystem attribute that identifies the full path to the original subsystem or the one that has been mutated, respectively. They both, in turn, contain block elements that have a path attribute that has the same semantics as the block elements in our described Clone Class Result format in Listing 6.1.
6.2. PERSISTING MUTANT METADATA

Listing 6.2: Example Mutant Metadata

```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>
```

So, in order to create this metadata file in our Simulink implementation, as our Matlab program creates and injects mutants, it also writes and stores metadata about the mutants. This is done through various Matlab functions we include in Appendix A. Each time mutateSystem, Listing A.2, is called for each system to be mutated, it creates a new XML file and gives it a root clones element. The function then creates an original element to represent the original system before mutation and makes a call to our helper function, presented in Listing A.6, which adds all the appropriate block elements and attributes to the XML file, as well as the full system path attribute to the parent element. More details can be found in the appendix.
6.3 Recall Calculation and Missed Mutant Identification

Once the clone results from all the tools and specific configurations are collected and transformed into the framework’s desired format and the appropriate mutant metadata has been persisted, we can then begin the process of calculating recall and identifying both “killed” and “missed” mutants.

The denominator in the recall function of the framework corresponds to the number of mutant systems. The numerator is the model clones detected by each tool during their respective tool runs that include our injected mutant systems.

For our Simulink prototype, all the tools we encountered provide clone classes, but not all explicitly identify clone pairs. So, for the prototype’s recall calculation, we check if the original SIS and injected SIS belong to the same class. This is consistent with how it is done in the code clone domain [31]. As formalized in Equation 6.1, where $M$ is the mutant and $OS$ refers to the original, unmutated, system, for each mutant metadata file, we iterate through all mutant elements and check for their coexistence with the SIS detailed in the original element in the transformed clone report. This is done by investigating each clone class reported by each specific tool and/or configuration. Again, here we utilize the notion of fragment containment as it applies to the modeling domain to determine if a mutant and original SIS are present in a given clone class. Recall for the specific SIS with respect to an individual tool run is the number of mutants detected divided by the number of mutants contained in the mutant metadata file. That denominator corresponds to the mutants injected as they are only written to the this file upon injection. We then calculate the total recall for a tool run by summing all the mutants detected for all the SIS and divide that by all the mutants injected as shown formulaically in Equation 6.2. In this case,
6.4. PRECISION CALCULATION

MI refers to the mutants injected in the framework.

\[
Recall_{M,OS} = \begin{cases} 
1, & \text{if } M & \text{& OS belong to same Clone Class;} \\
0, & \text{Otherwise.}
\end{cases} \tag{6.1}
\]

\[
Recall_{ToolRun} = \frac{\sum_{i=1}^{SIS} \sum_{j=1}^{MI(i)} Recall_{M(j),OS(i)} \sum_{j=1}^{MI}} \tag{6.2}
\]

We also explicate the mutants that were not killed as this information is likely useful for a number of purposes including tool refinement, comparison, and error checking. Each SIS gets its own individual report file, which we show an example of in the next chapter.

6.4 Precision Calculation

In the context of this framework, precision can be described as the number of correctly identified mutant model clones divided by all the results that are returned by a specific model clone detection tool. As mentioned previously, our implementation focuses on clone classes since that is what is reported consistently across the tools we encountered. In order to accommodate this, we modify the strategy employed in the code clone tool evaluation framework [31] in order to make it work on models. Specifically, we form model clone pairs from the constituents in each model clone class and use that in the precision calculation as the denominator.

For the numerator, we validate clone pairs analogously to what is done in the code clone framework. In their case, they validate clone pairs by doing a line by line comparison. In our case, we exploit the fact that we are working in the context of systems. Because our mutation operators are single operations, each mutated system
6.4. PRECISION CALCULATION

will have, at most, two block differences with respect to any other clone in its class: Upwards of two differences in the case of a clone pair containing two mutated systems, and upwards of one difference in the case of a clone pair with the original system and a mutant. The only exception to this is when we have mutated the system hierarchy by injecting a subsystem, since there are a number of different blocks generated to make that happen. So we account for that in our model clone pair validator by validating pairs that have only 30% or less difference in the identified blocks, as we previously established this as a reasonable difference threshold [4]. A key point to remember here is that our model clone pair validator is not a clone detector. Rather, similar to the code-clone evaluation framework, it is validating clones by utilizing the fact that we are validating a single pair and are aware of the specific mutations being injected.

We validate each pair of model clone instances in a model clone class by doing a block-by-block comparison allowing for Type 1, 2, and 3 differences. The number of valid clone pairs will be used as our numerator in the precision calculation. As demonstrated in Equation 6.3, total precision can then be calculated by a framework implementation as the summation of all valid clone pairs over all reported clone pairs across all SIS.

\[
\text{Precision}_{\text{ToolRun}} = \frac{\sum_{i=1}^{SIS} CP(i)_{\text{valid}}}{CP(i)_{\text{reported}}} \tag{6.3}
\]

Similar to what we did for recall, we explicitly indicate the invalid clone pairs to allow tool developers and evaluators to see and investigate any invalid pairs. This appears in the individual SIS evaluation report file that we show an example of in the next chapter.
6.5 Summary

In this chapter, we provided an overview of the framework evaluation phase and included how we accomplished it in our Simulink implementation. We began by defining our desired clone report format and the process by which we persisted the required metadata information for mutants. We then defined recall and precision as it applies in the framework and illustrate how we calculated both in our Simulink framework prototype. Recall involves dividing the number of mutants injected by the number of mutants killed, while precision is equal to the number of valid clone pairs divided by the number of clone pairs reported. With the mutation and evaluation phases implemented for Simulink, the next chapter describes our experiments in evaluating Simulink model clone detectors using the framework.
Chapter 7

Simulink Framework Implementation Results

In the previous chapter, we described how the evaluation phase of the framework can be realized by describing our Simulink prototype’s implementation of that phase. In this chapter, we present what a completed instance of the framework can actually do. We present our experiments in evaluating different tools and their configurations. We run our experiments on both Simone and ConQAT as they are the most mature Simulink model clone detection tools available. In addition, when we contacted other Simulink model clone detection tool makers about acquiring their tools for experimentation they were either discontinued and unavailable, like eScan and aScan [29]; or not for public release, like Naive Clone Detector [28].

We begin by summarizing the process as a whole for each new tool run that we evaluate. This includes the transformations we needed to create to allow for each specific tool to work with our framework. We then introduce and describe the models and mutations we employed when evaluating Simone and ConQAT. Subsequently, using these models and mutations, we present a quantitative evaluation of the tools and configurations, and introduce interesting cases of missed mutants and invalid clone pairs we encounter. Much of this chapter has been submitted to a conference.
7.1. SPECIFIC PROCESS FOR PROTOTYPE

on software maintenance and evolution [41].

7.1 Specific Process for Prototype

Once the SIS have been automatically selected and prepared, as laid out in the mutation phase in Section 4.3.1, the framework then involves running and evaluating the different tool runs as discussed in Section 4.3.2. For our implementation, this began with us creating transformations for both Simone and ConQAT clone reports, which we describe in this section. After those were ready and tested, we executed the clone detectors on all the SIS. The transformations were then performed on the clone reports and manually organized so they could be fed into our evaluation program. Figure 7.1 illustrates an example of how we organized our files for our implementation. The root folder, “powerwindow”, representing the model contains mutant metadata files, each representing a system that was mutated. Each folder underneath the root folder represents a tool run and must contain transformed clone reports that have the same name as the system that was evaluated in order to link them with the appropriate mutant metadata file. There can be as many folders as necessary for each tool run and the names of the folders do not matter, as long as they are unique.

After we organized the reports and mutant metadata in this fashion for each tool run, we were ready to start evaluating the tools by executing the automatic analysis program we created. For each SIS and tool run, a corresponding “.tooleval” file is generated that indicates the recall and missed mutants, and precision and invalid clone pairs for each individual system. This is possible because we inject only a single mutant to a single system in isolation. In addition, the total recall and precision for that tool run, across all SIS, is output to the user. We present examples of this later.
7.1. SPECIFIC PROCESS FOR PROTOTYPE

7.1.1 Simone Report Transformation

The Simone clone reports are made available in both HTML and XML format, with or without the model source, and with or without clone classes. The option closest to what we require for our framework prototype implementation is XML format, with source, and with the clones sorted into classes. During our model clone evolution research alluded to earlier [39], we wrote a transformation to put the Simone results in a format conducive to evolution analysis. This format was very similar to what we

Figure 7.1: Prototype File Organization of Clone Reports and Mutant Metadata on in this chapter.

7.1.1 Simone Report Transformation

The Simone clone reports are made available in both HTML and XML format, with or without the model source, and with or without clone classes. The option closest to what we require for our framework prototype implementation is XML format, with source, and with the clones sorted into classes. During our model clone evolution research alluded to earlier [39], we wrote a transformation to put the Simone results in a format conducive to evolution analysis. This format was very similar to what we
required, so we used it as a starting point and made slight modifications to it. We present the transformation in Appendix B in Listing B.1.\textsuperscript{1} Because of the relatively complex nature of this transformation, we used the powerful TXL transformation language \[13\], like we did for the mRUE mutation previously. At a high level, we utilize our Simulink grammar, which we have not made public yet, and create a grammar representing the Simone reports. We then use these grammars to transform Simone reports to have them include the fully qualified paths to the blocks, which is required by our framework, rather than the raw source, and also format the XML attributes to make sure they have their special characters properly escaped so the files can be processed as valid XML.

### 7.1.2 ConQAT Report Transformation

ConQAT’s Simulink model clone reports are in XML format and are made viewable through an HTML interface. ConQAT’s model clone reports differ significantly from Simone reports, which, as discussed, was a large part of the original motivation for our framework. Specifically, ConQAT reports clone classes only, termed “finding-group”s; and has each clone instance, or “finding”; contain its respective Simulink blocks, represented as “qualified-name” elements with path attributes. Because these reports were in a format much closer to our desired clone report format described in Section 6.1 we decided to write a relatively simple, ten-line, shell script rather than use TXL.\textsuperscript{2} For this script, we use a combination of the Unix stream editor and Perl commands, with printing loops, to turn “finding groups” into “classes”, “finding” elements into “source” elements, and to modify the blocks appropriately.

\textsuperscript{1}Transformation created mostly by Andrew Stevenson
\textsuperscript{2}Script written by Andrew Stevenson
7.2 Experiment Models and Mutations

The mutation operators we implemented mutate randomly each time. For example, we mutate a random block in a system, add a block between two random blocks or as a destination from a random line, delete a random block, et cetera. As such, the models we select should not have too much impact on the evaluation of tools. However, it is important to see how the mutations effect different systems and systems of varying sizes. For our experiments with our prototype we select models from the PW and AVS projects presented earlier in Section 5.2. We forgo including our industrial system in this specific case because 1) we want to present and discuss models from these systems, and 2) the specific set of models we received from our industrial partners were not ideal for mutation because some of the model files contained many systems that were not conducive to our mutations. This was because many of those systems were small, modular, systems (three or less blocks), reference systems, or placeholders for future systems. Since our framework selects systems in a given model randomly, the AVS system is on scale with our industrial system, and the specific models being mutated should not really have an impact on the mutations themselves, we believe this is not a concern in terms of demonstrating our framework or the tool evaluation.

Table 7.1 displays information about the models we mutate to demonstrate our prototype. The “Project” column indicates what project the model belongs to, while the “Model” column indicates what specific models were used. The PW project has only one model, so we decided to use two different versions of that. For the AVS system, we decided to go with two of the larger library models that seemed to
7.3. CONQAT RESULTS

Table 7.1: Models Mutated by Prototype for Tool Evaluation

<table>
<thead>
<tr>
<th>Project</th>
<th>Model</th>
<th># Systems Mutated</th>
<th># Mutations Injected</th>
</tr>
</thead>
<tbody>
<tr>
<td>PW</td>
<td>PowerWindow(V1)</td>
<td>7</td>
<td>83</td>
</tr>
<tr>
<td></td>
<td>PowerWindow(V3)</td>
<td>13</td>
<td>153</td>
</tr>
<tr>
<td>AVS</td>
<td>fc_KTH_lib</td>
<td>12</td>
<td>146</td>
</tr>
<tr>
<td></td>
<td>lib_fuel_Cell</td>
<td>13</td>
<td>164</td>
</tr>
</tbody>
</table>

contain rich systems. The “# of Systems Mutated” column indicates the number of randomly selected systems to be mutated that were selected by the Matlab function we created in Listing A.3. We had our program select a dozen or so systems from each model, if that many existed. We could have selected more, however the key number is the amount of mutations injected, which is listed in the “# of Mutations Injected” column. With all but the first version of the original and relatively small PowerWindow model, we ended up with roughly 150 random mutations injected from 14 different mutation operators for each model, for a total of 546 mutations. For this column, while it may seem like we can simply multiple the number of systems selected randomly by the mutations we implemented, fourteen, we have to remember that 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. That being said, there were more than enough systems that had all fourteen mutations injected successfully.

7.3 ConQAT Results

Table 7.2 illustrates the evaluation of ConQAT’s handling of our mutated models. It is very important to mention that the way ConQAT is implemented at this time, it is only capable of detecting type 1 and some type 2 clones. It would be possible
Table 7.2: Results of ConQAT Evaluation

<table>
<thead>
<tr>
<th>Model</th>
<th>Recall</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>PowerWindow(V1)</td>
<td>33%</td>
<td>99%</td>
</tr>
<tr>
<td>PowerWindow(V3)</td>
<td>23%</td>
<td>100%</td>
</tr>
<tr>
<td>fc_KTH_lib</td>
<td>34%</td>
<td>100%</td>
</tr>
<tr>
<td>lib_fuel_Cell</td>
<td>35%</td>
<td>89%</td>
</tr>
</tbody>
</table>

for them to eventually detect near-miss type 3 clones if they refined their tool, as we discussed in our initial evaluation paper [4]. While we were aware of this early on, having this framework in place allows us to actually quantify it. Because roughly two thirds of our mutations are type 3, it is expected that ConQAT’s recall will be about 33%.

We present the evaluation of ConQAT’s clone detection on the mutated detect_endstop system from the first version of the PowerWindow model in Figure 7.2. As expected, the tool was unable to detect any of the type 3 mutations including adding or deleting a block in the middle, or adding or deleting a block as a source or destination. In terms of the changing block type mutation, “detect_endstop_mCBT”, this is also expected because ConQAT uses block type as the discerning element to distinguish between two blocks and, since they are incapable of detecting near-miss system clones, this mutant would not be killed. It was able to detect the type 2 mutations of renaming a block and renaming a line. One missed mutant of interest is the changing of a block’s value. ConQAT’s algorithm normalizes blocks into a label containing information they deem relevant for similarity comparison, which sometimes includes the block’s value and sometimes does not [17]. In this specific mutation, our mutation operator mutated the original detect_endstop system by changing a constant block’s numerical value. This caused ConQAT to view that block as a completely
7.3. CONQAT RESULTS

Figure 7.2: Evaluation of ConQAT on Powerwindow/detect_endstop System

Recall:
0.385
Mutants Missed:
detect_endstop_mCBV/Subsystem
detect_endstop_mCBT/Subsystem
detect_endstop_mAIB/Subsystem
detect_endstop_mDIB/Subsystem
detect_endstop_mCAB/Subsystem
detect_endstop_mCABS/Subsystem
Precision:
1.0

different block and, because they match only exact systems and not near-miss, they missed this mutant. While having that block be different is an implementation decision on their part, it is clear that these two systems should be reported as, near-miss, clones of one another.

Another interesting observation we made upon evaluating ConQAT using our framework occurred when we noticed that there were a few instances where ConQAT did not kill a mRB rename block mutation. For example, consider the “CathodeOver-potential” system in the fc_KTH_lib model from the AVS project. Specifically, the mutated system we display in Figure 7.3 in which the highlighted block was randomly selected and had its name changed from “po”. What we learned in this case is that ConQAT does not consider nor include inner inport and outport blocks from systems in their evaluation and results. Specifically, during their flattening algorithm, where they remove all hierarchy, they ignore the inner “version” of the ports and use the outer versions that are connected to the subsystem being flattened. In this case, the
inner output was mutated but disregarded in the clone report. As such, while the inner blocks with the higher level ports were reported in a related “exact” clone instance, this mutant was appropriately considered missed because this mutated block was not reported and, thus, does not fit our model-based definition of fragment containment since not all blocks are included. This is important in this case, because the block being omitted is actually the block of interest.

ConQAT’s lower precision on the lib_fuel_Cell can be accounted to the fact that there are some very small systems that were included in the random system selection, which we elaborate on in the next section, and, thus, even one block change can result in systems that were more than 30% different according to our clone pair validator.

When using ConQAT to detect subsystem clones, there are no configurable settings. Thus we execute ConQAT only once on our models.
7.4 Simone Results

7.4.1 Simone with Default Settings

Table 7.3 presents the analysis of Simone’s clone detection on the models from our experiments using the default settings we established previously [4]. Seeing as Simone is intended to detect all three types of clones, one would hope that it has relatively high recall.

A number from the table that immediately jumps out is the 77% precision in the lib_fuel_cell model. Upon further investigation of this, we notice that there are a number of systems that were randomly selected and mutated that had zero clones reported by Simone, thus bringing down the recall significantly. One of the tunable parameters in Simone is the minimum number of source lines that must comprise a system in order for it to be considered for clone detection. During our construction of Simone and early experimentation, we decided that systems clones with less than 100 lines were likely trivial and not very prevalent in larger systems. While the former point is debatable, it should be noted that three of the thirteen systems that were randomly selected from that model were less than 100 lines: The “CoolantPump”, “FuelPump”, and “Vent Purge” systems. So, these systems had no clones reported and thus had zero recall. Whether or not it is appropriate to configure the minimum number of lines to be lower for system clones, which is a semantic question for engineers, the key takeaway here is that the question arose because of metrics reported by our framework prototype. So, after executing our prototype on a specific set of models, engineers can decide if they want to configure Simone differently.

The corresponding tool evaluation report for Simone’s handling of the detect_endstop System, corresponding to ConQAT’s in Figure 7.2, can be seen in
### 7.4. SIMONE RESULTS

Table 7.3: Results of Simone Evaluation

<table>
<thead>
<tr>
<th>Model</th>
<th>Recall</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>PowerWindow(V1)</td>
<td>98%</td>
<td>100%</td>
</tr>
<tr>
<td>PowerWindow(V3)</td>
<td>99%</td>
<td>95%</td>
</tr>
<tr>
<td>fc_KTH_lib</td>
<td>97%</td>
<td>94%</td>
</tr>
<tr>
<td>lib_fuel_Cell</td>
<td>77%</td>
<td>97%</td>
</tr>
</tbody>
</table>

Figure 7.4: Evaluation of Simone on Powerwindow/detect_endstop System

Recall:
1.0

Precision:
1.0

Figure 7.4. In this case, Simone had perfect recall and precision.

Although recall was quite high overall for Simone, one class of mutations that were sometimes, but not always, missed was the changing subsystem mutation, mCSCH. This included systems that had varying numbers of blocks extracted into subsystems. Figure 7.5 displays the non-mutated system “SaturationTemperature” from the fc_KTH_lib. We present this example because, in this case, the mCSCH mutation, which selects a random number of blocks from a subsystem, ended up selecting three blocks to extract into a subsystem. Specifically, the prototype selected the three highlighted blocks to extract. Simone was unable to identify these two systems as clones despite the fact that only a single Matlab operation was executed and that all the blocks are still present. While Simone was able to detect other subsystem clones with three or more blocks extracted, it is understandable that Simone is unable to kill this mutant clone because of all the additional gluing that is done in extracting the blocks into a subsystem and the specific blocks selected in this case. However, a clone should still be identified because the two systems are still relatively
similar, with one block replacing three. What this indicates is that there might be too much textual noise in Simone's normalized form when accounting for embedded subsystems. There are two ways of addressing this, 1) either tweak and try to reduce noise in Simone’s normalization facility, or 2) loosen Simone’s configurable similarity threshold for near-miss clones to allow for pairs/classes with more variation. Simone can pair clones with more than 30% difference, however, its clustering algorithm can only cluster related pairs with up to 30% difference due to the way it does its line comparison. So the former option would likely be the better approach and could be started by doing a line by line inspection of the original system and the mutated one, in their normalized forms, to see what noise exists.

As discussed in the previous chapter, precision in our prototype was implemented by a clone pair validator that exploits the knowledge of the system context and the mutations. Thus, any invalid clone pairs would indicate two systems that were shown to be at least 70% identically textually, but not by their blocks. Simone, and ConQAT, did very well in regards to their precision. For Simone, there were only a small number of invalid clone pairs, all of which were related to subsystem hierarchy mutations that had matched that were invalidated because of the number of additional blocks added. For example, a clone pair involving a system that had a block removed and one that had blocks extracted into a subsystem would have a notable amount of pure block (path) differences, which is what our validator is based on. Thus, these very few cases of clones pairs that were invalidated are more a result of our clone pair validator falling victim to the same problem as Simone in that subsystem extraction is weighted too heavily. This is acknowledged further in future work in Section 8.1.
Figure 7.5: The Saturation Temperature System Before Mutation


Table 7.4: Results of Simone Evaluation at 20% Difference

<table>
<thead>
<tr>
<th>Model</th>
<th>Recall</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>PowerWindow(V1)</td>
<td>96%</td>
<td>100%</td>
</tr>
<tr>
<td>PowerWindow(V3)</td>
<td>96%</td>
<td>98%</td>
</tr>
<tr>
<td>fc_KTH_lib</td>
<td>94%</td>
<td>97%</td>
</tr>
<tr>
<td>lib_fuel_Cell</td>
<td>77%</td>
<td>98%</td>
</tr>
</tbody>
</table>

7.4.2 Simone with 20% Difference Threshold

While part of our motivation in developing this framework was to provide a facility to compare and contrast tools, we also wanted to allow for tool developers to refine their own tools. As such, we use our Simulink prototype to demonstrate this ability. While we could have adjusted the min lines parameter discussed earlier, we thought it would be more interesting to adjust the main parameter, the near-miss difference threshold. Specifically, we restrict Simone by configuring it to accept only 20% differences among systems rather than the default 30%. We choose 20% because, during our code- and model- clone experiments, we found that 20% still revealed interesting type 3 clones. Any lower setting resulted in trivial clones and anything much higher is too similar to 30%. The results are presented in Table 7.4 and can be directly contrasted with Table 7.3. In general, when something is more particular about what is taken in, it will likely experience a decreased recall and an increase in precision [11]. This holds true with our experiment in changing Simone’s main parameter. Comparing the two tables, we see a total of eight recall percentage points dropped across all four models and an increase of seven precision points.

As expected, a big reason for the precision increase in this case is the reduction of clone pairs and even, in some cases, clone classes. For example, in the “CathodeOutletSystem” in the fc_KTH_lib model, default Simone detected fifty six clone pairs and
clustered them into three classes. Simone with 20% detected forty six clone pairs and clustered them into two classes. One of the classes that was dropped included a 74% similar class. In this case, the recall for this system remained the same. However, other systems, like the “CathodeOxygenMolarFraction” system from the lib_fuel_cell model, had its recall go from 100% to 92% but saw its precision increase to 100%. So, as is often the case, we need to make a decision between what is more important to us, recall or precision, given the specific context/usage of Simone at the time. While it is no surprise that this trade off exists, at least the framework and the Simulink implementation of it we created for this thesis will allow model clone detector users to better quantify that decision.

7.5 Summary

This chapter presents our experiments in using the framework to evaluate two Simulink model clone detectors, Simone and ConQAT as well as different configurations of Simone. We begin by outlining the steps we took in our Simulink implementation including the two clone report transformations for the two different tools. We then introduce the models and mutations used in our experiment, which included four models, forty five systems, and 546 mutations.

Figure 7.6 summarizes the recall for each tool run. Simone’s recall was typically in the high nineties, except for the one exceptional case caused by models that were deemed too small to be considered by Simone. 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.
Figure 7.6: Recall (%) of Each Tool Run

Figure 7.7 provides an overview of the precision for the tool runs in our experiment. Both ConQAT and Simone have very high precision. Using our framework, we also identified the precision and recall trade off in reducing Simone’s difference threshold from 30% to 20%.

We believe experiments like these can help fuel many interesting developments in model clone detection. In the next chapter, we present future work and conclude the thesis.
Figure 7.7: Precision (%) of Each Tool Run
Chapter 8

Future Work and Conclusion

Having described the framework, walked through the process of implementing it, and ran experiments with it, this chapter illustrates some future research topics that we thought of along the way. In addition, we provide our overall conclusions on this work.

8.1 Future Work

In this section we discuss interesting areas of future work and research questions that arose from the research performed for this thesis. Section 8.1.1 was published previously during our original presentation of the evolution study [38].

8.1.1 Automating the Evolution Study

The first area of future work involves completely automating the process that was performed semi-automatically in our evolution case study in Section 5.2. Because of the issues with the Matlab Simulink Report Generator XML comparison tool and since its internal workings are proprietary, we would have to perform quite a bit of tweaking on the output of the tool to try to account for the errors. We would need to
8.1. FUTURE WORK

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 in this thesis; Almost like a grammar for mutation matching. We held off on doing that for now as we needed only to see if our mutation taxonomy was applicable in the various MDE projects we had access to for the purpose of our model-clone detection evaluation framework and prototype we build in this thesis.

It would be beneficial to have access to more multi-version Simulink projects, especially from industry, and do more evolution studies on them. This would allow us to further validate our proposed taxonomy and would help improve its generality, allowing it to be more of a general Simulink mutation taxonomy rather than one intended only for injecting clones.

8.1.2 Semantic Clones

As alluded to in Chapter 3, type 4 semantic model clones for Simulink is a relatively unexplored area that could be incorporated into our prototype, and the framework in general, once more work is done on it. The only preliminary work on these types of models clones involves transforming existing Simulink models into their equivalent forms while preserving the semantics [3]. With regards to the framework and even our Simulink prototype, type 4 clones could be treated the same way as the rest of the clones types in that they can be injected and validated for recall and precision, respectively. A key difference however, is that we would have to first do a search on the model, or sets of models, to find a valid source model to mutate since not all the semantic preserving transformations would work on every system. This more aligns with the definition of a model transformation than mutation, so this would
require more of a dual approach. It is definitely possible though, since even some of the mutations we implemented thus far have some constraints on the systems they operate on, for example, the deleting a block in between mutation, changing a block’s value, and changing a block’s type. However, our constraints are likely not as complex as those involved in finding a source for a semantic preserving transformation would be.

8.1.3 Other Types of Models

Aside from choosing Simulink because it was of interest to our industrial partners, we also focused on it because Simulink model clone detection was far and away the most mature type of model clone detection. As discussed in our background chapter, Chapter 2, there have been attempts at other model types like UML in general [42], behavioral models [5], and state machines [12]. However there are not many tools for these model types, rather, most model types have a single approach at this time.

We attempted to make the framework as generic as possible, while also presenting a specific implementation made for Simulink. In terms of the mutation phase outlined in Figure 4.4 the same process would apply to any model type once an appropriate granularity was selected as the focus of the mutations. So, perhaps, for UML structural models, each class diagram could be considered a system. Multiple class diagrams would be duplicated, mutated, and organized similarly. The evaluation phase, presented in Figure 4.5, would apply the same way to any types of models.

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

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

### 8.1.4 Refining Precision and Clone Pair Validation

Similar to what was done in the code-clone tool evaluation approach [31], we had to create a model clone validation process for model clone pairs. However, validation in the modeling domain was slightly more complex. Rather than just comparing text similarity of reported clones, we had to determine how to validate two systems. Considering the only consistent information reported across tools, albeit not always explicitly, was block paths, we had to use those. As explained in Chapter 6, this works since we are aware of the systems we are working with and the nature of the mutations occurring. That being said, this validation technique is not perfect. It would be better if we could devise some heuristic that is not clone detection, but could traverse the models quickly to validate a clone pair thus using structure, including block type, rather than qualified block paths. This is not a trivial task, however.
8.2 Conclusion

In this thesis, we introduced problems that existed in objectively and easily evaluating model clone detectors. We encountered these problems while trying to assess our own model clone detector that works on Simulink models, Simone. These issues involve not knowing what clones should be detected, or recall computation; the varied nature of clones reported by different tools, where some reported nested clones and some do not; and the dissimilar output clone report formats/schemas used by different tools.

In order to address these issues, we devised a Mutation Analysis based framework for quantitatively evaluating model clone detectors. The purpose of such a framework is to allow for automatic and statistical comparison between different tools or different configurations of the same tool. We break down the framework into two phases: mutation and evaluation. The first phase involves executing an appropriate set of randomized mutation operators, covering all mutation classes, on systems. These systems can be selected randomly or manually. The evaluation phase then involves using the knowledge of where and how the mutations were injected during the mutation phase to evaluate the model clone detectors. This knowledge allows for recall and precision calculation. Recall becomes a matter of seeing if the model clones injected are reported by a respective tool. Precision involves validating the pairs or classes of clones being reported.

To demonstrate the feasibility of the framework, we develop an implementation of it intended to evaluate Simulink model clone detectors, the most mature and prevalent kind of model clone detector. A key and necessary part of the framework is devising mutations that inject model clones. In addition, it is ideal if these mutations represent realistic edit scenarios based on observed model evolution. We devised a taxonomy of
eight Simulink mutation classes that inject Simulink model clones. We then validated these classes through a model evolution study whereby we looked at how our mutation classes fit with witnessed evolution in three projects across multiple versions: Two open source projects and one industrial project. Overall, our mutation classes ended up fitting well with the changes exhibited by the various model projects. The only exceptions were unconnected annotation or reference blocks. We implemented our Simulink mutations using the native Matlab programming language, as this best mimics how the models would actually be cloned and edited. The only mutation class we do not implement in this way and use TXL for, is the one that involves mutating the underlying textual representation of a model. We provide and discuss our mutation implementations in the respective sections describing each mutation class.

For the evaluation of tools, it is necessary to have clone reports in a format conducive for evaluation and comparison. This was a major hurdle during our original attempts in comparing our tool with others and refining it. In this thesis, we elaborate on this idea and walk through the process we took in our Simulink implementation to achieve it. This begins by outlining our described format, which contains all the information we need yet still remaining relatively simple. It is essentially an XML format that organizes clone reports into classes with clone instances that list out each comprising block along with its full path. We decided on classes because each clone tool we encountered reported classes but not all explicitly reported clone pairs. Then, consistent with our outlined framework’s evaluation phase, we devised transformations that turn respective tools’ clone reports into our format. For Simone, this was slightly more difficult than ConQAT because we had to turn the raw model source
into blocks. We then described how reports in our desired format can be analyzed to ascertain recall and precision. Recall for a specific execution of a tool involves summing all the mutants killed across all systems over all the mutants injected for all systems. Precision involves validating all the clones pairs reported by a specific tool run and dividing the valid clone pairs by the total number of clone pairs across all systems.

With our Simulink framework implementation complete, we could now run experiments with it. Using four models, and forty five systems from those models, we inject 546 mutations. Each mutation injection is random in that it selects a random element to mutate in a system and mutates in a randomized way, if applicable. We then perform three tool runs: ConQAT, Simone with default settings, and Simone with a 20% difference threshold. As expected, ConQAT was able to detect only type 1 and some type 2 clones. Sometimes it considers a block’s value for block comparison and sometimes it does not. We provide an example of the former. In addition, their flattening algorithm disregards the inner version of an inport or outport, which caused them to miss some mutants they likely should have killed. Simone with default settings achieved very high recall and precision. One lower recall score was caused by the fact that the default Simone settings involves disregarding smaller subsystems. This was a design decision and can still be argued either way. There were a small number of cases were Simone missed mutants that involved changing a system’s inner hierarchy. We believe this is a factor of textual noise associated with embedded subsystems and can be resolved through refinement of Simone’s normalizing and filtering process. Running Simone with 20% difference threshold, in contrast to 30% difference, yielded expected results in that increasing the precision lowered the recall, fairly equally in
this case. The nice aspect of our framework is that this was now quantified and would allow engineers and Simulink model clone detector users to tweak configuration values in order to achieve desired results for a specific context.

There were a number of interesting areas of future work that arose from our research. Performing more validation on our Simulink taxonomy of mutations of model clones by obtaining more models and doing more experiments would help solidify and perhaps generalize it. Incorporating type 4 semantic Simulink clones would be a nice addition to the framework, involving more of a model transformation approach than mutation one, once that area is more mature and Simulink clone detectors are updated to detect type 4 clones. Also related to maturity of the field is implementing our framework for other types models once more research is conducted and tools are developed for other model types. Lastly, while our clone pair validation approach is adequate, it could definitely be refined by means of a model traversal heuristic that does not redo model clone detection, but rather validates using a priori knowledge.

We believe that we have contributed to the model driven engineering field by creating a framework that uses mutation analysis in order to achieve quantitative analysis of model clone detectors. In addition, we developed a Simulink implementation of that framework that works on Simulink model clone detectors to demonstrate its feasibility. This included a taxonomy and implementation of Simulink mutations that cover the various model clone types and are also relatively realistic when it comes to actual engineering operations on Simulink models. We hope that having a framework of this nature, and Simulink implementation of it, will help cultivate further research gains in Model Clone Detection, allowing the area to grow and flourish.
Bibliography


Appendix A

Additional Mutation Code Employed in Prototype

A.1 Full Reorder Underlying Elements Mutation Function

Listing A.1 provides the complete TXL transformation for randomly reordering two textual elements in a system.\(^1\) It is discussed in detail in Section 5.1.1 of the thesis.

Listing A.1: TXL Code for Reordering Textual Elements

```txl
1  include "random.rul"
2  include "simulink.grm"
3  include "stringhelper.rul"
4
5 redefine system_list
6   System { [NL][IN]
7      [repeat default_single_element]
8      [repeat compound_element] [EX]
9   } [NL]
10  end redefine
11
12 define compound_element
13   [block_list]
14   | [annotation_list]
15   | [line_list]
16  end define
17
18 function main
19   replace [program]
20      P [program]
21   construct RandomInit [number]
```

\(^1\)Created by Andrew Stevenson
A.1. FULL REORDER UNDERLYING ELEMENTS MUTATION
FUNCTION

    _ [randinit]
    import TXLargs [repeat stringlit]
    deconstruct TXLargs
        SystemPath [stringlit]
    construct SystemPathSegments [repeat stringlit]
        _ [split SystemPath "/"]
    by
        P [mutateSystem SystemPathSegments]
    end function

rule mutateSystem PathSegments [repeat stringlit]
    deconstruct PathSegments
        SystemName [stringlit] RestOfPath [repeat stringlit]
    replace $ [system_list]
        System {
            Name SystemName
            SingleElements [repeat default_single_element]
            ListElements [repeat compound_element]
        }
    by
        System {
            Name SystemName
            SingleElements
            ListElements [swapBaseCase PathSegments]
    [mutateSystem RestOfPath] % strip off the path head
            and call recursively with tail
        }
    end rule

function swapBaseCase PathSegments [repeat stringlit]
    deconstruct PathSegments
        _ [stringlit] % base case active during last path segment only
            (i.e. targeted system)
    replace [repeat compound_element]
        Elements [repeat compound_element]
    by
        Elements [swapRandomPair]
    end function

function swapRandomPair
    replace [repeat compound_element]
        Elements [repeat compound_element]
    construct Length [number]
        _ [length Elements]
    construct Rand1 [number]
        _ [rand Length]
    construct Rand2 [number]
        _ [rand Length]
    construct E1Seq [repeat compound_element]
        Elements [select Rand1 Rand1]
    construct E2Seq [repeat compound_element]
A.2 ADDITIONAL MUTATION EXECUTABLES AND HELPER FUNCTIONS

Elements [select Rand2 Rand2]

deconstruct E1Seq

E1 [compound_element]

deconstruct E2Seq

E2 [compound_element]

construct Temp [compound_element]

Line { }

by

Elements [$ E1 Temp]

[$ E2 E1]

[$ Temp E2]

end function

A.2 Additional Mutation Executables and Helper Functions

Listing A.2 is the major function in our mutation framework. Given a specific system, it then steps through and executes each of the mutations we have created. We tried to extract out as many helper functions as possible in order to avoid redundancy.

Listing A.2: Function to Mutate a Specific System

function f = mutateSystem( subSystemToMutate )

% This is the high level mutate function for the SMCD Evaluator framework.
% It extracts the specified system and makes
% a copy of it as its own model file for each mutation that will be made.
import matlab.*;
try
    blockDiagram = subSystemToMutate;
    indexOfSlash= strfind(subSystemToMutate,'/');
    if numel(indexOfSlash) ~= 0
        blockDiagram = strtok(subSystemToMutate, '/');
    end
    load_system(blockDiagram);
    mkdir(blockDiagram);
    cd(blockDiagram);
    % start building XML
    % start building the mutationRepositoryfile
    XMLdocument = com.mathworks.xml.XMLUtils.createDocument('clones');
    clones = XMLdocument.getDocumentElement;
    if (strcmp(get_param(subSystemToMutate,'type'), 'block_diagram'))==1
        disp('The provided subsystem is a Block Diagram. Use
             mutateBlockDiagram instead.');
    elseif (strcmp(get_param(subSystemToMutate,'type'), 'block'))==1 &&
        strcmp(get_param(subSystemToMutate,'BlockType'), 'SubSystem'))==1
        % If the selected subsystem is a subsystem block.
resolvedSystemName = get_param(subSystemToMutate,'Name');
resolvedSystemName = genvarname(resolvedSystemName);
if (length(resolvedSystemName) >= 63)
    % Can't work with system names longer than 63 characters.
    return;
end
mkdir(resolvedSystemName);
% make a duplicate of our original SIS
newModelName = createNewModelWithSubsystem(resolvedSystemName,
    subSystemToMutate,'');
original = XMLdocument.createElement('original');
pathToNewSubsystem = strcat(newModelName,'/Subsystem');
addSystemBlocksToXMLContainer(XMLdocument, original,
    pathToNewSubsystem);
clones.appendChild(original);
close_system(newModelName);
% execute mutation mRB
newModelName = createNewModelWithSubsystem(resolvedSystemName,
    subSystemToMutate,'_mRB');
pathToMutatedSubsystem = strcat(newModelName,'/Subsystem');
mutRenameBlock(newModelName, pathToMutatedSubsystem);
addMutantXML(XMLdocument, clones, pathToMutatedSubsystem);
close_system(newModelName);
% execute mutation mRL
newModelName = createNewModelWithSubsystem(resolvedSystemName,
    subSystemToMutate,'_mRL');
pathToMutatedSubsystem = strcat(newModelName,'/Subsystem');
if (mutRenameLine(newModelName,resolvedSystemName,
    pathToMutatedSubsystem))
    addMutantXML(XMLdocument,clones,pathToMutatedSubsystem);
end
close_system(newModelName);
% execute mutation mBC
newModelName = createNewModelWithSubsystem(resolvedSystemName,
    subSystemToMutate,'_mBC');
pathToMutatedSubsystem = strcat(newModelName,'/Subsystem');
mutChangeBlockColour(newModelName, pathToMutatedSubsystem);
addMutantXML(XMLdocument,clones,pathToMutatedSubsystem);
close_system(newModelName);
% execute mutation mBP
newModelName = createNewModelWithSubsystem(resolvedSystemName,
    subSystemToMutate,'_mBP');
pathToMutatedSubsystem = strcat(newModelName,'/Subsystem');
mutChangeBlockPosition(newModelName, pathToMutatedSubsystem);
addMutantXML(XMLdocument,clones,pathToMutatedSubsystem);
close_system(newModelName);
% execute mutation mCBV
newModelName = createNewModelWithSubsystem(resolvedSystemName,
    subSystemToMutate,'_mCBV');
pathToMutatedSubsystem = strcat(newModelName,'/Subsystem');
if (mutChangeBlockValue(newModelName, resolvedSystemName, pathToMutatedSubsystem))
    addMutantXML(XMLdocument, clones, pathToMutatedSubsystem);
end

close_system(newModelName);

% execute mutation mCBT
newModelName = createNewModelWithSubsystem(resolvedSystemName, subSystemToMutate, '_mCBT');
pathToMutatedSubsystem = strcat(newModelName, '/Subsystem');
if (mutChangeBlockType(newModelName, resolvedSystemName, pathToMutatedSubsystem))
    addMutantXML(XMLdocument, clones, pathToMutatedSubsystem);
end

close_system(newModelName);

% execute mutation mABD
newModelName = createNewModelWithSubsystem(resolvedSystemName, subSystemToMutate, '_mABD');
pathToMutatedSubsystem = strcat(newModelName, '/Subsystem');
mutAddBlockAsDest(newModelName, pathToMutatedSubsystem);
addMutantXML(XMLdocument, clones, pathToMutatedSubsystem);
close_system(newModelName);

% execute mutation mDBD
newModelName = createNewModelWithSubsystem(resolvedSystemName, subSystemToMutate, '_mDBD');
pathToMutatedSubsystem = strcat(newModelName, '/Subsystem');
if (mutDeleteBlockAsDest(newModelName, resolvedSystemName, pathToMutatedSubsystem))
    addMutantXML(XMLdocument, clones, pathToMutatedSubsystem);
end

close_system(newModelName);

% execute mutation mDBS
newModelName = createNewModelWithSubsystem(resolvedSystemName, subSystemToMutate, '_mDBS');
pathToMutatedSubsystem = strcat(newModelName, '/Subsystem');
if (mutDeleteBlockAsSource(newModelName, resolvedSystemName, pathToMutatedSubsystem))
    addMutantXML(XMLdocument, clones, pathToMutatedSubsystem);
end

close_system(newModelName);

% execute mutation mAIB
newModelName = createNewModelWithSubsystem(resolvedSystemName, subSystemToMutate, '_mAIB');
pathToMutatedSubsystem = strcat(newModelName, '/Subsystem');
mutAddBlockInbetween(newModelName, pathToMutatedSubsystem);
addMutantXML(XMLdocument, clones, pathToMutatedSubsystem);
close_system(newModelName);

% execute mutation mDIB
newModelName = createNewModelWithSubsystem(resolvedSystemName, subSystemToMutate, '_mDIB');
pathToMutatedSubsystem = strcat(newModelName, '/Subsystem');
if (mutDeleteBlockInbetween(newModelName,resolvedSystemName,pathToMutatedSubsystem))
    addMutantXML(XMLdocument,clones,pathToMutatedSubsystem);
end
close_system(newModelName);
% execute mutation mCS
newModelName = createNewModelWithSubsystem(resolvedSystemName, subsystemToMutate,'_mCS');
pathToMutatedSubsystem = strcat(newModelName,'/Subsystem');
mutCreateSubsystem(newModelName, pathToMutatedSubsystem);
addMutantXML(XMLdocument,clones,pathToMutatedSubsystem);
close_system(newModelName);
% execute mutation mABS
newModelName = createNewModelWithSubsystem(resolvedSystemName, subsystemToMutate,'_mABS');
pathToMutatedSubsystem = strcat(newModelName,'/Subsystem');
mutAddBlockAsSource(newModelName, pathToMutatedSubsystem);
addMutantXML(XMLdocument,clones,pathToMutatedSubsystem);
close_system(newModelName);
% run the TXL mRUE mutation (only works in Unix)
newModelName = createNewModelWithSubsystem(resolvedSystemName, subsystemToMutate,'_mRUE');
pathToMutatedSubsystem = strcat(newModelName,'/Subsystem');
newModelNameFilePath = strcat(resolvedSystemName,'/',newModelName,'.mdl');
mutationCommand = strcat({'txl␣','mRUE-mdl.txl␣' }, pathToMutatedSubsystem);
[status,cmdout] = system(mutationCommand{1});
if (status == 0)
    addMutantXML(XMLdocument,clones,pathToMutatedSubsystem);
else
    delete(newModelNameFilePath);
end
close_system(newModelName);
% write our XML mutant meta data file.
xmlwrite(strcat('mutant_',resolvedSystemName,'.xml'),clones);
end
catch exception
    bdclose('all');
    rethrow(exception);
end
cd('..');
bdclose('all');
simulink('close');
end

Our highest level Matlab call involves specifying a model file and the number of randomly selected systems to mutate. It is presented in Listing A.3.
A.2. ADDITIONAL MUTATION EXECUTABLES AND HELPER FUNCTIONS

Listing A.3: Function to Mutate Systems for a Specified Model

```matlab
function f = mutateModel( modelToMutate, numberOfSystemsToMutate )

% This is the high level mutate function for the SMCD Evaluator framework.
% It extracts the specified system and makes
% a copy of it as its own model file for each mutation that will be made.
try
    blockDiagram = modelToMutate;
    indexOfSlash = strfind(modelToMutate, '/');
    if numel(indexOfSlash) ~= 0
        blockDiagram = strtok(modelToMutate, '/');
    end
    load_system(blockDiagram);
    if (strcmp(get_param(modelToMutate,'type'), 'block_diagram'))==1
% If the selected subsystem is high(est) level block diagram
        mkdir(modelToMutate);
        subSystems = find_system(modelToMutate,'type', 'Block', 'BlockType', 'SubSystem');
        if numel(subSystems) == 0
            mutateSystem(modelToMutate);
        else
            for subSystemIterator = 1:numberOfSystemsToMutate
                stringOfSubsystemToMutate = subSystems(randi(numel(subSystems )));
                mutateSystem(stringOfSubsystemToMutate{1});
            end
        end
    else
        disp('The model name provided does not resolve to a Simulink model file');
    end
catch exception
    cd('..');
    bdclose('all');
    rethrow(exception);
end
bdclose('all');
simulink('close');
end
```

This listing, Listing A.4 creates an XML element that represents the metadata of an injected mutant. It creates the outer mutant element and calls the helper function to add the respective blocks in that element.

Listing A.4: Function to add XML for a Created Mutant

```matlab
function f = addMutantXML(XMLDocument,cloneContainer,pathToSystem )
    mutant = XMLDocument.createElement('mutant');
```
A.2. ADDITIONAL MUTATION EXECUTABLES AND HELPER FUNCTIONS

The helper function in Listing A.5 is called to copy the contents of an indicated subsystem and extract it to its own model file. It then ensure that the contents are once again placed in a subsystem as they were in their original location.

Listing A.5: Function to Create a New Model to Mutate

```matlab
function newModelName = createNewModelWithSubsystem( resolvedSystemName, fullPathToSubSystem, newSuffix)
    newModelName = strcat(resolvedSystemName,newSuffix);
    newModelFileName = strcat(resolvedSystemName,'/',newModelName,'.mdl');
    newbd = new_system;
    load_system(newbd);
    Simulink.SubSystem.copyContentsToBlockDiagram(fullPathToSubSystem, newbd);
    blocks = find_system(newbd, 'SearchDepth', 1);
    bh = [];
    for i = 2:length(blocks)
        bh = [bh blocks(i)];
    end
    Simulink.BlockDiagram.createSubSystem(bh);
    save_system(newbd,newModelFileName);
    mi = Simulink.SimulationData.ModelLoggingInfo.createFromModel(newModelName);
    mi.overrideMode_ = 0;
    set_param(newbd, 'DataLoggingOverride', mi);
    save_system(newbd,newModelFileName);
end
```

Listing A.6 was a useful helper function that adds XML block elements that correspond to all the blocks in a provided subsystem. It makes sure to avoid duplication and that it properly records subsystem blocks.

Listing A.6: Function to Add Blocks from a System to an XML document

```matlab
function f = addSystemBlocksToXMLContainer(xmlDocument, xmlContainer,systemName)
    % Works on an unopened system indicated by the systemName variable.
    load_system(systemName);
    % find the names of all blocks within the specified system
    blocks = find_system(systemName,'type', 'Block');
    if numel(blocks) > 0
```
system = blocks(1);
fullSubSystemName = strrep(getfullname(char(system)),sprintf('\n'),'\n');
xmlContainer.setAttribute('subsystem',fullSubSystemName);
end
for currentBlockIterator = 2:numel(blocks)
currentBlock = blocks(currentBlockIterator);
if strcmp(get_param(char(currentBlock),'BlockType'),'SubSystem')==0
% if its not a subsystem block itself
newBlockNode = xmlDocument.createElement('block');
fullName = getfullname(char(currentBlock));
fullName = strrep(fullName,sprintf('\n'),'\n');
newBlockNode.setAttribute('path',fullName);
xmContainer.appendChild(newBlockNode);
end
end
Appendix B

Clone Report Transformation Code Employed in Prototype

This transformation takes in a Simone report and generates a report that is consistent with the desired format for our prototype\(^1\). It involves acquire blocks and their full paths instead of the raw Simulink source. It is described in detail in Chapter 7.

Listing B.1: TXL Code for Transforming Simone Reports

```txl
% Converts a raw Simone (NiCad) report with sources into
% a Simone report containing lists of full-path blocks
% in place of the sources.
%
% Input files are typically named ...-withsources.xml

include "simulink.grm"
include "nicad-report.grm"
include "stringhelper.rul"

redefine program
  [cloneTag]
  | ...
end redefine

define sourceContents
  [system_list]
  | [repeat block_or_line_list]
end define
```

\(^1\)Transformation written mostly by Andrew Stevenson
define block_or_line_list
   [block_list] | [line_list]
end define

redefine system_list
   [attr srclinenumber] ...
end redefine

%added for the blocklist transformation - Matthew Stephan

redefine blocktype_value
   [id] | [stringlit]
end redefine

%endAdded

function main
   replace [program]
      P [program]
   construct MdlFiles [repeat model_map]
      % initial empty map
   export MdlFiles
      by
         P [cacheMdlParses]
            [clonesToBlocklist]
end function

define model_map
   [stringlit] [program]
end define

rule cacheMdlParses
   replace $ [attribute]
      file= SourceFile [stringlit]
   import MdlFiles [repeat model_map]
   deconstruct not * [model_map] MdlFiles
      SourceFile _ [program]
   construct OptModel [opt program]
      _ [message SourceFile] [read SourceFile]
   deconstruct OptModel
      Model [program]
   deconstruct * [default_single_element] Model
      Name TopLevelSystemNameSegments [repeat stringlit]
   construct TopLevelSystemName [stringlit]
      _ [+ each TopLevelSystemNameSegments]
   export MdlFiles
      SourceFile Model [fullyQualifyBlockNames TopLevelSystemName]
         % 
            [fullyQualifyLines TopLevelSystemName]
         % 
            [fullyQualifySystems TopLevelSystemName]
   MdlFiles
by
    file = SourceFile
end rule

rule clonesToBlocklist
    replace [sourceTag]
        <source file = SourceFile [stringlit] startline = StartLine [stringlit]
        Attrs [repeat attribute+] />
        LN [srclinenumber]
        System {
            _ [repeat system_element]
        }
    </source>
    construct StartLineNum [srclinenumber] % convert stringlit to number
    LN [unquote StartLine] % convert stringlit to number
import MdIFiles [repeat model_map]
deconstruct * [model_map] MdIFiles
    SourceFile Model [program]
deconstruct * [system_list] Model
    StartLineNum
    System {
        Name CloneSystemName [repeat stringlit]
        Elements [repeat system_element]
    }
    construct SubsystemClone [sourceContents]
    % StartLineNum
    System {
        Name CloneSystemName
        Elements
    }
deconstruct * [block_list] Elements
    Block {
        BlockType _ [blocktype_value]
        Name FullyQualifiedBlock [stringlit]
        _ [repeat default_element]
    }
    construct ConcatanatedCloneSystemName [stringlit]
        _ [+ each CloneSystemName]
    construct BlockNameSegments [repeat stringlit]
        _ [split FullyQualifiedBlock "|" [trimTail
            ConcatanatedCloneSystemName]
    construct FullyQualifiedSystemPath [stringlit]
        _ [join BlockNameSegments "|" [escapeXMLCharacters]
    construct Blocks [sourceContents]
    SubsystemClone [extractBlockNames]
    construct Lines [sourceContents]
    SubsystemClone [extractLines]
deconstruct Blocks
    BlocksAsXML [repeat block_or_line_list]
deconstruct Lines
    LinesAsXML [repeat block_or_line_list]
<source file=SourceFile startline=StartLine subsystem=FullyQualifiedSystemPath Attrs '>
BlocksAsXML [. LinesAsXML]
</source>
end rule

%Formats a string to be XML acceptable by doing escape character
%author mstephan
function escapeXMLCharacters
    replace [stringlit]
    ReturnString [stringlit]
    construct FormattedString [stringlit]
    ReturnString [substringReplace "&" "&amp;";] [substringReplace "<" "&lt;";] [substringReplace ">" "&gt;";] [substringReplace "\" "&quot;";] [substringReplace ";" "&apos;";]
    by
    FormattedString
end function

rule fullyQualifyBlockNames ParentName [stringlit]
    replace $ [block_list]
    Block {
        BlockType [blocktype_element]
        Name BlockNameSegments [repeat stringlit]
        More [repeat default_element]
    }
    _ [+ each BlockNameSegments]
    where not
    BlockName [grep "|"] % block name cannot already contain a separator
    % guards against rule recursion
    construct NewName [stringlit]
    ParentName [+ "|"] [+ BlockName]
    by
    Block {
        BlockType
        Name NewName
        More [fullyQualifyBlockNames NewName]
        [fullyQualifyLines NewName]
    }
end rule

rule fullyQualifyLines ParentName [stringlit]
    skipping [block_list]
    replace $ [default_single_element]
    ElementName [id] BlockName [stringlit]
    where
    ElementName [=} 'SrcBlock'][=} 'DstBlock]
    by
ElementName ParentName [+ "|"] [+ BlockName]
end rule

rule fullyQualifySystems ParentName [stringlit]
skipping [system_list]
replace $ [system_list]
   LN [srclinenumber]
   System {
      Name SubsystemName [stringlit]
      Rest [repeat system_element]
   }
construct NewName [stringlit]
   ParentName [+ "|"] [+ SubsystemName]
by
   LN
   System {
      Name NewName
      Rest [fullyQualifySystems NewName]
   }
end rule

function extractBlockNames
   replace [sourceContents]
      Subsystem [system_list]
   construct Blocks [repeat block_list]
      _ [^ Subsystem] [removeSubsystemBlocks] [blockToXML]
   construct Result [repeat block_or_line_list]
      _ [reparse Blocks]
   by
      Result
end function

function extractLines
   replace [sourceContents]
      Subsystem [system_list]
   construct Lines [repeat line_list]
      _ [^ Subsystem] [lineToXML]
   construct Result [repeat block_or_line_list]
      _ [reparse Lines]
   by
      Result
end function

redefine line_list
   ...
| [leafTag]
end redefine

rule lineToXML
   replace [line_list]
   Line {

    LineElements [repeat default_element]
    }
deconstruct * [default_element] LineElements
    'SrcBlock SrcBlock [repeat stringlit]
deconstruct * [default_element] LineElements
    'SrcPort SrcPort [default_value]
deconstruct * [default_element] LineElements
    'DstBlock DstBlock [repeat stringlit]
deconstruct * [default_element] LineElements
    'DstPort DstPort [default_value]
    construct SrcBlockWithElementsJoined [stringlit]
    _ [+ each SrcBlock]
    construct DstBlockWithElementsJoined [stringlit]
    _ [+ each DstBlock]
    construct NullString [stringlit]
    ""
    by
    <line srcblock= SrcBlockWithElementsJoined [escapeXMLCharacters]
    srcport= NullString [quote SrcPort]
    dstblock= DstBlockWithElementsJoined [escapeXMLCharacters]
    dstport= NullString [quote DstPort] />
end rule

redefine block_list
...
| [leafTag]
%| [SPOFF] <block [SPON] path [SPOFF] = [stringlit] /> [SPON][NL]
end redefine

rule blockToXML
    replace [block_list]
    Block {
        BlockType BlockTypeName [blocktype_value]
        Name BlockNameSegments [repeat stringlit]
        Rest [repeat default_element]
    }
    construct OptPortElement [repeat default_element]
    Rest [getPorts]
    construct BlockName [stringlit]
    _ [+ each BlockNameSegments]
    deconstruct not BlockTypeName
    'SubSystem
    construct NullString [stringlit]
    ""
    construct QuotedPort [stringlit]
    NullString [quote OptPortElement]
    by
    <block path= BlockName [escapeXMLCharacters] type= NullString [quote
    BlockTypeName] [escapeXMLCharacters]
    ports= QuotedPort [substringReplace "Ports " ""] />
end rule
This relatively simple shell script is used to transform a ConQAT report into one that is compatible with our prototype. The specific details are described in the thesis in Chapter 7.

Listing B.2: Shell Script for Transforming ConQAT Results

```bash
#!/bin/bash
sed '/<key-value-pair/d' $1 | \
    sed '/<?xml version=/d' | \
    sed '/finding-report/d' | \
    perl -pe 's/<finding-category name="Simulink Clones">/<clones>/' | \
    perl -pe 's/<finding-group description/<class id/' | \
    perl -pe 's/<finding origin-tool="ConQAT Model Clone Detection">/<source>/' | \
    perl -pe 's/<qualified-name name="(.+?)" uniform-path=".+?"/block path="/1"/' | \
    perl -pe 's|</finding>|</source>|' | \
    perl -pe 's|</finding-group>|</class>|' | \
    perl -pe 's|</finding-category>|</clones>|'
```