

## ABSTRACT

Title of dissertation: A BINARY CLASSIFIER FOR TEST CASE  
FEASIBILITY APPLIED TO AUTOMATICALLY  
GENERATED TESTS OF EVENT-DRIVEN  
SOFTWARE

Bryan Robbins, Doctor of Philosophy, 2016

Dissertation directed by: Professor Atif Memon  
Department of Computer Science

Modern software application testing, such as the testing of software driven by graphical user interfaces (GUIs) or leveraging event-driven architectures in general, requires paying careful attention to context. Model-based testing (MBT) approaches first acquire a model of an application, then use the model to construct test cases covering relevant contexts. A major shortcoming of state-of-the-art automated model-based testing is that many test cases proposed by the model are not actually executable. These *infeasible* test cases threaten the integrity of the entire model-based suite, and any coverage of contexts the suite aims to provide.

In this research, I develop and evaluate a novel approach for classifying the feasibility of test cases. I identify a set of pertinent features for the classifier, and develop novel methods for extracting these features from the outputs of MBT tools. I use a supervised logistic regression approach to obtain a model of test case feasibility from a randomly selected training suite of test cases. I evaluate this approach with a set of experiments.

The outcomes of this investigation are as follows: I confirm that infeasibility is prevalent in MBT, even for test suites designed to cover a relatively small number of unique contexts. I confirm that the frequency of infeasibility varies widely across appli-

cations. I develop and train a binary classifier for feasibility with average overall error, false positive, and false negative rates under 5%. I find that unique event IDs are key features of the feasibility classifier, while model-specific event types are not. I construct three types of features from the event IDs associated with test cases, and evaluate the relative effectiveness of each within the classifier.

To support this study, I also develop a number of tools and infrastructure components for scalable execution of automated jobs, which use state-of-the-art container and continuous integration technologies to enable parallel test execution and the persistence of all experimental artifacts.

A BINARY CLASSIFIER FOR TEST CASE FEASIBILITY  
APPLIED TO AUTOMATICALLY GENERATED TESTS OF EVENT-DRIVEN  
SOFTWARE

by

Bryan Thomas Robbins, III

Dissertation submitted to the Faculty of the Graduate School of the  
University of Maryland, College Park in partial fulfillment  
of the requirements for the degree of  
Doctor of Philosophy  
2016

Advisory Committee:  
Professor Atif Memon, Chair/Advisor  
Professor Hal Daumé  
Professor Jordan Goodman  
Professor James Reggia  
Professor Alan Sussman

© Copyright by  
Bryan Thomas Robbins, III  
2016

## Dedication

To my very patient wife, Christine.

## Acknowledgments

I thank my advisor Atif Memon for his guidance and support throughout my degree. I am very grateful to have been afforded the freedom to pursue a number of my own ideas, and that process (though a long one) has shaped me into a much better researcher and practitioner of software engineering.

I also thank my committee members: Drs. Daumé and Sussman, for their straightforward feedback and support during my proposal in 2013; and Drs. Reggia and Goodman, for their willingness to serve on my final dissertation committee.

I also thank my colleagues at the Financial Industry Regulatory Authority (FINRA) for their support and understanding as I continued to pursue my degree while working full-time. Daniel Koo, Nil Weerasinghe, Mohamed Ibrahim, Mike Dillon, Jacob Sheppard, and Stephen Mele have all been particularly supportive throughout my various sleepless rants.

I also thank UMD CS Alumnus Dr. Jeffrey Carver for his encouragement to pursue a Ph.D. at the University of Maryland, and his recommendation to the Department that I be considered as a student.

As of this writing, I have been “in school” continuously for 28 years. I would not have been able to do this research without the support of my parents, my family, my church, countless friends, teachers, and professors along the way. I thank God for all of you, and for everything that has led to this opportunity.

## Table of Contents

List of Tables	vii
List of Figures	viii
1 Introduction	1
1.1 Finding Bugs in Software	1
1.1.1 Software Defects	1
1.1.2 Software Quality Assurance Activities	2
1.1.3 Software Testing Activities	4
1.1.4 Testing Event-Driven Software	5
1.2 Automated GUI Testing with Models	9
1.2.1 GUI Testing	10
1.2.2 Existing Model-based Approaches which Incorporate Context	11
1.3 Test Case Feasibility	12
1.3.1 Defining Feasibility	13
1.3.2 Problems Caused by Infeasibility	13
1.3.3 The Need to Model Feasibility	15
1.4 Thesis Statement	15
1.5 Results	17
1.5.1 Intellectual Merit	18
1.5.2 Broader Impacts	19
2 Background and Related Work	21
2.1 Model-Based Testing	21
2.1.1 State-Based and Data-Flow Models	22
2.1.2 Event-flow Models	23
2.2 GUI Testing	25
2.2.1 Tool Support	26
2.2.2 Test Oracles	28
2.3 Related Work on Test Case Infeasibility	28
2.3.1 Summary of Related Work	32

3	Modeling Feasibility	33
3.1	The GUITAR Standard Workflow	33
3.1.1	Choice of GUI Platform	36
3.1.2	The GUI Ripper	36
3.1.2.1	Ripper Algorithm	37
3.1.2.2	Configuration Options and Customizations	38
3.1.2.3	Event Properties	40
3.1.3	Conversion	41
3.1.4	Test Case Generation	43
3.1.5	Test Case Replay	45
3.2	Adding a Binary Classifier for Feasibility	46
3.2.1	Supervised Learning and Binary Classification	47
3.2.2	Generalized Linear Model	47
3.2.3	Adding Regularization with Lasso	49
3.2.4	Feature Selection	51
4	Implementation Details	54
4.1	Portable Configurations with Docker	56
4.1.1	Data Volumes	58
4.1.2	Docker Image Details	59
4.1.2.1	Job Execution with Jenkins CI	59
4.1.2.2	Test and Experiment Artifact Persistence with MongoDB	61
4.1.2.3	Code Library Artifact Persistence with Maven and Sonatype Nexus	62
4.1.2.4	GUITAR Slave	62
4.1.2.5	R Slave	63
4.2	Persistence with TestData and MongoDB	64
4.2.1	ArtifactProcessors	65
4.2.2	Additional Processing	66
4.3	Execution of Automated Jobs in Parallel with Jenkins CI	67
4.3.1	Replaying Test Cases in Parallel	69
4.4	Custom R Scripts	70
4.4.1	Prepare Data	70
4.4.2	Train and Predict	73
5	Experiments	75
5.1	AUTs	75
5.2	Methodology	77
5.2.1	Overview	77
5.2.2	Training and Test Suite Construction	77
5.2.3	Training and Prediction	79
5.3	Planned Analysis	80
5.3.1	Overview	80
5.3.2	Sanity Checks	82
5.3.3	RQ1: Prevalence	83
5.3.4	RQ2: Binary Classifier	84
5.3.5	RQ3: Important Features	85
5.4	Results and Analysis	87

5.4.1	Sanity Checks . . . . .	87
5.4.1.1	EFG Complexity . . . . .	87
5.4.1.2	Training Data Complexity . . . . .	89
5.4.1.3	Inconsistent Test Cases . . . . .	91
5.4.2	RQ1: Prevalence . . . . .	92
5.4.3	RQ2: Binary Classifier . . . . .	93
5.4.3.1	Model Fit . . . . .	93
5.4.3.2	Classifier Performance . . . . .	96
5.4.4	RQ3: Important Features . . . . .	97
5.4.4.1	Event Types vs. Event IDs . . . . .	98
5.4.4.2	Most Frequently Selected and Highest Cumulative Magnitude Feature Type . . . . .	99
5.5	Published Artifacts . . . . .	101
6	Conclusions and Future Work . . . . .	103
6.1	Outcomes . . . . .	103
6.1.1	RQ1: Prevalence . . . . .	103
6.1.2	RQ2: Classification . . . . .	104
6.1.3	RQ3: Classifier Features . . . . .	104
6.1.4	Additional Outcomes . . . . .	105
6.2	Threats to Validity . . . . .	106
6.2.1	Threats to Internal Validity . . . . .	106
6.2.2	Threats to External Validity . . . . .	107
6.3	Future Work . . . . .	108
6.4	Conclusions . . . . .	110
A	Docker Infrastructure Scripts . . . . .	112
B	Custom R Scripts . . . . .	114
	Bibliography . . . . .	120

## List of Tables

1.1	Considerations of Context in Testing of Event-Driven Applications . . . . .	7
1.2	Research Questions and Hypotheses . . . . .	16
2.1	Comparison of Testing Frameworks . . . . .	27
3.1	Candidate Features for Feasibility Model . . . . .	52
4.1	Docker images . . . . .	58
4.2	TestData ArtifactProcessor Implementations . . . . .	65
4.3	ArtifactProcessor Interface Methods . . . . .	66
4.4	Automated Jobs . . . . .	68
5.1	Experiment AUTs . . . . .	76
5.2	Experiment Steps . . . . .	77
5.3	Sanity Checking Metrics Summary . . . . .	80
5.4	Research Question Metrics Summary . . . . .	81
5.5	RQ3 Feature Categories and Regular Expressions for Analysis . . . . .	86
5.6	Sanity Check Metrics (all but SANI) . . . . .	88
5.7	RQ1 Metrics . . . . .	92
5.8	RQ2 Metrics (all but RQ2A) . . . . .	93

## List of Figures

1.1	A basic event-driven software application . . . . .	6
1.2	Comparison of Event-based and Classic Test Cases . . . . .	7
1.3	A simple test case on the JabRef application . . . . .	9
2.1	GUI and corresponding EFG for a simple application. . . . .	24
2.2	Model-based GUI testing techniques enabled by GUITAR . . . . .	26
3.1	GUITAR Standard Workflow for model-based testing, with Artifacts . . . . .	34
3.2	GUIStructure and EFG of a simple application . . . . .	35
3.3	Pseudocode for the Ripper . . . . .	37
3.4	Pseudocode for computing <i>may-follow</i> relationships from an event . . . . .	42
3.5	Pseudocode for sequence-length test case generation . . . . .	44
4.1	Infrastructure Components and Dependencies . . . . .	55
4.2	TestData Persistence Model . . . . .	64
5.1	AUT Feature Frequencies . . . . .	89
5.2	RQ2A: Training Misclassification as Function of Lambda and Number of Variables . . . . .	94
5.3	RQ3A: Coefficient Counts by AUT and Variable Category . . . . .	98
5.4	RQ3B: Cumulative Magnitudes by AUT and Variable Category . . . . .	99

## Listings

4.1	Parallel GUITAR Replay - replay-suite . . . . .	70
4.2	Converting data.frame to model.matrix . . . . .	72
4.3	Training and Prediction with Logistic Regression and Lasso . . . . .	73
A.1	Jenkins GUITAR Slave . . . . .	112
A.2	Jenkins R Slave . . . . .	113
B.1	R Script: Common Utilities . . . . .	114
B.2	R Script: Prepare Data . . . . .	115
B.3	R Script: Predict . . . . .	118

## Chapter 1: Introduction

### 1.1 Finding Bugs in Software

Software “bugs” are notoriously difficult to find. Even companies spending billions on their software still find themselves humbled by problems after release. Some now argue that writing bug-free software is practically impossible (or at best, not cost effective). The high-stakes economy around technology also presents its own tradeoffs, as the cost of being late to market is often much higher than the cost of fixing bugs.

But what do we even mean by the colloquial term *bug*? At a high level, the real problem is that the software did not behave as expected. In practice, there are many people who have expectations about the behavior of any given software application: the end-users, the owners, the maintainers, and so on. Before we go on, we need a much more general (yet more precise) definition.

#### 1.1.1 Software Defects

IEEE Standard 1044 [1] provides a definition for a “defect” as “an imperfection or deficiency in a work product where that work product does not meet its requirements or specifications.” Importantly, defects can occur throughout the Software Development Lifecycle (SDLC). The artifacts of so-called *early-lifecycle phases* of the SDLC, such as requirements gathering and design, can have defects. Defects in these artifacts can carry

through to downstream artifacts such as source code and binaries. Defects which occur in the running, executable components of a software application are often classified as failures (departures from the expected behavior of the application) or faults (any manifestation of a human error in a piece of software).

For the purposes of this research, I will not typically distinguish between the different types of defects in software applications. I do, however, assert that the *quality* of artifacts is a function of the defects present. Furthermore, I assert that every defect detected and removed from a piece of software improves the quality of that software. While there are many tradeoffs to consider, finding and fixing defects as quickly as possible remains an important goal in modern-day software development.

Because the focus of this research is software testing, which is only possible on the executable artifacts of the SDLC, I will also refer to these artifacts as simply “software” or “software applications.” I will be specific when referring to any other artifacts of the SDLC.

### 1.1.2 Software Quality Assurance Activities

The discipline known as Software Quality Assurance (or simply “QA” within the software development industry) focuses on the development and application of techniques for measuring and assuring the quality of software. Regardless of the development and management methodologies being used on a software project, *QA activities* will always be required. At a high level, common QA activities carried out on software applications include:

- **Planning:** How should each artifact be checked for defects?

- **Finding Potential Defects:** Given the requirements or expectations of an artifact, are there any potential “imperfections or deficiencies” present?
- **Triaging Potential Defects:** Does each potential defect really qualify as a defect? If so, what is the likelihood and impact of the defect?

These activities can occur on any artifact in the SDLC. While planning and triaging are more intellectual in nature (e.g., weighing available information, options, and priorities), techniques for finding defects vary widely by artifact, and require technically complex tooling for late-lifecycle artifacts such as code and executable software. Defect-finding techniques also greatly impact the planning which is necessary for finding defects.

Given the focus of this study, here are some common techniques for finding defects in executable artifacts:

- **Compilation or Interpretation:** Source code translates into executable formats, and this process requires parsing and semantic analysis that may reveal defects related to conformance to the programming language.
- **Static Analysis:** Tools can inspect source code or compiled code for common mistakes.
- **Reviews:** Team members can read one another’s source code and offer feedback on possible defects.
- **Testing:** Executable artifacts can be evaluated by running and inspecting their output under certain conditions.

While the other techniques listed above are certainly valuable [2,3], *software testing* continues to be the most ubiquitous defect-finding technique in software development. Entire development methodologies in the industry (such as the very popular Test Driven Development technique [4]) have been developed around the idea of testing early and

often. I focus on software testing in this research.

### 1.1.3 Software Testing Activities

For projects which rely on testing as a primary way to find defects, QA activities are still required throughout the SDLC. A great deal of work related to testing a piece of software actually occurs prior to test case execution.

In *test suite design*, testers develop requirements for a test suite. We often refer to these test suite design requirements as *coverage criteria*. Testers commonly develop separate test suites which cover, for example, all requirements, all user interface (UI) components, all new features for a release, all web server endpoints, and so forth. Test suites designed in this way often overlap, but the separate suites can be executed in isolation to measure a particular aspect of quality.

Once a tester designs a test suite, he or she will *construct test cases* to be added to suites. The test cases for a specific suite should collectively achieve the coverage criteria of the test suite. Because of this requirement, each test case usually contributes to a specific coverage goal of the test suite. For example, if I design test suite A to accomplish a goal of verifying the functionality of all UI elements, each test case in A may be constructed to cover a single UI element.

Test cases take the form  $\langle I, E \rangle$ , where  $I$  describes some input to the application, and  $E$  any expected output to be verified against actual output during test case execution. These sets can both include multiple elements, corresponding to the number of inputs to be provided and expected outputs to be verified, respectively.

Apart from their construction, testers must also *maintain test cases and test suites* over time as an application changes. For example, a test suite with a goal of covering all

UI components of an application requires maintenance any time the UI components of the application change. Some maintenance requires the addition of new test cases, and some involves repair of existing test cases which may no longer be valid.

At a higher level, testers can *develop an overall testing strategy* which drives the design of test suites, creation of test cases, and maintenance of these artifacts. This higher-level view is critical for optimization and proper management of testing activities. For example, many test cases will be appropriate for more than one test suite, and should not be maintained separately in isolation. Even within a single test suite, individual test cases designed with targeted goals tend to overlap, wasting resources and maintenance effort. An effective strategy balances the important variables of testing effort and defect risk.

#### 1.1.4 Testing Event-Driven Software

Despite the complexity of software testing activities outlined above, I have only considered the simplified, general case. Today's most popular software applications, including those on Web, mobile, and Desktop platforms, leverage a common underlying model of event-driven interaction. Users (or in many cases, other software or hardware components) perform events. These events can produce output and change the runtime state of the application. This event-driven paradigm presents several challenges for many software testing activities.

Consider a standard event-driven application as depicted in Figure 1.1. An application defines an interface, and an actor performs events on this interface. The system receives the event, but this action may or may not affect the actual state of the application, and the system may or may not send a response to the user due to this action. Typically,

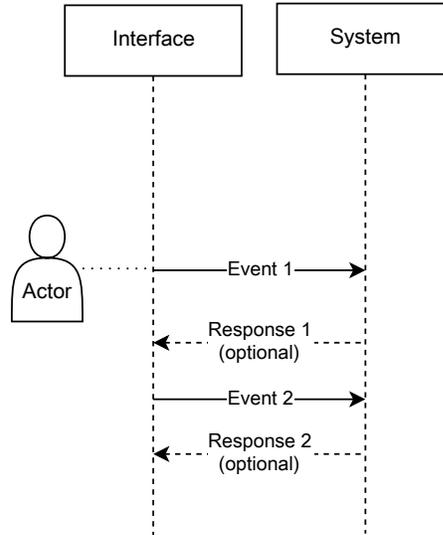
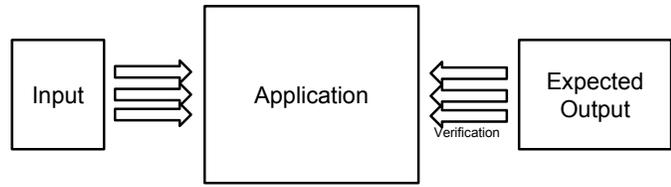


Figure 1.1: A basic event-driven software application

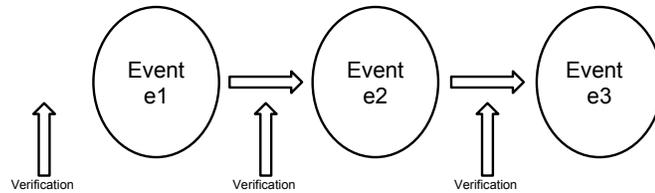
the actor may immediately perform additional events which follow the same pattern.

The ability of actors to perform events in sequence blurs the boundary between traditional notions of input and output in software testing, as the outputs of one event become the input of the next. The inputs specified for each test case take the form of *sequences of events*, with expected output specification allowed at any point during the event sequence. Figure 1.2 illustrates the contrast between this model and its pre-event predecessors. Similarly, output verification may occur at any point - even before or between the execution of input events.

In general, I assert that effective testing of event-driven software requires consideration of the software and any integrated systems' context throughout all testing activities. More formally, by *context* I mean the *state* of a running software component or integrated system. I assume that the behavior of any piece of software is a function of its compiled code and runtime state. To be sure, testers must consider this runtime state for effective testing of any software (even those which are not event-driven); but in the case of



(a) Classic test case



(b) Event-based test case

Figure 1.2: Comparison of Event-based and Classic Test Cases

event-driven software, there is typically much more context to consider.

Table 1.1 details some of the potential considerations of event context desirable for several testing activities throughout the SDLC. The need to consider context while carrying out each of these activities and others when testing event-driven applications directly compounds software testing’s fundamental problem of exponential growth.

Testing Activity	Considerations of Context
Test Suite Design	Sequence-based coverage criteria on various event domains Sampling according to event sequence characteristics
Test Case Construction	Control and evaluate contribution to sequence-based coverage criteria
Test Case Execution	Collect runtime event execution feedback on various event domains
Test Suite Maintenance	Minimize duplicate sequence coverage Compare sequence coverage between test suites
Test Case Maintenance	Evaluate contribution to coverage criteria

Table 1.1: Considerations of Context in Testing of Event-Driven Applications

Every necessary input event or verification added to a test case stands to increase the number of possible test cases required for effective testing exponentially. Considerations of context have analogous exponential effects throughout the SDLC.

In the example event-based test case from Figure 1.2(b), the verification performed after  $e2$  must take into account not only the effects of  $e2$ , but also any residual effects of  $e1$  and the interaction between  $e1$  and  $e2$ . Beyond that, the verification may even need to compensate for the existing state of an application prior to the execution of the current test case - for example, if the application is in any way sensitive to previously entered data. Often, testers can control for inter-test dependencies by applying techniques such as “testing the delta,” (i.e., designing the test and its verifications to ignore unrelated data). Clearly, testers must *consider the context* of event execution when constructing test cases for event-driven systems.

Working our way back up the hierarchy of testing activities, testers must also consider context during test suite design. Verifying the correct functionality of  $e2$  after the execution of the sequence  $e1$  does not directly imply correct functionality of  $e2$  after the execution of any other sequence. Therefore, a test suite designed to verify the correct functionality of  $e2$  may have to consider the execution of  $e2$  from multiple relevant contexts.

For example, consider the popular **Copy** → **Paste** operations available in many applications. If **Copy** has not been performed, then **Paste** should have no effect; but if **Copy** has been performed correctly, then we know that a **Paste** event should insert the copied text. Going back further, we know that **Copy** only has an effect after text selection, and most applications only support text selection after text has been inserted or loaded, and so on.

## 1.2 Automated GUI Testing with Models

Testers must become domain experts to appropriately incorporate context while performing testing activities. If left as a purely manual exercise, important effects of context can easily be overlooked. If the same knowledge could be obtained and modeled in an automated way, some of these problems could potentially be avoided.

Researchers have made a great deal of progress over the last decade in the area of automated model-based testing of applications which are driven by a graphical user interface (GUI). GUI-driven applications comprise a large subset of the broader group of event-driven applications considered so far. As with the larger class of applications,, system-level testing of GUI-driven applications involves the execution of input events on the graphical interface - an activity commonly referred to as *GUI testing*.

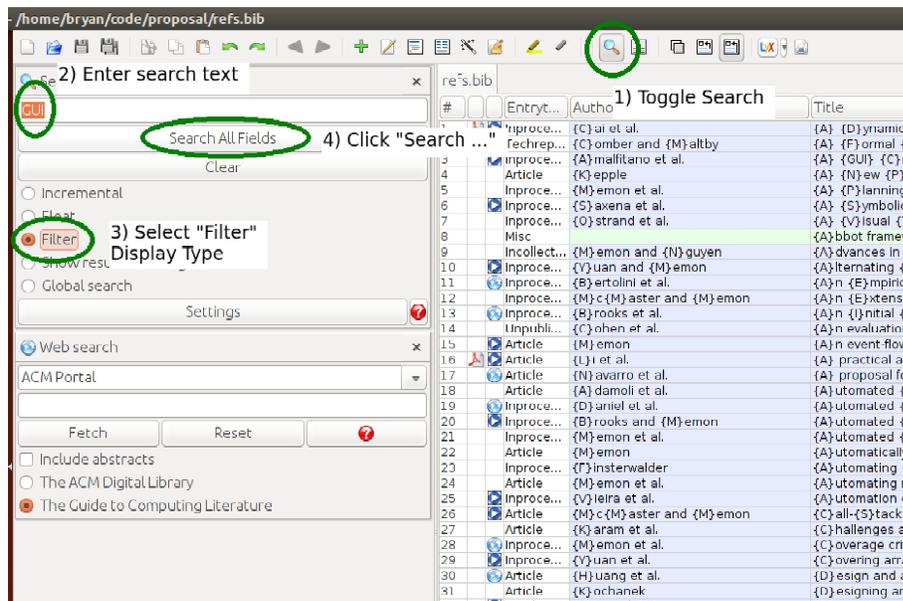


Figure 1.3: A simple test case on the JabRef application

### 1.2.1 GUI Testing

Figure 1.3 outlines an example input sequence on the JabRef Reference Manager, an open-source Java Desktop application (<http://jabref.sourceforge.net/>) used as a subject application in this study. This test case involves enabling the search panel (step 1), entering a search term (step 2), selecting a display option (step 3), and clicking to trigger the search (step 4). Each step acts as a single event in the input sequence. Notice several things about this input sequence which highlight specifics of GUI testing:

- Events are associated with and performed on GUI controls (also known as *widgets*).
- Widgets typically support only a very small set of actions. For example, only the text box in step 2 supports text entry.
- Events may or may not require accompanying input data (e.g., step 2).

Testers can incorporate any conceivable type of verification into a GUI test case. The use of a GUI by an application does not limit the available sources of output verification, though GUI-driven applications do introduce the GUI's state as an interesting subset of verifiable application state. Tests can perform perhaps the simplest possible output verification by checking only for crashes by the GUI or in the application's log output. If an application uses a back-end database, a test can verify database state at any point during execution, and similarly compare current values to expectations for any observable part of an application's state.

A verification's type (e.g., log, GUI, database, etc.) determines the information needed to completely specify a test case. At a minimum, specification of each input step requires information necessary to identify widgets and perform events (including input data, if applicable). A number of popular industry tools can automatically replay well-

specified test cases for GUI-driven applications in various domains (see Selenium WebDriver, Android Robotium, Quick Test Professional, and others). Each of these tools, however, depends on error-prone manual creation of test cases, and prior to that, manual test suite design and maintenance decisions. As an alternative, I previously described GUITAR [5], a multi-platform framework for automated model-based testing of GUI applications on various platforms. At its core, GUITAR reverse engineers event-based models of GUI applications, and traverses these models to generate suites of replayable test cases which meet tuneable event-based criteria.

I elaborate on additional considerations for GUI testing in Section 2.2. For this study, I use GUITAR’s tools for automated model-based testing.

### 1.2.2 Existing Model-based Approaches which Incorporate Context

The standard GUITAR workflow builds an *Event-flow* model of the application and generates a test suite which covers all event sequences of length 2 from the model. The Event-flow model incorporates knowledge of *may-follow* relationships between events, where A *may-follow* indicates that an event B may be executed after an event A. Test suite generation algorithms use information about *may-follow* relationships to construct test cases and satisfy coverage criteria.

Additional methods derived from the event-flow model have also incorporated additional context information. The Event-interaction Graph (EIG) model enhances the event-flow model with information collected at runtime about the effects of event execution on underlying, non-GUI application state [6–8].

The Probabilistic Event Flow Graph (PEFG) of Brooks and Memon augments the event-flow graph with probabilities according to event sequence collected from real users

[9]. On a different domain, McMaster and Memon consider sequences of method call stacks to inform a reduction technique [10]. The current research effort builds on these two approaches most directly in an attempt to consider event context collected from arbitrary domains.

Model-based approaches defined on the GUI domain appear to be much more scalable than approaches defined directly on lower levels of application state. State-based test generation attempts to capture the entire application state [11–13], which has not proven scalable for event-driven applications. Similarly, data-flow approaches require analysis of statements of code [14, 15]. Neither method directly informs the general consideration of context in arbitrary domains.

I elaborate on additional model-based approaches 2.1, and on the specific modeling approaches used in this study in Chapter 3.

### 1.3 Test Case Feasibility

Model-based testing approaches consider context by modeling the behavior of applications, and generating test cases which cover the contexts represented by the model systematically. However, model-based testing is not without its problems. One of the most significant problems with popular model-based techniques is test case feasibility, which I define here and aim to directly address in this research.

Infeasibility is not a new problem, and has been addressed by a number of existing empirical studies of model-based GUI testing in particular [16–19]. In this research, however, I am proposing to deal with this problem in a new way: through the use of a binary classifier using a linear model. I elaborate on the current study’s relationship with existing work in Section 2.3.

### 1.3.1 Defining Feasibility

The shortcomings of automated, model-based GUI testing techniques largely stem from imprecise model construction. In the case of GUITAR, a tool attempts to construct the model in an automated way. Because tests are derived from the model, an imprecise model will lead to tests which do not behave as expected when replayed. Due to model imprecision or other possible application-specific issues, model-based techniques often generate test cases which are *infeasible*.

I refer to an infeasible test case as one which cannot be completely executed as planned. Such test cases may provide some of their intended coverage and verifications, but not all.

### 1.3.2 Problems Caused by Infeasibility

Infeasible test cases introduce a number of problems for a model-based testing approach:

1. An infeasible test case provides less coverage than designed, causing the parent test suite to also provide less than its designed coverage.
2. Determining the root cause of infeasibility requires manual intervention and investigation into an otherwise fully automated process.
3. Infeasible test cases cause false positives in test results, which can be expensive to manually investigate.
4. Any cost associated with dealing with infeasible test cases will grow as the application changes.

Coverage guarantees are a primary motivation for the use of model-based testing

approaches in the first place. Through coverage, the same techniques offer appropriate consideration of context for event-driven software. The presence of infeasible test cases threatens the integrity of these techniques altogether. For example, a test case  $e_c \rightarrow e_p$  designed to cover **Paste** after **Copy** only truly covers **Paste** after **Copy** if both events execute successfully. If for any reason either event cannot be executed as planned, the test case is infeasible. The larger suite, if designed to cover each event within one or more prior events' context, no longer successfully provides this guarantee.

Because model-based testing approaches often deal with very large numbers of test cases (in the 100s of thousands), manual intervention in even a small percentage of executed test cases can take many hours. For example, consider a typical GUI-driven application, with 500 unique events in its GUI. Covering this application for all pairs of events requires up to 250,000 test cases in a test suite. Each infeasible test case could easily take several minutes, at best, to triage as infeasible or legitimately failed. If even 5% of the original test cases in the suite fail due to being infeasible, this would result in 12,500 false positives and several weeks to triage (86 24-hour days, if each false positive required 10 minutes of investigation). This time due to wasted effort, infeasible test cases threaten the viability and usefulness of any model-based testing approach.

As mentioned earlier in this section, there are additional QA activities outside of simply executing test cases, and each of these is likewise impacted due to the presence of any infeasible test cases. Any manual effort for dealing with feasibility concerns would be amplified within these activities as well. For example, with each new version of the application, new infeasible test cases could be introduced that must be triaged. For each test case rejected due to infeasibility, a new test case must be generated and executed in order to obtain desired coverage criteria.

### 1.3.3 The Need to Model Feasibility

While the feasibility of test cases can be directly observed through execution, full execution of proposed model-based test suites in order to observe and verify test case feasibility is not ideal due to the effort and computational resources required. Also, as mentioned above, determining infeasibility even after execution can also require some degree of manual intervention.

*A high-accuracy, pre-execution feasibility indicator* would be much more efficient for the detection of potentially infeasible test cases. In machine learning terms, this problem is a *binary classification problem*. Given a test case, I require a model which can classify the test case as feasible or infeasible with high accuracy. Such a model would greatly improve the overall effectiveness of model-based testing approaches as a whole, by avoiding the problems caused by infeasible test cases. Prior to test case execution, only the inputs and expected outputs of a GUI test case are available, and any such model would only be able to leverage the features of these artifacts.

## 1.4 Thesis Statement

In this research study, I consider the research questions presented in Table 1.2. For research question RQ1, I consider the prevalence of infeasibility. I hypothesize that the generation of infeasible test cases is indeed prevalent, at rates of up to 20%, in model-based testing workflows. To test this hypothesis, I generate and execute 44,000 total model-based test cases for four Java applications (11,000 per application). From test execution results, I develop a method to infer the feasibility of each test case from log output.

For research question RQ2, I consider the ability to develop an application-specific

ID	Question	Hypothesis
RQ1	How prevalent are infeasible test cases in MBT workflows?	Infeasible test cases will occur in all MBT test suites, at a rate of up to 20%.
RQ2	Can an application-specific model of test case feasibility be used to predict the feasibility of unseen model-based test cases?	A binary classifier can be constructed which will outperform at least a random (50%) baseline in predicting the feasibility of unseen test cases.
RQ3	Which of the considered feature types of model-based test cases contribute the most to feasibility prediction?	Hypothesis A: Test case “before-pairs” and n-grams of length=2 will contribute most often and most significantly to feasibility models.  Hypothesis B: Test case “before-pairs” will contribute more significantly to feasibility models than n-grams.

Table 1.2: Research Questions and Hypotheses

binary classifier for feasibility. I hypothesize that an effective binary classifier for test case feasibility (that is, one that at least outperforms a random baseline) can be constructed from a reasonable amount of test executions observed as training data. To test this hypothesis, I divide the 11,000 generated test cases per application into Training and Test suites of test cases. I develop a set of features for the binary classifier, and a tool to extract these features from each test case. As with RQ1, I infer the feasibility of each test case from the execution log. I train the classifiers from the features of test cases in the Training suite and the observed feasibility of each test case. I then use the classifier to predict the feasibility as observed from the output of each test case in the Test suite.

Finally, in research question RQ3 I consider which of the features chosen by the training process of the binary classifier contribute the most to its effectiveness. I hypothesize that features which incorporate context (in particular, n-grams and “before-pairs” of event IDs) will contribute the most to feasibility models. Further, I hypothesize that

the loosely ordered event ID before-pairs will contribute more often and more significantly than strictly ordered n-grams. I define each of these categories precisely in Section `sec:feature:select`. To test these hypotheses, I inspect the features selected by each binary classifier, and compare the relative counts and magnitudes of coefficients in the trained model.

The experiments outlined below require the development of a number of novel techniques, infrastructure components, and tool support. Prior to the carrying out experiments, I develop extensions for the model based testing workflow to include the training of the feasibility classifier in Chapter 3. I discuss the implementation of custom infrastructure components, tools, and scripts which support modeling and other activities required for experiments in Chapter 4. I elaborate on the steps of these experiments in full detail in Chapter 5.

## 1.5 Results

I carry out a number of planned analyses (described in Section 5.3) for each research question to arrive at the following outcomes, which are the primary contributions of this paper:

- O1: I confirm that infeasibility in model-based test suites is a significant problem, with infeasible test cases generated at rates of 7.8% or higher for every test suite I considered, and as high as 53.5%.
- O2: I confirm that the tendency to generate infeasible test cases varies widely across applications.
- O3: I develop and train a novel binary classifier of feasibility with overall error, false positive, and false negative rates under 5%, constructed by observing the executions

of randomly sampled model-based test cases.

- O4: I find that the event types of the the GUITAR Ripper’s `GUIStructure` output artifact do not correlate with infeasibility, confirming that initial model imprecision (rather than the inferences of the Converter and Test Case Generation algorithms) is the dominant source of infeasibility in model-based testing workflows.
- O5: I find that event IDs formed by creating hashes of GUI properties during model construction served as effective features for the binary classifiers constructed in this study.
- O6: I identify three types of features formed from event IDs that serve as effective inputs for the feasibility classifier: loosely ordered before-pairs of event IDs, and n-grams of event IDs of length=1 and length=2.
- O7: I find that all three of the feature types identified in O6 contributed to feasibility classifiers, no single type of the three appeared unilaterally more effective across the four AUTs considered in my experiments.

Section 5.4 explains the analysis which led to each of these findings, within the context of the research questions RQ1, RQ2, and RQ3 presented above.

### 1.5.1 Intellectual Merit

In this research, I demonstrate the application of machine learning to an important problem from the field of software testing. I assert that model-based testing needs new types of models to deal with its most fundamental challenges, and I develop and evaluate one such model in this work. Development of the model requires identification of features pertinent to feasibility, and the development of novel techniques and tooling for feature extraction.

### 1.5.2 Broader Impacts

As I practicing software engineer, I have seen very little practical application of model-based testing. In my observation, model-based testing is not inapplicable. Teams of dedicated QA Engineers spends months developing effective test automation. In my opinion, model-based testing in its current forms are often viewed as impractical because:

- MBT cannot really provide the very guarantees that it claims (such as coverage criteria and full automation).
- Model construction and verification are tedious.
- Model construction activities are seen as “reimplementing the application” rather than verifying its logic.

I believe that this research directly addresses the first bullet above, introducing a new class of techniques for dealing with model-based testing’s shortcomings. I show in this study the construction of feasibility models, but believe that similar models may be able to predict other important variables such as code coverage, HTTP call and database coverage, and other important variables in the future.

As a secondary goal, I also emphasize the scalability of model-based testing throughout this research. I believe that cloud computing in particular stands to revolutionize testing strategy and techniques through scalability. Just as test automation continues to replace manual testing, I believe that newfound scalability demands more thorough coverage criteria, such as those provided by model-based testing techniques. This research demonstrates the efficient use of hardware resources for model-based testing of GUI applications, through the use of state-of-the-art open-source tools for containerization and continuous integration.

I also emphasize the repeatability of all experiments and analysis performed as a part of this research. I persist every model-based testing and analysis artifact, as well as configurations for every tool used. I believe that my approach for achieving highly repeatable experiments is relevant for empirical studies of software engineering, software testing, and potentially studies from other fields as well.

## Chapter 2: Background and Related Work

In this chapter, I elaborate on background information and existing research relevant to this study. First, I review existing research in the two related fields of model-based testing and GUI testing. I then compare this research to a number of related research studies which consider test case feasibility.

### 2.1 Model-Based Testing

Modeling has become essential for modern software development. Developers now have models for many things: models of the Software Development Lifecycle (SDLC), such as Waterfall or Spiral; models which formalize a system's requirements; models of a system's objects or runtime components; and models for every phase of the SDLC. The goal of modeling systems is usually abstraction. For example, during software requirements gathering activities, stakeholders do not care about the implementation details of the system. They can abstract these details away by modeling only the user and the system as a whole. As development of any one feature moves through the SDLC, different models abstract away less relevant details from the task at hand. Often, multiple models - each emphasizing different software characteristics - are needed in order to effectively make decisions.

As I assert in Chapter 1, effective software testing depends on the ability to incorporate relevant context. Uniformly, all consideration of context requires the presence of

some type of “model” of the application. Even in cases where a formal abstraction is not defined, a domain expert’s own mental model is driving testing activities. In this study, I focus on the more formal class of models. Researchers have predominantly proposed three classes of models capable of considering context and drive testing activities, all of which can be applied to event-based applications: state-based, data flow, and event flow.

### 2.1.1 State-Based and Data-Flow Models

In state-based models of testing, testers construct state machine models which attempt to model an application in terms of its runtime states. Test case generation, then, attempts to satisfy various coverage criteria defined on the state machine. Though referred to by various names in the literature, coverage of transitions or *edges* in the model is a commonly used criteria [11–13]. Note that coverage of transitions requires the consideration of a single-length history of prior state. The event-driven domain of protocol testing was in many respects a catalyst for state-based testing approaches in the early 1990s [20, 21], though protocols considered at that time were much simpler domains than the event-driven protocols of today’s applications.

In practice, automated construction of state machine models of non-trivial applications is often infeasible, despite a number of enhancements which attempt to limit the number of states and transitions that must be captured by the model. Cheng and Krishnakumar describe the Extended Finite State Machine (EFSM), which limits states by adding additional counters and runtime components [22]. Similarly, Shehady and Siewiorek describe an extension of FSMs to use variables, which limits the number of states to be explored [23], and the work of White, et al. suggests subdividing models into usage-specific categories [24, 25].

In practice, these techniques provide some utility in well-defined or highly critical domains, but would likely be ineffective in the more complex domains of larger event-driven systems. In addition to the high cost of model construction at the often hidden and complex level of application states, the task of mapping back to a concrete, executable input domain remains an error-prone manual exercise.

In contrast to control-flow models, data-flow models of software track context by following traces of variables in a program. A model known as a *control flow graph* details the flow of a program through its executable statements. The primary indicators of context are the uses of variables. Variable reads and writes and their use in interesting combinations are analyzed. Rapps and Weyuker first introduced a set of adequacy criteria for data-flow testing [14], focusing primarily on coverage of their Def/Use program graph. Frankl and Weyuker later expanded on these original criteria, claiming that many programs could not satisfy the central criteria of the original work [15], for reasons as common as a simple `for` loop. In practice, once again, reliable construction of program graphs and generation of tests which satisfy these criteria does not scale to the complexities of modern applications.

### 2.1.2 Event-flow Models

Given the issues with state-based and data-flow approaches to capturing context, Memon proposed the event-flow model for GUI testing [26, 27]. In this work, the nodes in a graphical model of the GUI capture behavioral *may-follow* relationships. Given the event-flow model, coverage criteria can be defined which consider the coverage of the *may-follow* relationships present in a GUI [28]. Important coverage criteria include pairwise coverage (covering all edges in the event-flow model’s graph) and a more generic sequence

length coverage concept, which requires covering all sequences of events in the graph of a given length. (Note that pairwise coverage is a length-2 version of this criteria.)

Because test case construction considerations follow the construction of the event-flow model, the accuracy of the model determines the accuracy of any inferred and generated test cases. Given an accurate model, automatically generated test cases will be executable. Given an inaccurate model, infeasible test cases will be produced.

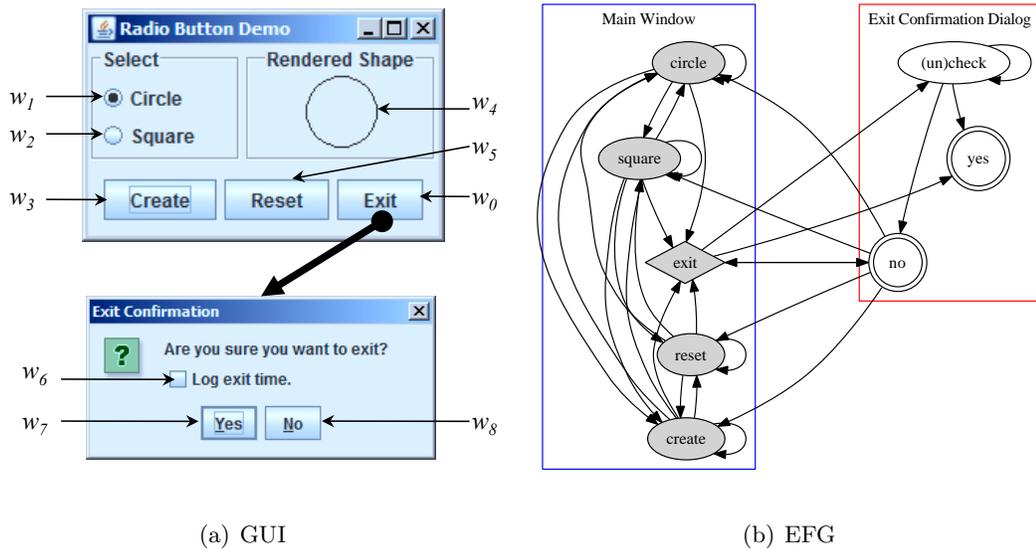


Figure 2.1: GUI and corresponding EFG for a simple application.

Figure 2.1 shows a GUI and the corresponding event-flow model for a simple Java application. Nodes in the graph represent events in the application’s GUI, and the presence of edges in the model indicate that an event can successfully follow another. While most buttons are always available from any other, the Yes button is only available after clicking “Exit,” because this button confirms the Exit operation. This relationship is evident in the event-flow model as well, as the corresponding edge between these nodes is missing.

Memon et al. later developed a reverse engineering tool known as the Ripper for constructing event-flow models in an automated way [29], giving this model some advantages

over the previously discussed models. Automated model construction makes construction of models of modern GUI applications possible, for the first time. More recent work with the event flow model has enhanced the model to focus on events more likely to find faults. To identify these events, automated testing workflows collect runtime information during test execution. Data collected during one execution can be re-used to improve the model used in the next, and this process repeated iteratively [6–8]. The work of Yuan et al. explicitly aimed to consider events in context [30] based on the application of covering arrays to an adaptation of the event-flow model called the event-interaction graph (EIG).

## 2.2 GUI Testing

Since the start of the 21st century, GUI testing as a subdiscipline of software testing research has continued to grow significantly alongside the rise in the popularity of GUI-driven applications. Researchers often focus on model-based testing of GUI-driven applications, though industrial trends have yet to follow. This disconnect was also noted in a recent systematic mapping of published literature on the topic of GUI testing by Banerjee et al. [31]. In many cases, however, large-scale research efforts enabled by automated, model-based testing have been able to inform testing practice.

One practical line of research which followed from the event-flow model was the Daily Automated Regression Testing (DART) framework [32], which was shown to be effective for identifying bugs which caused crashes in daily or nightly builds. The DART tools would later become the GUI Test Automation Framework (GUITAR), an open-source framework<sup>1</sup> for end-to-end automated testing on various GUI application platforms. Figure 2.2 shows the types of automated, model-based testing workflows enabled by GUITAR.

---

<sup>1</sup><http://guitar.sf.net>

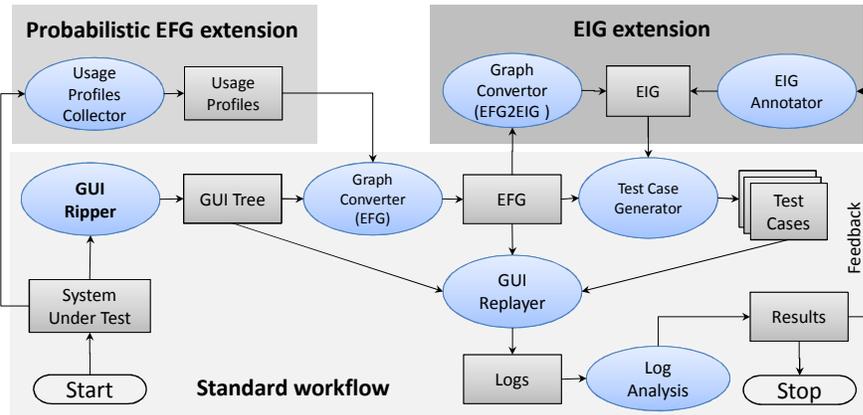


Figure 2.2: Model-based GUI testing techniques enabled by GUITAR

In 2013, Nguyen et al. published an updated description of GUITAR from an architectural perspective [5], carrying out a number of case studies which highlighted some of the pitfalls of the automated model-based approach. Most objections arose from the reverse engineering Ripper tool, due to difficulties with widget identification and the simplicity of its algorithm (which executes in a single application instance). The authors observed that the one-time execution of the Ripper led to invalid model construction due to the effects of event and application context. Incomplete or otherwise incorrect models lead to the generation of infeasible test cases. This *infeasibility problem* is exactly what I address in this study.

### 2.2.1 Tool Support

Table 2.1 offers a concise, high-level comparison of the features of GUITAR with the features of other frameworks popular in the industry (re-printed from [5]). The alternative tools include:

1. Monkey, a tool for random walking of Android application GUIs.

Framework Name	Model Generation	Test Case Generation	Supported Platforms
GUITAR	Rev. Eng (A)	Model based (A)	Multiple†
Monkey	None	None	Android
NModel	Scripted (M)	Model based (A) & Scripted (M)	C#
Quick Test Pro	None	Scripted (M) & Captured (M)	Multiple‡
Selenium	None	Scripted (M) & Captured (M)	Web

A = automated, M = manual  
†JFC, SWT, Web, Android, other platforms in alpha  
‡Java, Web, .NET

Table 2.1: Comparison of Testing Frameworks

2. NModel, a model-based testing framework for C# programs.
3. Quick Test Pro, a popular proprietary, multi-platform tool for test automation.
4. Selenium WebDriver, a popular API for browser automation.

While I am not particularly concerned with multiple-platform support for this study, a comparison of the approaches of each tool is informative. Tools from the industry (in particular, the very popular Selenium WebDriver and HP Quick Test Professional tools) tend to focus on giving a tester direct control over test case creation. Academic researchers have argued for many years that completely manual approaches do not scale well enough to meet advanced coverage criteria, and this intuition seems correct.

One difficulty common to GUI testing which all testing tools must address is the identification of widgets in a GUI. In the case of Selenium, an API in Java, Python, or other available languages allows a tester to programmatically specify an identifying

string for each widget in supported ways (e.g., XPath or CSS selector notation) . Quick Test Professional accomplishes identification with tool support, allowing testers to select a desired widget graphically while constructing a test. Like the GUITAR Ripper, both of these approaches can lead to high maintenance costs if widget identification does not hold across future versions of the software.

The Monkey tool is also an interesting use case. Testers in the mobile domain, where the set of available widgets and the scale of program state are limited, have found that randomly walking through applications and checking for crashes and specific, tool-supported constraint violations is often effective for finding faults.

### 2.2.2 Test Oracles

Another important characteristic of testing tools is test oracle support. By “test oracle,” I mean the method of verification for a test. One long-standing problem with GUI testing is the cost of maintaining a set of expected values to be verified during test execution. As far back as 1992, Richardson et al. studied the challenges of oracles for event-based systems (referring to these systems as *reactive systems*) [33, 34].

More recently, Memon et al. began to look at oracles for GUIs [35]. Xie and Memon carried out a large-scale empirical study with GUITAR that showed the utility of more advanced oracles over crash testing [36]. Examples of other, more effective oracles in that study include those defined on the GUI structure to be checked during test execution.

## 2.3 Related Work on Test Case Infeasibility

As discussed in Chapter 1, model-based considerations of context applied to the testing of event-driven systems often generate infeasible test cases. Existing research has

explored the ability to detect and fix model-based test cases which were either directly generated as infeasible or became infeasible across application versions.

In 2008, Memon proposed an EFG comparison technique for detecting test cases which become infeasible due to GUI changes across application versions [16]. Once detected, one of four repairing transformations described can be used to update the test case to work successfully with the new application version while maintaining relevant portions of its original event coverage.

In this research, I am concerned with generated test cases which are immediately infeasible rather than those which become infeasible across application versions. Also, Memon's technique here relies on the EFG model for detection and resolution of GUI changes. However, producing accurate EFGs by hand requires an intractable amount of manual effort. The EFGs produced by the fully automated Ripper algorithm are typically inaccurate due to limitations of the algorithm [5, 18, 19].

In general, approaches other than Memon's original work in 2008 have focused on the use of feedback loops (a concept first formalized for model-based testing by Yuan and Cohen in 2008 [37]) to inform models and test case generation strategies in particular about possible sources of infeasibility. My technique in this research is also based on feedback, as I use test execution results to construct the binary classifier.

Given the shortcomings of fully automated model construction, Huang, Cohen and Memon, in their 2010 research, consider the more general case of test case infeasibility [18]. Using a feedback loop, they developed a genetic algorithm for selecting feasible test cases. The algorithm starts with candidate test cases from a seed test suite, and mutates the original suite while preferring feasible test cases. The algorithm did show the ability to converge and produce test suites of feasible test cases; however, it required execution of

test cases in order to classify them as feasible or not. As a genetic algorithm, very large numbers of possibly feasible, alternative test cases were considered, and execution of these during the iterations of the algorithm took days or weeks.

In this work, I do use a seed test suite to construct a model. In my case, the model is a binary classifier rather than a covering array as in Huang, Cohen, and Memon’s work. I focus on the ability to extract features from test cases which indicate infeasibility rather than directly executing and observing test cases and only extracting feasibility as a binary value. My algorithms for training and prediction do depend on the ability to converge, and are subject to “garbage in, garbage out” just as a genetic algorithm would be. However, the larger feature space available to my technique should make my approach less susceptible to negative outcomes.

More recently, Nguyen and Memon proposed a similarly feedback-based approach for modeling and test case generation called the OME\* paradigm [17]. Under this paradigm, a starting EFG model is augmented with additional information after test case execution, leading to what the authors call the EFG+ model. The EFG+, armed with this additional information, can generate test cases which are guaranteed to be feasible.

In this work, I do not focus on feedback or model updating as in the OME\* approach. My technique can therefore be applied to a much broader class of applications and event-based testing domains (e.g., HTTP calls, database queries, and other event-based domains). My technique also does not depend on any particular model or source of test cases at all, which can be advantageous when incorporating domain knowledge that any one particular model (such as the EFG or EFG+) is not designed to capture. The OME\* approach also depends on observing an event within a specific context before performing model updating. By contrast, my technique can extrapolate possible causes of infeasibility

from a much broader class of test case characteristics than an EFG alone allows.

Finally, work by Gao et al. in 2015 addressed feasibility by abandoning a fully automated process for manual corrections. They developed a tool SITAR which suggests model and test case corrections to a manual user (presumably, a domain expert for the application). They also introduced a new annotation to the EFG model, *dominates*. The dominates relationship, when identified by a Ripper or manual feedback, indicates that an event must proceed another (in contrast to the *may* proceed implied by the existing annotation). In their case, they found the tool to be very effective at suggesting possible corrections for infeasible test cases.

I consider my current work to be in-line with the approach of Gao et al. in that I agree that additional information, whether readily presented in an EFG-like model or not, is necessary for dealing with infeasibility. The more generic model used in my research here would allow for a broader class of domain knowledge in general to be captured than an EFG alone allows. My technique requires that such domain knowledge be encoded as features to a binary classifier.

Also of note is the empirical study of Bae, et al. in 2014 [38]. Within a much broader comparison of GUITAR’s sequence-length coverage criterion and a dynamic event extraction testing approach, they confirmed that GUITAR’s model-based test cases began to have problems with infeasibility (what they refer to as “partially executable test cases”). They assert, as I have already in motivating the need for a feasibility classifier, that partially executable test cases lead to redundant coverage that still falls short of the desired coverage criteria. They note that redundancy, especially within the context of a more exhaustive model-based testing approach, leads to large amounts of wasted execution effort.

### 2.3.1 Summary of Related Work

In summary, I believe that my approach in this research largely expands on existing work which considers test case feasibility. While my technique may still be applicable across application versions, the direct problem I am addressing is the generation of infeasible test cases from a model-based technique, presumably due to model imprecision. My technique allows feasibility considerations to remain automated, and it is generalizable to any conceivable type of domain knowledge which can be extracted from or annotated onto test cases by any automated or manual process.

The idea of using an EFG or annotated EFG is synonymous in the machine learning world with using probabilistic N-gram models for feasibility prediction. Probabilistic models certainly could have additional use cases beyond binary classification; but if binary classification of some characteristic is the goal, use of a more general linear model is much more flexible. I take advantage of that flexibility to consider additional types of features in this work, and I assert that this flexibility leads to a much more effective approach for feasibility prediction in general.

## Chapter 3: Modeling Feasibility

In this chapter, I outline the models I use to predict the feasibility of unseen model-based test cases of event-driven software. I start with an overview of GUITAR’s model-based workflow, and append additional steps to that workflow for the prediction of feasibility.

### 3.1 The GUITAR Standard Workflow

As discussed, this research primarily seeks to improve on the techniques of the GUITAR framework for model-based GUI testing [5]. Numerous existing studies have leveraged GUITAR’s sequence-length test case generation workflow (which we referred to as the “standard workflow” in the 2013 paper), shown in Figure 3.1.

At a high level, a Ripper tool produces a GUI Tree output called a `GUIStructure`. The Converter then converts the `GUIStructure` into an Event-flow Graph (EFG). The Generator traverses the EFG to output many executable test cases. Finally, the Replayer executes each test case (one at a time). For this study, I use default algorithms and configurations for each tool, and a Random Sequence Length strategy for the Generator.

As Figure 3.1 indicates, each tool in this workflow outputs one or more XML-formatted documents, which in turn drive the next tool and step in the workflow. In the subsections below, I describe each tool. I use a running example of a toy application, shown in Figure 3.2 (also reprinted from our 2013 paper [5]).

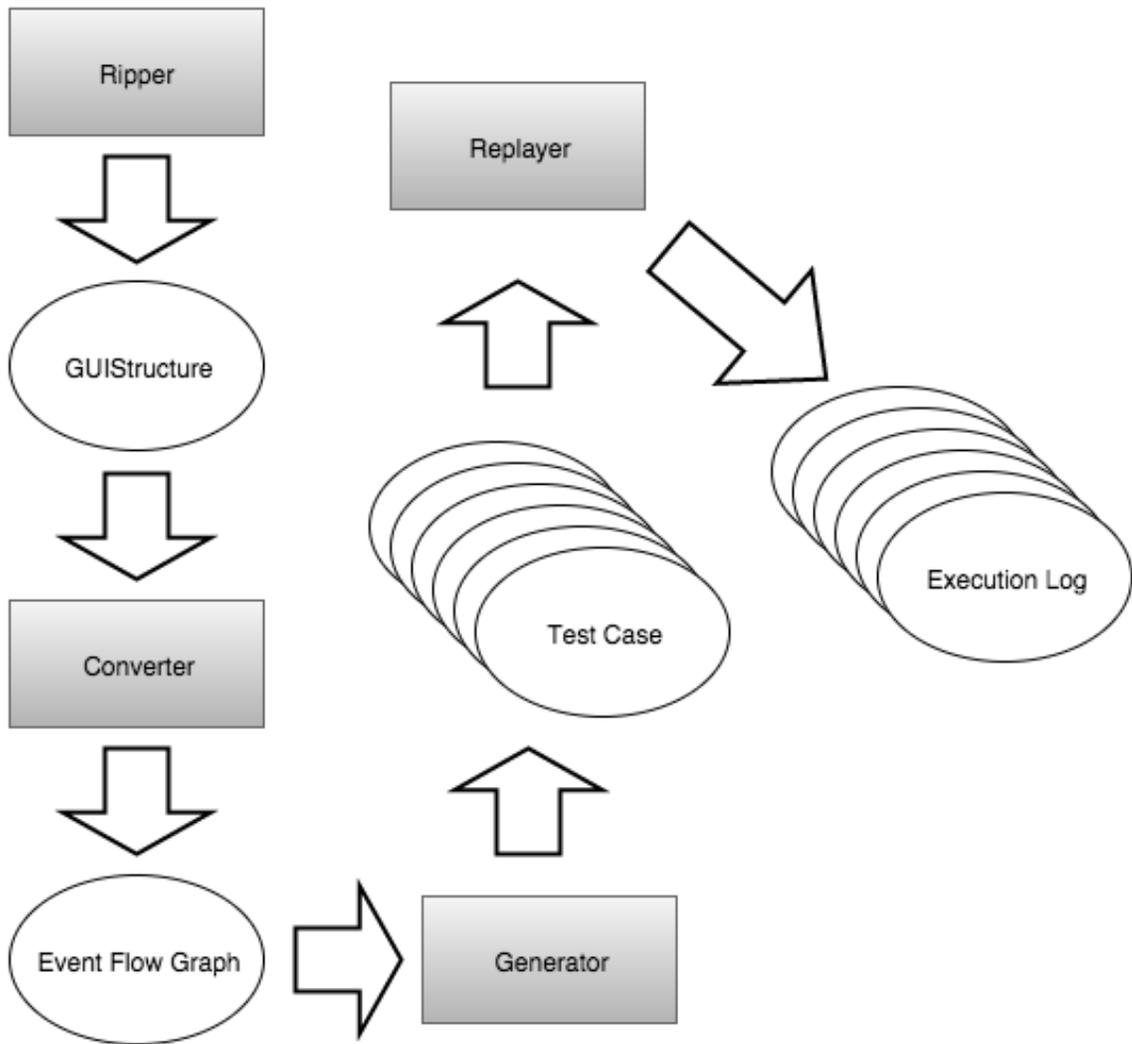
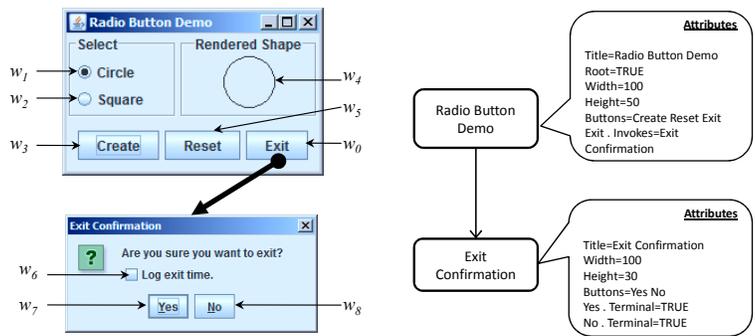
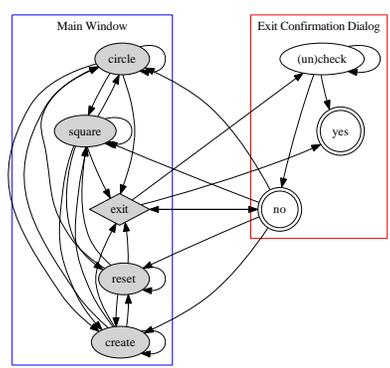


Figure 3.1: GUITAR Standard Workflow for model-based testing, with Artifacts



(a) (b)



(c)

Figure 3.2: GUIStructure and EFG of a simple application

### 3.1.1 Choice of GUI Platform

For my experiments, I use the GUITAR Java tools, which are based on the Java Foundation Classes (JFC)<sup>1</sup> and the Swing UI components. Of the four tools depicted in Figure 3.1, only the Ripper and Replayer have platform-specific components. While the core ripping and replaying logic is shared across platforms, these tools do need to know how to interact with the GUIs of a specific platform. To be sure, GUI-based Java applications are not as popular as more contemporary UI platforms (e.g., Web and Mobile), but the Java GUITAR implementations are the longest-standing platform and have the longest history of use in empirical studies of model-based GUI testing techniques.

Additionally, I do not expect the choice of this or any particular platform to threaten the interpretation of results for my experiments. The binary classification of feasibility which is the focus of this study is platform agnostic, and would require no modification to plug in to additional application platforms. I discuss this platform choice in more detail when elaborating on threats to this research’s validity in Chapter 6.

### 3.1.2 The GUI Ripper

The workflow starts by leveraging GUITAR’s implementation of the GUI Ripping technique [39]. GUI ripping automatically constructs a model of the GUI of the application under test. The GUITAR tool for GUI Ripping is called the **Ripper**. The Ripper takes as input a number of parameters related to launching the application and guiding the ripping process. The output of the Ripper is a single `GUIStructure` file, which captures the windows and widgets the Ripper encountered.

Importantly, GUITAR’s default Ripper uses only a single pass through an appli-

---

<sup>1</sup><https://docs.oracle.com/javase/tutorial/uiswing/start/about.html>

cation, attempting to “rip” as many windows and widgets as possible recursively before exiting and producing a consolidated `GUIStructure`. Figure 3.3 contains pseudocode for the ripping algorithm, reprinted here from Banerjee, et al. [39]. A hypothetical ideal Ripper would require potentially infinite passes through the application (e.g., to capture every event in every possible context).

### 3.1.2.1 Ripper Algorithm

---

$GUI = \phi$

**PROCEDURE** RIP (SUT  $A$ )

$\tau = \text{get-top-level-windows}(A)$  1

$G = \tau$  2

FORALL  $t \in \tau$  DO 3

    RIP-RECURSIVE( $t$ ) 4

**PROCEDURE** RIP-RECURSIVE (Window  $t$ )

$\Psi = \text{extract-widgets-and-properties}(t)$  5

$\epsilon = \text{identity-executable-widgets}(\Psi)$  6

FORALL  $e \in \epsilon$  DO 7

    execute( $e$ ) 8

$C = \text{get-invoked-windows}(e)$  9

$GUI = GUI \cup C$  10

FORALL  $c \in C$  DO 11

    RIP-RECURSIVE( $c$ ) 12

---

Figure 3.3: Pseudocode for the Ripper

Walking through the algorithm as applied to the running example from Figure 3.2,

`get-top-level-windows` would return a single window (the Radio Button Demo window) for the example application. The Ripper would then call `RIP-RECURSIVE` to extract the widgets and properties from this window, and identify and interact with each widget. Upon interacting with the Exit button, a new window would be discovered by `get-invoked-windows` (the Exit Confirmation window), and that window subsequently inspected as well. The result is a `GUIStructure` with two windows, each with several widgets and properties. The `GUIStructure` also captures the fact that clicking the “Exit” button will lead to the spawning of the Exit Confirmation window.

One complexity not immediately obvious from the pseudocode is the identification and utilization of terminal widgets. Upon launch, the Ripper takes a list of terminal widgets properties as a configuration option. The algorithm excludes any widgets with these properties from the return values of `identify-executable-widgets`. For the sake of simplicity, the pseudocode also omits the algorithm’s execution of terminal widgets as needed to exit windows (i.e., once all other executable events have been executed). `get-invoked-windows` also detects closed windows, in the case that a window was closed by the execution of an event unexpectedly.

### 3.1.2.2 Configuration Options and Customizations

As we note in the 2013 paper [5], the Ripper is very sensitive to configuration and customization, such as the configuration of terminal and ignored widgets. In this study, building larger GUI models was not a focus, and as such, I have chosen to avoid any additional complexities of further customization of the Ripper. I use very basic configurations for terminal and ignored widgets (e.g., treating “Ok”, “Cancel”, and other common exit buttons as terminal widgets for all AUTs). For all AUTs, I add enough configura-

tion to terminal and ignored widgets to ensure that the extracted `GUIStructure` models have a reasonable, non-trivial size; but I do not perform any detailed inspections of the `GUIStructure` to search for ways to extract a more complete `GUIStructure`. Importantly, such customizations do not impact the feasibility of test cases. They only affect the size of the extracted `GUIStructure` model.

For example, assume that a terminal widget is missing from the Ripper's configuration. This omission would cause the Ripper algorithm to treat the terminal widget as it would any other widget in the application, causing one or more windows to exit prior to having extracted all possible information. This would lead to a less accurate, smaller `GUIStructure` in terms of widgets and properties; but it would not cause the generation of infeasible test cases later in the workflow.

As another example of the impact of Ripper customizations, in our 2013 paper we describe customization of the Ripper and Replayer to deal with custom widget types [5]. These customizations lead to a much better discovery of the application's GUI, as more windows, widgets, and properties are discovered. In turn, the `GUIStructure` produced by the Ripper's algorithm from Figure 3.3 is a much more accurate model of the application in that it covers much more of the GUI's components. Lacking such customizations (as I do in my experiments), the Ripping algorithm extracts fewer windows, widgets, and properties, and effectively *skips over* the missed components altogether, as the algorithm makes clear. The components (and any children) are completely absent from the `GUIStructure`, so they will not cause any issues with feasibility for any test cases derived later in the workflow.

For consistency, the Ripper and Replayer used in my experiments were configured with support for the same basic GUI components. Any component extracted and placed into the `GUIStructure` by the Ripper was likewise able to be replayed by the Replayer.

### 3.1.2.3 Event Properties

To set up model conversion and test case generation steps, the most important pieces of information extracted by the Ripper are related to events. Structurally, each Window in the GUI has one or more Components, where Components can be Widgets (which are interactive) or groups of Components (which are collections of other components or Widgets). For example, a Menu is a Component in this paradigm. A Menu Button (which is clickable) is a Widget. Widgets subsequently have events, and each event has properties. Most properties are structural, but two key computed properties are also essential to downstream processing:

- **Event ID** : a unique ID for the event, created as a hash of the properties from the widget and its parent window.
- **Event Type** : an inferred category for the event, based on observed effects on the GUI after execution.

Both the EFG and executable test cases use the event ID as a unique identifier. The ID is computed rather than random to assist the Ripper's own algorithm in avoiding infinite loops of property extraction (e.g., in the `get-invoked-windows` method referred to from the pseudocode), and to keep IDs consistent across otherwise identical executions of the Ripper. Because the IDs are based on a hash, any difference in structural properties (e.g., the width, height, or screen location) of the widget will cause it to be considered a new widget for inclusion in the `GUIStructure`.

The event type in this context refers to the effects this event has on the GUI. (To be clear, it does not refer to clicking, typing, or any other *type* of events which can be carried out on a widget). There are five event types, each of which have a specific interpretation

in the event-flow model produced by the Converter in the next workflow step [27]:

- **menu-open**: an event which caused a menu to open.
- **termination**: an event which caused a window to close.
- **restricted-focus**: an event which caused a window to open, if the opened window blocked access to all existing windows.
- **unrestricted-focus**: an event which caused a window to open, if the opened window did not block access to existing windows.
- **system-interaction**: an event which did not cause any apparent change to the GUI.

After computing these values, the `GUIStructure` contains all necessary information for downstream processing.

### 3.1.3 Conversion

The next two steps in the standard GUITAR workflow work from model representations rather than directly with applications. As the next step, the Converter converts the `GUIStructure` model into a Directed Graph called the Event Flow Graph (EFG) [27]. The EFG has GUI events as nodes. Edges in the EFG represent presumed relationships between GUI events. In particular, the EFG captures *may-follow* relations between events in the GUI. If an event  $e_2$  has a connection from event  $e_1$  in the EFG,  $e_2$  should be immediately executable after the execution of  $e_1$ .

As the transition from subfigure (b) to subfigure (c) in Figure 3.2 shows, the EFG abstracts away all information except for events and their relationships. The conversion from GUI information to event relationships is purely structural, based solely on the information available in the `GUIStructure`. The XML representation of the EFG includes

event IDs, *initial* flags for each event, and a matrix representation of the graph itself.

---

```

PROCEDURE GET-FOLLOWS (VERTEX  $V$ )
 $W = \text{window}(v)$ ;  $B = \text{top-level-events}(W)$                                 1
IF EventType( $v$ ) == menu-open                                           2
    IF  $v \in B$                                                             3
        return(MenuChoices( $v$ )  $\cup$  {  $v$  }  $\cup$   $B$ )                        4
    ELSE                                                                    5
        return(MenuChoices( $v$ )  $\cup$  {  $v$  }  $\cup$  follows(parent( $v$ )))        6
IF EventType( $v$ ) == system-interaction                                   7
    return( $B$ )                                                                8
IF EventType( $v$ ) == termination                                         9
    return(top-level-events(invoker( $W$ )))                                    10
IF EventType( $v$ ) == unrestricted-focus                                  11
    return( $B \cup$  top-level-events(invoked( $v$ )))                            12
IF EventType( $v$ ) == restricted-focus                                    13
    return(top-level-events(invoked( $v$ )))                                    14

```

---

Figure 3.4: Pseudocode for computing *may-follow* relationships from an event

Figure 3.4 shows pseudocode for the primary algorithm used in computing the EFG, which computes *may-follow* relations for a given event (adapted from [27]). Specifically, this algorithm computes the set of events which *may-follow* a given event  $v$ . As mentioned above, this algorithm depends on the computed Event Type present in the `GUIStructure` for each event.

For each event type, the algorithm uses the relationships captured in the `GUIStructure`

to compute the events available from the current event. The logic for each case depends on the event type's effects on the GUI, with limitations introduced as windows and menus are opened and closed. `system-interaction` events were observed to have no effect on the GUI during ripping, and likewise can be followed by any events which were available before their execution.

The events available after executing a `menu-open` event depend on whether the event is available at the top level of its window. For example, if a user must click a `File` menu in order to click `Save`, the user can click `File` within its own window at any time; but `Save` can only be clicked once `File` is expanded. The `File` case is line 3 above, and the `Save` case is line 5.

For events which invoke new windows (`unrestricted-focus` and `restricted-focus`), the events which may follow depend on the nature of the window invoked. If the window invoked was *modal*, meaning that it blocked interaction with all other windows, then only events in the new window may follow. If the window was not modal, then the events in the new window as well as any events already available may still follow.

Events in the EFG are also marked with a true/false *initial* value, which indicates whether each event can be executed immediately after application start. All events available from the root windows of the application have this flag set to *true*, and all others have a value of *false*.

#### 3.1.4 Test Case Generation

Next, the Test Case Generator traverses the EFG to generate test cases. The Generator takes as input the EFG produced by the Converter and generation parameters. GUITAR's basic libraries support multiple test case generation strategies. In this re-

search, I chose to use the “random sequence length” test case generator, which randomly selects a given number of test cases from a much larger pool of test cases defined on a criteria of sequence length coverage.

The Generator outputs one file per each requested test case, which I refer to as the “test case input file.” Each file specifies the sequence of events which define the test case. Even for short, constant sequence lengths, tests can include a varying number of steps. Because I am using only *click* events (as the only type provided by the default GUITAR Ripper and Replayer), my test case files include only event and widget identifiers (IDs) - one per step.

---

```

PROCEDURE SEQUENCE-LENGTH-TESTS (EFG G, LENGTH L)
    for v ∈ vertices(G)                                     1
        S = S ∪ GENERATE-SEQUENCES(L, v)                  2
PROCEDURE GENERATE-SEQUENCES (LENGTH L, SEQUENCE POSTFIX)
    IF L == 1                                             3
        P = pathToRoot(first(POSTFIX))                   4
        writeTest(P, POSTFIX)                             5
    ELSE                                                 6
        for v ∈ successors(last(POSTFIX))                7
            NP = concat(POSTFIX, v)                       8
            GENERATE-SEQUENCES(L - 1, NP)                 9

```

---

Figure 3.5: Pseudocode for sequence-length test case generation

Figure 3.5 includes pseudocode for the sequence-length test case generation algorithm. Given a sequence length L, this algorithm generates one test case for each sequence

of length  $L$  in the EFG. Sequences up to length  $L$  are identified by walking forward from each node up to length  $L$ , then a “path to root” is prefixed onto each sequence. The `pathToRoot` method walks backward from a given vertex (i.e., the first vertex in the sequence being covered) until reaching an event which has an *initial* flag set to *true*.

As mentioned, I use a random variation of this algorithm in my experiments. The randomness is introduced after the sequence length generation algorithm completes by choosing a random set of indices from the entire pool of available tests. Therefore, my experiments provide two inputs to the test case generation step: a sequence length for generation of the pool, and a number of test cases to sample. Random selection of test cases is more appropriate for the training and evaluation of the binary classifier, and it also avoids the exponential blowup in runtime required for executing full sequence-length coverage suites.

### 3.1.5 Test Case Replay

After generation of test cases, the workflow includes execution of the generated test cases with the GUITAR Replayer tool. As a piece of software, the Replayer tool is very similar to the Ripper: a platform-specific tool with event and widget awareness. The Replayer, though, simply takes a Test Case, EFG, and `GUIStructure` as input and executes the steps from the Test Case in sequence.

The Replayer only directly outputs a text log of execution results. Most researchers of model-based testing approaches collect additional output artifacts from the application during test case execution, in order to apply a more interesting test oracle [36]. To determine feasibility, however, I am concerned with test case execution only. The Replayer’s log provides enough information for this analysis on its own.

In my experiments, I treat all Replayer failures as a possible indication of an infeasible input test case. I updated the GUITAR Replayer used in my experiments to report four types of errors of its own:

- **COMPONENT NOT FOUND** - A required widget (one required by the input test case) cannot be found.
- **COMPONENT DISABLED** - A required widget can be found in the GUI, but cannot be executed. The widget may be directly disabled, or may be blocked by some other GUI component.
- **STEP TIMEOUT** - The Replayer has failed to carry out a test case event within a reasonable amount of time.
- **JAVA EXCEPTION** - The Replayer has failed due to a Java error (e.g., a bug in the Replayer such as a Java `NullPointerException`).

In all of the cases above, the GUITAR workflow steps prior to the use of the Replayer have produced a test case which was ultimately not executable. For this reason, I consider any such failure a possible indication of test case infeasibility.

### 3.2 Adding a Binary Classifier for Feasibility

To this point, I have described the use of models to extract and process GUI information, and to generate test cases. To predict feasibility, I propose the training and use of a binary classifier as an additional step in a model-based testing workflow. To this end, I use supervised learning to train a binary classifier for test case feasibility. In this section, I elaborate on the machine learning techniques used by this process.

### 3.2.1 Supervised Learning and Binary Classification

Machine learning involves the use of computers to learn patterns from data. In practical applications, the learned patterns are often used to predict something about the nature of unseen data. Within the context of this research, for example, I want to predict the feasibility of a test case prior to its execution. I provide a test case as input to the machine learning algorithm, and expect to receive a prediction (feasible or infeasible) as output. This type of problem is known as a binary classification: classification because I want to categorize the input, and binary because there are only two categories of the predicted value.

A machine learning technique such as binary classification requires at least two complementary algorithms which work in separate phases. During a *training* phase, one algorithm fits a statistical model based on a set of inputs. After training, a separate algorithm can use the trained statistical model to predict outcomes given previously unseen inputs. Mathematically, these algorithms operate on numerically coded *features* of input data, where each feature maps to an input variable  $X_i$  in the underlying statistical model. Algorithms which also know the correct output category of an input during training are known as *supervised* (as opposed to *unsupervised* techniques, which do not know the output category of inputs during training). I use supervised learning to construct the binary classifier for feasibility in this study.

### 3.2.2 Generalized Linear Model

Above, I explain that binary classification requires fitting a statistical model during training, then using that model to make predictions. Assuming a set of  $p$  features  $X_j$  used to predict an outcome  $\hat{Y}$ , a linear model is one that estimates  $Y$  as:

$$\hat{Y} = \hat{\beta}_0 + \sum_{j=1}^p X_j \hat{\beta}_j \quad (3.1)$$

Under this model, training involves solving an optimization problem which chooses the coefficients  $\hat{\beta}$  that lead to the best approximations for  $\hat{y}$ . After training, predictions can be made by simply summing the  $X_i \hat{\beta}$  terms from the features of a previously unseen data point. The  $\hat{\beta}_0$  added to every prediction is known as the *intercept* of the model. The intercept may also be added to the model by modeling an extra feature term  $X_0$  which always has a value of 1. From the remainder of this section, I assume that formulation for simplicity.

The *Generalized Linear Model* or GLM [40] provides a general formulation of such models. The GLM formulation makes three fairly general assumptions:

1. Outcomes are assumed to come from a probability distribution function, from a broad class of certain acceptable forms with parameters  $\theta$  and  $\phi$ .
2. Predictions are made with a linear predictor  $\eta$  (i.e., the linear combination of  $X_j$  and  $\hat{\beta}_j$  given above).
3. A *link* function  $g$  provides an estimate for the parameter  $\theta$  of the probability distribution function, given labeled training data.

Under this formulation, estimating the parameters  $\hat{\beta}_j$  means solving an optimization problem which minimizes the difference between the observed and predicted values for  $Y$ . The original authors of the GLM showed an expectation maximization formulation of this model fitting which works for all GLMs [40]. The training problem can also be expressed in terms of a loss function defined on  $Y$  and  $\hat{Y}$ :

$$\min_{\hat{\beta}} \ell(Y, \hat{Y}) \tag{3.2}$$

The acceptable forms of the probability distribution function include the very broad exponential family of probability distribution functions, including the normal, gamma, and binomial distribution functions, among many others. The simplest and most common form of GLM is linear regression, which assumes a normal distribution and only requires the identity function as its linking function. For binary classification such as my case in this study, logistic regression provides a more robust alternative.

Formulated as a GLM, logistic regression uses a binomial distribution and *logistic* function as a loss function. Logistic regression makes sense for binary classification problems, because its predicted values are defined on a range from 0 to 1. Further, the shape of the logistic function more easily fits data which prefer values at exactly 0 and 1.

### 3.2.3 Adding Regularization with Lasso

As described thus far, I could use logistic regression over all available features to fit a model for feasibility given training data. In practice, this model would quickly encounter issues with convergence, and would tend to *overfit* the training data, especially when the number of training examples is much less than the number of features in the training data. (The specific features I consider for the model are covered in the next section.)

To counter these problems, I could add more tests to the training data; but test execution is expensive, and adding tests would tend to add more features until a very large number (e.g., 100s of thousands or more) tests were in the training set. Regularized models provide a more stable alternative. Revisiting the loss function formulation from above, regularized models add a new term to the optimization problem:

$$\min_{\hat{\beta}} \ell(Y, \hat{Y}) + \lambda R(\hat{\beta}) \quad (3.3)$$

A metaparameter  $\lambda$  accompanies the new term, meaning that there will now be many models to choose from, given the same training data. For this study, I use cross validation on the training error to choose the best model after training at several values of  $\lambda$ .

There are multiple options for the regularization function  $R$  above, which should cause the optimization to prefer the simplest possible model (i.e., with the lowest amount of “mass” spread across estimated coefficients). If possible, coefficients should also be allowed to reduce to 0 in the model, so that they may be ignored altogether.

Tibshirani formalized the use of an  $\ell_1$  norm as a regularization term, which he called the “least absolute shrinkage and selection operator” or “lasso” [41]. The  $\ell_1$  norm is given by taking the sum of the absolute values of the coefficients:

$$R = \|\hat{\beta}\|_1 = \sum_{j=1}^p |\hat{\beta}_j| \quad (3.4)$$

As the name suggests, use of this term for regularization does allow for both shrinkage of coefficients and selection of parameters, by allowing unimportant coefficients to have estimates of 0. More recently, Friedman, et al. provide a very fast solver for the lasso optimization problem known as `glmnet` [42]. I use their publicly available R package’s implementation of this algorithm for the training and predictions using lasso in this research. The `glmnet` package provides support for logistic regression, as required by our model.

### 3.2.4 Feature Selection

The feasibility model used in this study has a predicted variable *isInfeas* and features available from test case replay input artifacts. I use `glmnet` to fit a binomial model on training data, using logistic regression with regularization by the lasso. To label the predicted variable for the training phase, I extract the observed value of *isInfeas* from test execution logs.

As discussed in Section 1.3.3, and reinforced by existing work discussed in Section 2.3, infeasibility in model-based test cases occurs because of model imprecision. An event sequence which appears executable according to the model is not executable against the real application.

In the case of the GUITAR Ripper and test case generation based on the EFG, Nguyen and Memon show that *missing context* is the cause of infeasibility [17]. The Ripper, during its single pass through the application to discover the `GUIStructure`, is observing available events subject to the context of any events which have already been executed. The EFG, and any test cases generated from it, do not have the Ripper’s full context available. Nguyen and Memon go on to show that generated test cases may have events added or removed to make them feasible.

Assuming that the order, presence, and absence of events in a test case contribute directly to its feasibility, the model requires features related to the events in a test case as well as their order. Table 3.1 shows the set of candidate features for the feasibility model used in the binary classifier for this study.

For each test case, I extract four classes of features. I extract both *n-grams* and *before-pairs* from the sequence of event IDs associated with the test case, and also n-grams

Feature Group	Description	Example
Event ID n-grams	Ordered subsequences of length 1-2 of Event IDs in test case	ngram_e1, ngram_e2, ngram_START_e1, ngram_e1_e2, etc.
Event ID Before Pairs	Pairs of event IDs “ex_before_ey” where ex occurs anywhere before ey in test case	e1_before_e2, e2_before_e3, etc.
Event Type n-grams	Ordered subsequences of length 1-2 of Event Types in test case	ngram_SYSTEM, ngram_TERMINAL, ngram_START_SYSTEM, ngram_SYSTEM_TERMINAL, etc.
Event Type Before Pairs	Pairs of event types “tx_before_ty” where tx occurs anywhere before ty in test case	EXPAND_before_SYSTEM, SYSTEM_before_TERMINAL, etc.

Table 3.1: Candidate Features for Feasibility Model

and before-pairs from the sequence of event types. All of this information is directly available from the test case input files which define test cases.

n-grams [43] are commonly used as basic features in language models. In language models, researchers extract n-grams from sentences, which are treated as sequences of words. In the feasibility model, my sequences are test cases, and I propose that event IDs and types are analogous to words for the purposes of the n-grams used in this experiment. Importantly, I am not using a probabilistic n-gram model for any predictions in this study. I am simply using n-grams as features of test cases for a more generic classifier.

For the extraction of n-grams with length=2, I insert a *START* event, which is likewise a common practice in language modeling. I expect the presence of a *START* token to allow the model to capture any infeasibility due to events which were assumed to be reachable from the initial application state. I do not use an *END* token, as the end

of a test case should not be a source of infeasibility. For both kinds of n-grams, I extract subsequences of length 1 and 2.

While n-grams (for sequences longer than length 1) represent a *strictly ordered* dependency between events, I also add a second class of features for each of event IDs and types. I call this second class before-pairs. Features capturing these pairs are still ordered, but allow the preceding event to occur anywhere in the sequence rather than only immediately before. I only consider single events occurring before other single events.

For example, consider a test case with sequence of event IDs  $e_1 \rightarrow e_2 \rightarrow e_3$ . This test case would have n-grams of length 1 of  $e_1$ ,  $e_2$ ,  $e_3$ , and of length 2 of  $\text{START} \rightarrow e_1$ ,  $e_1 \rightarrow e_2$ , and  $e_2 \rightarrow e_3$ . It would have before-pairs of  $e_1\_before\_e_2$ ,  $e_1\_before\_e_3$ , and  $e_2\_before\_e_3$ .

As mentioned above with regard to regularization, the candidate set of features described here will be very large for any non-trivial application. The added regularization in the model should allow all but the most helpful coefficients to zero out during training.

Another source of features is the “level” of each feature considered for inclusion in the model. Because all of the candidate features for the feasibility model are binary, the presence *or absence* of any feature could potentially correlate with the predicted variable of feasibility. For this reason, I include two levels for each original feature in the model used by the classifier (notated as 0 and 1 for the absence and presence of each original feature, respectively). Given our use of a regularization parameter in the regression, these features should not confound the model or its interpretation.

## Chapter 4: Implementation Details

Carrying out a large-scale empirical study of testing techniques requires a robust execution infrastructure. To support my research, I first establish the following requirements for such an infrastructure:

- **Portable** : The infrastructure should be adaptable to any group of Linux machines.
- **Persistent** : The infrastructure should support persistence and easy backup and restore operations for input and output artifacts.
- **Configurable** : The infrastructure should support fully automated backup and restore of all machine configurations.
- **Compatible** : The infrastructure should support execution of arbitrary Java tools and R scripts.
- **Parallel** : The infrastructure should support the execution of many GUI test cases simultaneously.

Figure 4.1 shows the core runtime components of the infrastructure I have developed which achieves these goals, and dependencies between those components. I use Docker<sup>1</sup> to develop virtual machine images to automate the configuration of every infrastructure component. Each of the five infrastructure components in Figure 4.1 has a corresponding Docker component (called an “image” at build time, and a “container” at runtime) precisely describing its creation and configuration.

---

<sup>1</sup><https://www.docker.com/>

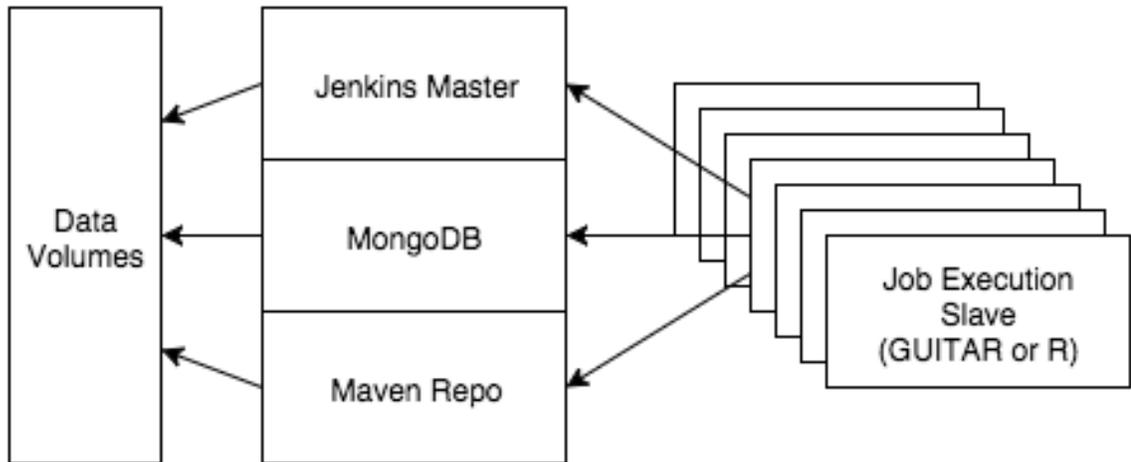


Figure 4.1: Infrastructure Components and Dependencies

Each task to be performed as a part of this research is automated via a Jenkins CI<sup>2</sup> job. The Jenkins Master controls the execution of automated jobs, and actual execution of commands occurs in completely separate slave containers. Before each job finishes, artifacts are persisted to one of a MongoDB<sup>3</sup> database or the Apache Maven<sup>4</sup> Java library repository (MongoDB for test execution and experiment artifacts, and Maven Repo for compiled code artifacts).

The Jenkins Master, MongoDB, and Maven Repo components use existing open-source tools directly off the shelf (Jenkins CI, MongoDB, and Sonatype Nexus<sup>5</sup>, respectively). The Job Execution Slave components use open source tools such as the GUITAR Java toolchain, Gradle, and R to carry out the specifics of the automated model-based testing and analysis tasks required for this research. I elaborate on the use of Docker to configure each of these infrastructure components (including two separate types of Job

<sup>2</sup><https://www.jenkins-ci.org/>

<sup>3</sup><https://www.mongodb.org/>

<sup>4</sup><https://maven.apache.org/>

<sup>5</sup><http://www.sonatype.org/nexus/>

Execution Slaves) in Section 4.1.

Persistence of experiment artifacts requires the development of a new tool, which I call TestData. I use this tool from within Job Execution Slave components during job execution to persist artifacts as appropriate. I describe the implementation of this tool in Section 4.2.

Jenkins CI natively supports execution of automated jobs, including concurrent execution of jobs in separate Job Execution Slaves. However, coordinating batches of parallel jobs requires custom code using the HTTP API provided by the Jenkins Master. I elaborate on my use of Jenkins CI jobs and my approach for parallel job execution in Section 4.3.

I make extensive use of R<sup>6</sup>, an open-source platform for statistics and graphics, for the construction and analysis of the binary classifier in my experiments. I elaborate on my custom R scripts (which make heavy use of third-party libraries) in Section 4.4.

#### 4.1 Portable Configurations with Docker

Docker is an open-source tool for defining and running software within containers, which “sandbox processes from each other [44].” Docker containers represent a lighter-weight virtualization alternative as compared to traditional Virtual Machines (VMs). The containers also make use of shared, read-only copies of filesystems to optimize disk usage. Ultimately, however, the processes running within containers consume at least the same amount of resources (CPU, Memory, File I/O, and otherwise) as the same processes would consume if run on the host operating system directly.

In terms of platform support, Docker runs on the majority of Linux platforms and

---

<sup>6</sup><https://www.r-project.org/>

inside of Linux VMs on other host operating systems. Moving my infrastructure components and their configuration to Docker images allows for replication of those components on any available hardware running a native Linux operating system or VM, as required.

To use Docker, I follow this lifecycle:

1. (Optional) Build a Docker *image* based on a customized script called a Dockerfile<sup>7</sup>.
2. Given an image ID and runtime properties such as *mapped data volumes*, *ports*, and *environment variables*, start a Docker container.
3. When done using the container, manually stop.

The broader Docker ecosystem also readily supports quick and reliable use of existing, publicly provided container images. The Docker Hub<sup>8</sup> indexes a full set of publicly available container images, and allows them to easily be “pulled” onto a machine via the Docker command-line interface (CLI). The images on the Docker Hub are open-source, so that their configuration can be completely inspected prior to use or enhancement. Container images support inheritance as well, so that images based on publicly available configurations can be further customized for particular use cases.

To support this research effort, I have leveraged five unique Docker images, as shown in Table 4.1. Docker images can be enhanced at build time to install and configure additional tools, and can also be configured to use mounted filesystem locations called volumes. The “Base Image ID” column in the table refers to the base image used for each Docker container I leveraged, where the base image is given as coordinates into Docker Hub (as of Summer 2015). The Enhancements and Volumes columns describe at a high level the Dockerfile enhancements and volume usage leveraged by each container, respectively. I

---

<sup>7</sup>The full Dockerfile spec is available online at <https://docs.docker.com/v1.8/reference/builder/>

<sup>8</sup><https://hub.docker.com/>

Name	Base Image ID	Enhancements	Volumes
Jenkins Master	jenkins	None	Job artifacts, configuration files
MongoDB	dockerfile/mongodb	None	Data directory
Maven Repo	conceptnotfound/sonatype-nexus	None	Jar files, index
GUITAR Slave	ubuntu:14.04	Install Jenkins agent, GUITAR tools, Subject applications	None
R Slave	ubuntu:14.04	Install Jenkins agent, R, R packages	None

Table 4.1: Docker images

elaborate on my general approach for leveraging arbitrary Docker volumes in Section 4.1.1 and on the details of each container in Section 4.1.2 below.

#### 4.1.1 Data Volumes

Data volumes also play a critical role in my usage of Docker. Docker images include a snapshot of the entire filesystem at creation time. For “always-on”, data-backed components which we would like to efficiently backup and restore from, full persistence at the image level would require a great deal of disk space, versioning, and other additional engineering overhead.

Alternatively, Docker provides options for mounting data into containers at runtime rather than baking data into images at image creation time. When Docker containers are started, local filesystem locations from the server which is running Docker can be mapped to a container’s filesystem. Also, containers can be configured at runtime to inherit all volume mappings from any already running container.

I leverage these features to create and populate a “data container” with all of my required data volumes defined. I can then run backup and restore operations from minimal containers which mount the volumes from this data container and read or write into them. I can also mount these volumes into any container requiring an easily backed-up data volume (such as Jenkins CI or MongoDB).

The Volumes column in Table 4.1 describes the volumes I use with each Docker image. I elaborate on the details of volume preparation and usage for each relevant image alongside the details of each image below.

#### 4.1.2 Docker Image Details

As described above, each Docker image starts from a base image, and executes one or more Linux commands within the base image to create a new reusable image. The image can then be launched with one or more supporting data volumes and several additional configuration options. In the case of this infrastructure, I use command-line options for mapping data volumes, TCP ports, and environment variables in particular.

##### 4.1.2.1 Job Execution with Jenkins CI

The Jenkins Continuous Integration Server<sup>9</sup> (also known as Jenkins CI) is used across the software development industry for the configuration of automated building and testing of software. Jenkins jobs are also very generic, and many now use Jenkins CI for the automation of more generic tasks. Jenkins jobs perform sequential steps such as version control checkouts, environment configuration, shell script execution, and many others. The Jenkins ecosystem also depends on hundreds of open-source plugins which provide their own customized, drop-in functionality for jobs.

---

<sup>9</sup><https://www.jenkins-ci.org/>

Jenkins also supports a Master-Slave mode of operation, whereby a Master server (which I am describing here) can be configured to use zero or more slaves to carry out jobs. I describe the GUITAR Slave and R Slave containers, which are used as slave nodes in this Jenkins setup.

The Jenkins Master also hosts a very elaborate HTTP API for starting, stopping, and querying the status of jobs. I utilize this API from Groovy scripts to coordinate the parallel execution of batches of jobs, when required.

For this research infrastructure, I did not directly extend the provided, official Jenkins CI Docker container. Instead, I used a data volume to persist all Jenkins configurations. With this volume in place, I manually configured my Jenkins Master server and then took a backup of the volume. Over time, I made many configuration changes to the Jenkins master to support my experiments and their execution on this infrastructure, including the use of many plugins. Significant Jenkins configurations included:

- The Xvfb plugin allows individual Jenkins jobs to be wrapped in a “virtual frame buffer,” so that jobs which require an X11 graphical environment use a headless (virtual) graphical environment rather than requiring a full-fledged display. This headless operation was essential to running GUI tests in parallel.
- Individual Jenkins jobs can “opt in” to concurrent execution of the same job, whereby each execution has an isolated working directory on a slave container.
- Jenkins jobs can be configured to delete their local workspace (e.g., any checked-out or derived files) when starting, and to archive important files upon completion.
- Jenkins jobs can use an “Execute Shell” step to run any Linux commands, including the use of any tools (e.g., GUITAR tools, R, etc.) which are installed within slave

containers.

With these basic features properly configured on the Jenkins Master, I am able to configure robust, automated Jenkins jobs for the execution of GUITAR tools and R scripts required by the experiments in this work.

#### 4.1.2.2 Test and Experiment Artifact Persistence with MongoDB

MongoDB is a full-featured document store, sometimes referred to as a “NoSQL database” because it does not use SQL as a query interface or use relational database schemae. In particular, the MongoDB server provides database instances. Each database contains zero or more collections. Each collection contains zero or more objects. Default collections contain text objects, but MongoDB also supports collections of binary objects through its GridFS extension <sup>10</sup>. Objects are represented in a schema-less, Javascript Object Notation (JSON)-like format, such that each Object has a unique ID and zero or more properties which can be primitives (e.g., Strings, Integers, Floating-point numbers), Objects, or Lists of Objects.

For my purposes, I use a MongoDB server to persist model-based testing and experiment artifacts. I interact with the MongoDB service itself through the TestData tool, described in Section 4.2. In terms of the base Docker image, I use a predefined, hardened MongoDB image from Docker Hub as-is, and mount a data volume to a running container of this image. The data volume for this container provides a “data” directory which is the root of MongoDB’s storage within the container’s filesystem.

In the case of this container, I only provide a data volume for the data directory location when starting the MongoDB container. The server nor the volume require additional

---

<sup>10</sup><https://docs.mongodb.org/manual/core/gridfs/>

manual configuration steps.

#### 4.1.2.3 Code Library Artifact Persistence with Maven and Sonatype Nexus

Apache Maven is one of the most popular build tools within the Java development ecosystem. One of the most attractive features of Maven as a build tool is its integration with standardized artifact repositories which are referred to simply as “Maven repositories” or “Maven repos.” Even as the build tool is beginning to be replaced with tools such as Gradle <sup>11</sup>, these next-generation tools still leverage Maven’s repository layout and community-hosted repositories.

At a high level, Maven repositories allow Java programs to publish and retrieve Java Archives (Jars, or “jar files”) by making HTTP calls to a repository URL. This capability greatly reduces the footprint of Java programs by allowing them to retrieve dependent libraries on-demand.

Sonatype Nexus is an open-source tool for configuring and serving Maven repositories. As with the Jenkins Master and MongoDB Docker containers, I leverage an existing, publicly available Docker image for Nexus. I mount a volume to allow persistence of Nexus artifacts and configuration files. I also manually configure Nexus to allow deployments by a given authenticated user and password combination. With this configuration in place, automated jobs can now push their artifacts to the Nexus Maven repository, and use any stored artifacts as required.

#### 4.1.2.4 GUITAR Slave

The Jenkins Slave containers are unique among the Docker containers included in this infrastructure, in that they are customized at the Docker image level rather than

---

<sup>11</sup><http://gradle.org/>

through data volumes. Image-level customization for slave containers is possible because of the archiving capabilities of the Jenkins Master server. Because important artifacts are archived to either MongoDB, the Maven Repo, or back to the master server, there is no need to persist any artifacts across executions of slave containers.

The GUITAR slaves in particular need to be able to run all of the tools from the GUITAR Java Toolchain. As Java tools, they can be launched with the Linux `java` command on any machine with the Java Runtime Environment (JRE) installed. They also require standard Linux utilities such as `wget`, `curl`, and `unzip` for unpacking and building software.

I have chosen to use Gradle rather than direct Linux commands to call Java tools, to simplify the management of Java tool dependencies. Gradle scripts are designed to easily fetch Java libraries from a Maven Repository at runtime, and can easily incorporate inline and external Groovy<sup>12</sup> scripts to drive Java tool execution.

You can view the full source of my GUITAR Slave Dockerfile in Appendix A.

#### 4.1.2.5 R Slave

While the majority of the work involved in my experiments was execution of GUITAR tools, I also require additional automated jobs for construction of the binary classifier and analysis of its predictions. In particular, I use the statistical software R<sup>13</sup> for model construction and data analysis. The Docker image for the R Jenkins Slave has R and all of my required R packages installed, so that they are available to relevant Jenkins jobs.

You can view the full source of my R Slave Dockerfile in Appendix A.

---

<sup>12</sup><http://groovy-lang.org/>

<sup>13</sup><https://www.r-project.org/>

## 4.2 Persistence with TestData and MongoDB

The GUITAR framework has long been used to carry out empirical evaluations of model-based testing techniques. Initiatives such as COMET [45] have sought to address the repeatability of empirical evaluations of event-based testing techniques more generally. In my observation, persistence of data and configurations continues to be a major difficulty in the repeatability of studies, especially when tools (such as the tools of the GUITAR framework or others) are so sensitive to configuration.

As a part of this research, I have chosen to implement a persistence framework for testing artifacts, which should be suitable for application across testing frameworks and empirical evaluations in the future. To this end, I implement a new tool called `TestData`, backed by the MongoDB Java API<sup>14</sup> and a running MongoDB server. Figure 4.2 shows the data model used by `TestData`.

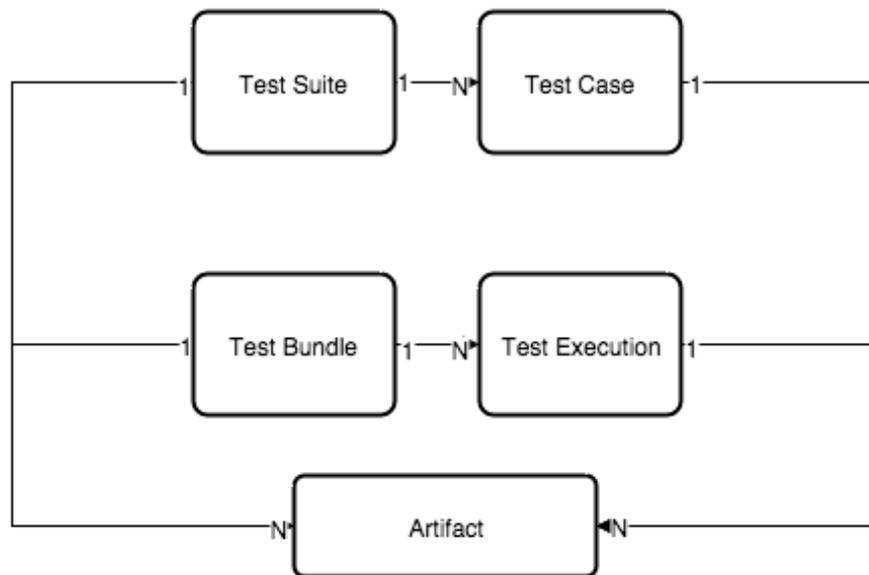


Figure 4.2: `TestData` Persistence Model

<sup>14</sup><http://mongodb.github.io/mongo-java-driver/2.13/getting-started/quick-tour>

At a basic level, `TestData` persists metadata about test cases and test case executions (or simply, test executions). Groups of test cases are called *suites*, and groups of test executions called *bundles*. All objects in the data model (suites, bundles, test cases, and test executions), can have one or more associated artifacts. Artifacts can be arbitrarily defined, but for my experiment, are artifacts such as the `GUIStructure`, EFG, test case inputs, and others produced by GUITAR tools during my experiments.

#### 4.2.1 ArtifactProcessors

In order to be persisted with `TestData`, each artifact must have a corresponding `ArtifactProcessor` implementation. I implement `ArtifactProcessors` for each GUITAR workflow artifact, as well as additional experiment artifacts. Table 4.2 details the `ArtifactProcessors` implemented as a part of this research.

<b>Processor</b>	<b>Artifact</b>	<b>Owner</b>
GUIProcessor	GUIStructure XML	Test Suite
EFGProcessor	EFG XML	Test Suite
TestcaseProcessor	Test Case Input XML	Test Case
FeaturesProcessor	Test Case Features	Test Case
LogProcessor	GUITAR Replayer Log Text	Test Execution

Table 4.2: `TestData` `ArtifactProcessor` Implementations

The current implementation of `TestData` focuses on persistence of the GUI Structure, EFG, and Test Case Input artifacts produced by GUITAR tools. Additionally, the Replayer logs are serialized by the `LogProcessor` into a simple JSON representation.

The `FeaturesProcessor` extracts and serializes a JSON representation of arbitrarily labeled test case features, given an existing test case input file. The specifics of this implementation ultimately depend on the features included for selection during experiments.

The JSON representation simply lists strings of text, allowing the implementation to easily be adapted for any desired features.

Method	Description
jsonFromOptions	Given a set of input options (e.g., a file path), provide a JSON representation of an artifact
objectFromOptions	Given a set of input options (e.g., a file path), provide an object representation of an artifact
jsonFromObject	Given an object, provide a JSON representation of an artifact
objectFromJson	Given a JSON representation of an artifact, provide an object representation

Table 4.3: ArtifactProcessor Interface Methods

At an implementation level, each `ArtifactProcessor` implements an interface of four methods. The methods and their purposes are summarized in Table 4.3. A processor specifies how to serialize and deserialize a single artifact. Serialization takes a list of artifact-specific options (such as file paths or test case IDs) as input, and produces Java Object and JSON representations. Deserialization takes the JSON as input and produces a Java Object.

#### 4.2.2 Additional Processing

In addition to the GUITAR artifact processing described so far, `TestData` also includes methods for the persistence of suite-level results and suite-wide feature sets. These two objects can be thought of as “experiment artifacts” rather than test artifacts, and currently have custom implementations in `TestData` which do not follow the `ArtifactProcessor` interface. They are included as a convenience to support smoother downstream processing (i.e., as an alternative to performing the same complex processing of MongoDB objects in

R).

As discussed in Section 3.1.5, the Replayer logs produced as the output of test case execution indicate whether a test case should be considered infeasible. To process results, a method `postResults` takes as input a list of test bundle IDs and inspects Replayer logs for each test execution in each bundle. Test cases are counted as feasible if Replayer logs contain no failures for any input bundle. Test cases are counted as infeasible if Replayer logs contain failures for all input bundles. Test cases which have inconsistent results over the input bundles are marked as inconsistent, so that they may be excluded from downstream analysis.

A second method, `postFeatures`, takes as input a test suite ID and adds Test Case Features artifacts to each test case, which contain the features associated with the test case. Then, the method stores the combined set of unique features from across all test cases in a separate object. Note that `postFeatures` runs on test case input artifacts only, and does not process any execution logs.

### 4.3 Execution of Automated Jobs in Parallel with Jenkins CI

As introduced in Section 4.1.2.1, Jenkins CI is a general-purpose job automation tool with many plugins designed to carry out automation of “continuous integration” tasks in software development. Table 4.4 details the specifics of each of the Jenkins jobs I have automated as part of this research. With the exception of the `replay-suite` job, all jobs automate the sequential execution of a sequence of Linux commands on a single Jenkins Job Slave node. Each job calls a sequence of Docker commands, GUITAR tools, or R scripts. The Jenkins slave Docker images (as described in Sections 4.1.2.4 and 4.1.2.5) have GUITAR and R tools built in, respectively; so the Linux command steps in each

<b>Job Name</b>	<b>Description</b>	<b>Tools Used</b>
build-libs	Build GUITAR TestData tools from source	Maven
prepare-aut	Build and package subject applications from source	Maven, Tar
build-jslave	Build GUITAR and R Jenkins slave images	Docker
start-slaves	Start GUITAR and R Jenkins slave containers	Docker
generate-random	Run the GUITAR Ripper, Converter, and Generator to generate test suites	GUITAR, TestData
replay-test	Execute a single test with the GUITAR Replayer, on a single slave	GUITAR, TestData
replay-suite	Execute a generated test suite in parallel	TestData, Jenkins API
post-results	Process test execution logs from an executed suite for feasibility	TestData
post-features	Process a test suite to collect combined set of features	TestData
prepare-data	Construct training and test data structures from posted results and feature sets	R
predict	Train a binary classifier and evaluate its ability to predict feasibility	R

Table 4.4: Automated Jobs

Jenkins job can simply call these tools as required.

Working from the top of Table 4.4, the first four jobs (`build-libs`, `prepare-aut`, `build-jslave`, and `start-slaves`) automate the process of creating the GUITAR and R Jenkins Slave container images, and starting containers based on these images. The `generate-random` job automates the execution of the GUITAR workflow, including calls to the Ripper, Converter, and Generator. At the end of `generate-random`, a new MongoDB database has been initialized with a new suite of test cases. The `replay-suite` launches `replay-test` jobs for each test case in any given suite, allowing the jobs to run in parallel over the available set of GUITAR Jenkins Slaves.

The next three jobs listed in Table 4.4 (`post-results`, `post-features`, and `prepare-data`) run in order to process the results of test case execution, to add test case features to test cases in a given suite, and to build an R object containing all relevant data from several test suite executions. Finally, the `predict` job uses the object produced by `prepare-data` as input, and trains then evaluates a feasibility classifier.

### 4.3.1 Replaying Test Cases in Parallel

In addition to the sequential jobs outlined above, I also leverage parallel execution with Jenkins CI to achieve my infrastructure requirements. In particular, I leverage the Jenkins Master’s REST API to launch and monitor the progress of a batch of child jobs and exit upon the batch’s completion. This distributed execution pattern can be applied to any batch of work that can be subdivided into independent jobs. In my experiments, I use this approach to run GUITAR Replayer execution jobs in parallel.

Listing 4.1 contains the Groovy code for the batch execution script used by the `replay-suite` Jenkins job. The input parameters to the script specify the test suite details. The script uses a `TestData` instance called “manager” to fetch the test IDs of the given test suite. For each test ID, the script makes a call to the Jenkins REST API which starts the single-node `replay-test` Jenkins job.

The final four lines in the script add delay as required to avoid overloading the Jenkins master with HTTP calls when there are no available slave nodes for execution. In particular, the script always waits at least 10 seconds between jobs. The script waits for additional time before submitting the next job if necessary, until the number of “waiting builds” (builds already submitted but which do not have an available slave node for execution) is less than 6 (`BUILD_COUNT_THRESHOLD + 1`). The script exits once all

jobs for the test suite's execution have been queued on the Jenkins master.

Listing 4.1: Parallel GUITAR Replay - replay-suite

```
static int BUILD_COUNT_THRESHOLD = 5
for(String id : manager.getTestIdsInSuite(args[2])){
    count++
    // build Map of params
    def jobParams = new HashMap<String, String>();
    jobParams.put("AUT_NAME", args[0])
    jobParams.put("DB_ID", args[1])
    jobParams.put("SUITE_ID", args[2])
    jobParams.put("TEST_ID", id)
    jobParams.put("BUNDLE_ID", args[3])

    // Use Jenkins client to launch job
    jenkinsClient.submitJob("replay-test", jobParams)

    // Wait at least 10 seconds
    sleep(10000)

    // Wait longer if there are waiting jobs
    while(jenkinsClient.getAwaitingBuildCount() > BUILD_COUNT_THRESHOLD) {
        sleep(1000)
    }
}
```

## 4.4 Custom R Scripts

Once tests have been executed with the GUITAR Replayer, I import the data into R for use with a feasibility classifier. I develop two separate R scripts for loading data from MongoDB (`prepareData`) and training and evaluating the binary classifier (`predict`), respectively. Each of these scripts also has a corresponding sequentially executed, automated job as shown in Table 4.4.

### 4.4.1 Prepare Data

Recall that the `TestData` tool places multiple artifacts into a MongoDB instance during the model-based testing workflow. The preprocessing of the `postFeatures` and

postResults methods of TestData, as described in Section 4.2.2, add a number of test case and test suite-level artifacts which simplify the logic needed for data preparation. The relevant data are stored in three types of objects:

- Results objects associated with each test suite indicate which tests from the suite are classified as feasible, infeasible, and inconsistent.
- Feature Group objects associated with each test suite contain a combined list of all features of test cases in the test suite.
- Test Case Feature objects associated with each test case contain a list of all features of the test case.

The `prepareData` R script fetches relevant artifacts and converts their contents into R objects, which can be more readily used by existing R libraries for the remaining analysis. The script takes MongoDB connection details and TestData artifact details as input. As output, the script writes a binary representation of a custom R object to Amazon's Simple Storage Service (S3)<sup>15</sup> which includes the test data, training data, and their labeled execution results in a pre-processed format. I use the third-party libraries `rmongodb`<sup>16</sup> and `RS3`<sup>17</sup> for interaction with MongoDB and S3, respectively.

The script first creates a matrix with size  $N$  rows by  $M + 2$  columns, where  $N$  is the number of test cases and  $M$  is the combined number of test case features. The two additional columns are for variables *isInfeas* and *isTraining*. As discussed in Section 3.2.4, *isInfeas* is the dependent variable for the classifier - an indication of whether the test case is considered infeasible (1) or not (0). Similarly, *isTraining* encodes whether a given test case is part of training data (1) or not (0). The remaining columns encode whether a

---

<sup>15</sup><https://aws.amazon.com/s3/>

<sup>16</sup><https://cran.r-project.org/web/packages/rmongodb/index.html>

<sup>17</sup><https://github.com/Gastrograph/RS3>

specific feature is present or not, where each column represents a single feature.

The matrix is then converted into a `data.frame` object. As a part of the conversion, all of the rows are treated as records in a dataset, and columns treated as variables. R detects that each of the variables in my particular datasets are categorical (which R refers to as *factors*) with two levels each. The levels are ultimately encoded as “1” and “2” by R, but this label is inconsequential for interpreting the data.

Finally, the `data.frame` is split into separate test and training objects, and pre-processed for use with the `glmnet` package for training of the classifier. A helper function `loadData` is particularly important. This code prepares the training and test matrices into the `model.matrix` required by `glmnet`. The snippet in Listing 4.2 from `loadData` shows the construction of a matrix from an R `data.frame` object of Training data. The code uses the `model.matrix` function, with some additional options to enable the use of one feature per factor of each feature. An analogous transformation is carried out on the Test data.

Listing 4.2: Converting data.frame to model.matrix

```
# Prepare training matrix
cat('Creating training matrix', '\n')
xm=model.matrix(isInfeas ~ . - 1, data=train.data, contrasts.arg = lapply(train.
  data[sapply(train.data, is.factor)], contrasts, contrasts=FALSE))
x=apply(xm, 2, as.numeric)
y=as.numeric(train.data$isInfeas)
```

A parent object with all components required by `glmnet` is written out in binary form, then saved to an S3 location. The full text of the `prepareData` script, and a script `common.r` containing common functions such as `loadData` which support it, are provided in Appendix B.

#### 4.4.2 Train and Predict

Given the preprocessing done by the `prepareData` script, the `predict` script simply loads the parent object output by the `prepareData` script and runs `glmnet` steps for training and prediction using logistic regression and lasso. Listing 4.3 contains a pertinent snippet for these steps from the `predict` script.

The script uses a standard R utility function to load the binary parent object written out by the `prepareData` script. It uses the `cv.glmnet` function from the `glmnet` package for training. This function trains several logistic regression classifiers with lasso, and selects the best classifier using 10-fold cross validation. Other than specification of training data, the options to this method include specifying the use of a binomial model (i.e., logistic regression) and specifying the use of classification error as the measure for selecting the best model during cross validation.

Listing 4.3: Training and Prediction with Logistic Regression and Lasso

```
# Load model from exported file
data <- readRDS(data.key)

# Run the lasso
cvfit = cv.glmnet(data$trainMatrix, data$trainY, family = "binomial", type.
  measure = "class")

# Run predictions, printing confusion matrix of t1 and t2 errors
confusion(predict(cvfit, newx = data$testMatrix, type = "class", s = c(cvfit$
  lambda.min)), data$testY)

# Run additional analysis (omitted)
```

The script then uses a `predict` function of `glmnet` to predict the *isInfeas* value given the features of the test data. Here, the `predict` call is wrapped in a method which produces a *confusion matrix* that includes the success and error rates of the classifier on both feasible and infeasible examples from the test data.

During the experiment, additional steps are included in this `predict` script to extract more information about the selected model. These steps are covered within the context of planned analysis and results in Chapter 5. The full text of the `predict` script is also included in Appendix B.

## Chapter 5: Experiments

In this Chapter, I describe the experiment carried out to address the research questions of this study. Recall the research questions and subsequent hypotheses as summarized in Table 1.2. I use the fully automated model-based testing workflow described in Chapter 3 and the infrastructure components from Chapter 4, applying these to generate and execute *Training* and *Test* test suites with GUITAR tools on four Java application subjects (Applications Under Test, or AUTs). I train a binary classifier for feasibility by applying logistic regression with lasso on the Training suite’s results, and apply the classifier to the Test suite to predict its results.

Throughout these steps, I gather metrics which directly address the research questions of the study. I analyze these metrics to evaluate each of my hypotheses. I provide details of the AUTs, specific steps of the methodology, planned analysis, and results in the sections below. I conclude with an overview of available experiment artifacts.

### 5.1 AUTs

The test subjects for this experiment are event-driven applications from a platform supported by the GUITAR framework. In particular, I chose to use the Java Foundation Classes (JFC) toolchain of GUITAR, which are the most common and most frequently used platform supported by the framework (and also, the platform most frequently used by existing empirical studies of model-based testing (MBT)).

Application	Version	Description
JabRef	2.10	Publication reference manager <a href="http://www.jabref.org/">http://www.jabref.org/</a>
ArgoUML	revision 19844	UML diagram creation <a href="http://argouml.tigris.org/">http://argouml.tigris.org/</a>
Buddi	3.4.1.11	Personal finance management <a href="http://buddi.digitalcave.ca/">http://buddi.digitalcave.ca/</a>
FreeMind	0.9.0	Mind-mapping diagram creation <a href="http://freemind.sourceforge.net/">http://freemind.sourceforge.net/</a>

Table 5.1: Experiment AUTs

For each of the AUTs, I require the following:

- **Java Environment:** The application should support execution on an OpenJDK JRE in a Linux OS (Ubuntu).
- **GUI Components:** The application should use Java Foundation Classes (JFC) for GUI components.
- **Graphics Environment:** The application should be displayable on the headless X11 display Xvfb.
- **GUIStructure Complexity:** A simple GUITAR Ripper configuration should be able to obtain a reasonably complex GUIStructure from the application, of at least 100 GUI events.

Table 5.1 summarizes the selected AUTs for the experiment. Each meets the listed requirements above, including the GUIStructure complexity requirement, which will be detailed in the “sanity check” results later in this Chapter. JabRef is a reference manager tool for researchers, which indexes and produces Bibtex and similar file formats. ArgoUML and FreeMind are both diagramming tools - ArgoUML for UML diagrams, and FreeMind for “mind maps.” Buddi is a personal finance tool for creating budgets and graphs.

Number	Step	Description
1	Generate Training Suite	Execute the GUITAR Ripper, Converter, and Generator tools, to construct Training suite (10,000 test cases from length=3 suite)
2	Generate Test Suite	Repeat Step 1, to construct Test suite (1,000 test cases, length=2 suite)
3	Execute Suites	Using the GUITAR Replayer, execute all test cases in the Training and Test suites 2 times each
4	Process Execution Results	Classify each test case as feasible, infeasible, or inconsistent based on results
5	Extract Features	Extract features from each test cases, and construct a combined set of features from the test cases in the Training suite
6	Prepare Data	Construct combined set of labeled training and test examples, including features and <i>is-Feas</i> values
7	Train Classifier and Predict	Train the feasibility classifier and classify test cases from Test suite

Table 5.2: Experiment Steps

## 5.2 Methodology

### 5.2.1 Overview

Given the goal of producing and evaluating a binary classifier for feasibility for each of the AUTs in the study, I develop the experimental procedure summarized in Table 5.2. Note that the steps here each leverage one or more fully automated jobs from the summary in Table 4.4.

### 5.2.2 Training and Test Suite Construction

In Steps 1 and 2, I generate two suites for training and testing of the classifier. Both suites are constructed using the Generator tool with the randomized adaptation of the

Sequence Length algorithm for test case generation described in Section 3.1.4. Following from the model-based testing workflow of GUITAR tools, the Ripper and Converter must be executed before the Generator in order to produce an EFG.

For the training suite, I sample 10,000 test cases from the sequence length coverage suite with length=3. Recall that the test cases in this suite will have a *minimum* length of 3, as a path to an initial event must be prepended to the length-3 sequence in each test case. Similarly, I sample 1,000 test cases for the Test suite from the sequence length coverage suite with length=2. During test case generation, in addition to the generation of test steps, test case features are also extracted. The extracted features are described in detail in Section 3.2.4 and summarized in Table 3.1.

In Step 3, I execute the test cases in each suite two times. I use the bundle abstraction provided by the TestData tool to group test case executions, and each execution stores its Replayer log as an artifact. This step leads to two bundles of test executions per suite. The multiple executions are a safeguard to protect against inconsistent execution of a test case for any reason.

In Step 4, I process the execution results of each suite, in order to classify each test case as feasible, infeasible, or inconsistent. I process the Replayer logs of each test case, and categorize each one as feasible or infeasible. If all test executions indicate feasibility, I classify the test case as feasible. If all executions indicate infeasibility, I classify the test case as infeasible. If test executions are inconsistent, the test case is classified as inconsistent (and removed from any further analysis). This step uses the `postResults` method of TestData described in Section 4.2.2.

In Step 5, I extract features from each test case to be used by the classifier. I also construct a combined set of features from all test cases in the Training suite, which will

be the only features known by the classifier. (Any features only appearing in test cases from the Test suite are ignored.) This step uses the `postFeatures` method of `TestData` described in Section 4.2.2 to extract the features detailed in Section 3.2.4.

### 5.2.3 Training and Prediction

In Step 6, I convert the artifacts stored during Steps 1-4 into an R object directly supporting remaining analysis using the `prepareData` script described in Section 4.4.1. Recall that this script outputs a binary file with data segregated as required into training and test partitions for use with `glmnet` in Step 7, given the features generated for each test case in the Training suite during Step 1, the results of execution of the training suite in Step 3, and the global set of features constructed in Step 5.

In Step 7, I use the `predict` script described in Section 4.4.2 to train the binary classifier for feasibility and predict the feasibility of test cases. Recall that the script uses functions from the `glmnet` package [42] to carry out logistic regression with the lasso regularization. I then use the trained classifier to predict the feasibility of test cases from the Test suite, which uses the `cv.glmnet` function from `glmnet` package to select the best classifier from multiple values of lambda. The classifier selection uses 10-fold cross validation on the classification error of the training data.

At this point, the core data collection of the experiment is complete. From here, I analyze details of the classifier's performance and the characteristics of the underlying model to address the hypotheses posed for each of the research questions presented in Section 1.4 (Table 1.2).

<b>ID</b>	<b>Metric</b>	<b>Description</b>
SANA	Event Count	Number of nodes (events) in the EFG
SANB	Initial Event Count	Number of initially available events in the EFG
SANC	Follow Count	Number of edges ( <i>may-follow</i> relationships) in the EFG
SAND	Full SL2 Size	Number of test cases in the full sequence length coverage suite (length=2)
SANE	Full SL3 Size	Number of test cases in the full sequence length coverage suite (length=3)
SANF	Feature Count	Number of features in training data
SANG	Inconsistent Count - Training	Number of Training test cases removed due to inconsistent results
SANH	Inconsistent Count - Test	Number of Test test cases removed due to inconsistent results
SANI	Feature Frequencies Histogram	Distribution of feature counts in training data

Table 5.3: Sanity Checking Metrics Summary

## 5.3 Planned Analysis

### 5.3.1 Overview

To address the three research questions of this study, I develop a set of metrics for evaluating each research question. I also include a set of “sanity check” metrics to ensure sufficient complexity of the models and other artifacts obtained from each AUT’s model-based workflow steps. Table 5.3 contains a summary of the sanity check metrics, and Table 5.4 a summary of metrics directly related to research questions. I discuss each metric in detail in the sections below, including expectations for each, as relevant. I collect each metric from each AUT in the experiment, and analyze trends across AUTs to answer each research question.

<b>ID</b>	<b>Metric</b>	<b>Description</b>
RQ1A	Feasible Count - Training	Number of Training test cases considered feasible
RQ1B	Infeasible Count - Training	Number of Training test cases considered infeasible
RQ1C	Feasible Count - Test	Number of Test test cases considered feasible
RQ1D	Infeasible Count - Test	Number of Test test cases considered infeasible
RQ2A	Lambda Curve	Plot of lambda values vs. classification error on Training data
RQ2B	Selected Variable Count	Number of features selected by the model
RQ2C	Overall Error Rate	Percentage of test cases from Test suite classified incorrectly
RQ2D	False Positive Rate	Percentage of feasible test cases from Test suite classified as infeasible
RQ2E	False Negative Rate	Percentage of infeasible test cases from Test suite classified as feasible
RQ3A	Selected Variable Counts by Category	Number of variables selected by the model from each category of features
RQ3B	Selected Variable Magnitudes by Category	Magnitude of coefficients selected by the model from each category of features

Table 5.4: Research Question Metrics Summary

### 5.3.2 Sanity Checks

Before proceeding with any analysis more directly related with the research questions of the study, I start by collecting a set of sanity check metrics. These metrics indicate whether the model-based testing workflow for each AUT obtained a reasonably complex model. Because each AUT used in this study has been provided with a basic configuration for the Ripper, I do not expect any issues with construction of a trivial model. However, the complexity of the applications and their model-based testing suites is also relevant for the generalizability of results. I provide seven such metrics.

From the EFG, I first obtain an event count (SANA) by counting the number of nodes in the graph. Relatedly, I capture the number of initial events in the EFG (SANB), and the number of edges in the EFG (SANC). I expect these metrics to indicate that the EFG of each AUT is sufficiently complex, with 100s of nodes and edges, and at least a dozen initial events.

I then check the size of the sequence-length coverage suites of length=2 (SAND) and length=3 (SANE) respectively, from which test and training suites are sampled, to ensure that there is sufficient diversity in the randomly selected test cases drawn from these suites. I expect that the sequence length suites for length=2 (SL2) sizes will be in the tens of thousands ( $> 10K$ ) and the length=3 (SL3) sizes in the hundreds of thousands ( $>100K$ ).

From the training data, I obtain two metrics to characterize the extracted features. The overall feature count (SANF) indicates the total number of features extracted across all test cases in the Training suite. I also generate a histogram of feature counts (SANI) in the training data, to characterize the distribution of feature counts. I expect these

numbers to vary, as they depend on the structure of the randomly selected test cases. I expect to see numbers at least in the thousands ( $> 1K$ ) for feature counts, and for the histogram to indicate that at least some reasonable percentage of features (e.g., 20%) occur more than once in the training data.

Finally, I check the number of test cases from both the Training (SANG) and Test (SANH) suites whose execution was inconsistent. I expect this number to be less than 10% of the total size of the sampled suites ( $< 1K$  test cases for the Training suite,  $< 100$  test cases for the Test Suite). Recall that test case inconsistency is not the same as test case infeasibility. Inconsistent test case execution indicates intermittent or systematic issues with infrastructure, or possibly an issue with Replayer timing configuration (e.g., Replayer steps counting as “timed out” or being missed for some reason).

### 5.3.3 RQ1: Prevalence

Recall research question RQ1, and my hypothesis:

- RQ1: How prevalent are infeasible test cases in MBT workflows?
- Hypothesis: Infeasible test cases will occur in all MBT test suites, at a rate of up to 20%.

To evaluate this hypothesis, I collect metrics for the occurrence of feasible and infeasible test cases in both the Training (RQ1A and RQ1B, for feasible and infeasible counts, respectively) and Test (RQ1C, RQ1D) suites. The feasible counts are included for reference, as they will also be relevant for the computation and interpretation of additional metrics for other research questions. I expect the infeasible counts for each test suite to be  $> 1$ , and as high as 20% of the Training and Test suite sizes, which have theoretical limits of 10,000 and 1,000 respectively (minus any inconsistent test executions which have

been removed).

#### 5.3.4 RQ2: Binary Classifier

Recall research question RQ2, and my hypothesis:

- RQ2: Can an application-specific model of test case feasibility be used to predict the feasibility of unseen model-based test cases?
- Hypothesis: A binary classifier can be constructed which will outperform at least a random (50%) baseline in predicting the feasibility of unseen test cases.

To evaluate this hypothesis, I collect two metrics characterizing model fit, and three indicating classifier accuracy. First, I obtain a plot of  $\lambda$  values (RQ2A) characterizing the chosen “best” model from among other model choices. Recall that training with lasso provides multiple possible models (one for each value of the metaparameter  $\lambda$  considered), and I use cross validation to choose the best model. A plot of lambda values vs. classification error on the training data characterizes the relative fit of the best model as an alternative from all others. I expect to see the chosen value of  $\lambda$  to have a global minimum value for training classification error. I also expect to see that model performance “ramps up” (potentially rapidly) for larger values of  $\lambda$ , and holds steady for smaller values.

Next, I get a count of the number of variables selected by each model (RQ2B). Recall that the regularization parameter for lasso takes an  $\ell_1$  norm, which allows unused variables from the model to have their coefficients fit to 0. I expect this value to be much less than the total number of features in the training set (SAND), which would indicate that the lasso actually achieved its desired effect of selecting variables.

To evaluate the accuracy of the classifier, I use the popular metrics of overall error rate (RQ2C), false positive rate (RQ2D) and false negative rate (RQ2E). Because the

classifier is trained to predict the value *isInfeas*, a false positive indicates a test case which was predicted as infeasible, but was actually feasible. A false negative, then is the other error case: a test case which was predicted as feasible, but was actually infeasible. Because this research is the first known investigation of feasibility classification, I enforce only a random baseline for performance. To consider the classifier successful, I expect all three error rates to be less than 50%.

The ratio of false positive rate to false negative rate is also interesting to consider, as false negatives (i.e., accidentally letting an infeasible test case go undetected) could be considered more damaging than false positives. However, as I discuss when motivating the need for feasibility modeling in Section 1.3.3, test execution also has a potentially prohibitive cost. I argue that interpretation of “preferred” errors for this classifier depends heavily on the broader use cases. Since no specific broader use cases are in-scope for this initial evaluation, I have no *a priori* requirements for the ratio of false positives to false negatives in order for the classifier to be considered effective.

### 5.3.5 RQ3: Important Features

Recall research question RQ3, and my hypotheses:

- RQ3: Which of the considered feature types of model-based test cases contribute the most to feasibility prediction?
- Hypothesis A: Test case before-pairs and n-grams of length=2 will contribute most often and most significantly to feasibility models.
- Hypothesis B: Test case before-pairs will contribute more significantly to feasibility models than n-grams.

As discussed in Section 3.2.4 (and summarized in Table 3.1), I use four high-level

categories of features for the training of the binary classifier: n-grams and before-pairs of both event IDs and event types. For the purposes of addressing RQ3, I divide the n-gram categories into “length=1” and “length=2” subdivisions, as shown in Table 5.5.

Feature Category	Category ID	Regular Expression
Before Pairs of Event IDs	<code>before_id</code>	<code>^[A-z0-9]+_before_</code>
Before Pairs of Event Types	<code>before_type</code>	<code>^[A-z0-9 ]+_before_</code>
n-grams of Event IDs, length=1	<code>ngram_1_id</code>	<code>^ngram_[A-Za-z0-9]+\$</code>
n-grams of Event IDs, length=2	<code>ngram_2_id</code>	<code>^ngram_[A-Za-z0-9]+_</code>
n-grams of Event Types, length=1	<code>ngram_1_type</code>	<code>^[ngram_[A-Z ]+'[01]]\$</code>
n-grams of Event Types, length=2	<code>ngram_2_type</code>	<code>^[ngram_[A-Z ]+'[01]]\$</code>

Table 5.5: RQ3 Feature Categories and Regular Expressions for Analysis

I expect before-pairs of either type of feature (event IDs or event types) to be the most effective category of features types among the types considered. To evaluate this hypothesis, I compare the number and magnitude of coefficients from each category in the model of each AUT’s feasibility classifier. To compare magnitude, I look at cumulative magnitude in particular, and sum the absolute value of all coefficients in each category.

The inclusion of separate classes in this analysis to distinguish between length=1 and length=2 n-grams provides another interesting comparison point. In motivating the need for a feasibility classifier throughout Chapters 1 and 2, I assert that “event context” is closely related to feasibility, and that infeasible test cases are often made infeasible by missing context. n-grams of length 1 have very little context available: they only consider that an event or event type occurred anywhere in a test case, without any knowledge of events occurring before or after. n-grams of length=2, however, assume a strict context. before-pairs capture even more context by allowing a less-strict ordering of events or types. I also expect before-pairs of either type of feature to contribute more to the classifier

than n-grams, because they are “loosely” instead of strictly ordered. Importantly, there is overlap between before-pairs and n-grams, in that each n-gram has a corresponding before-pair. I expect the before-pair to be preferred by the classifier.

The final column in Table 5.5 shows the regular expression used to extract feature occurrences from the list of features selected by each model. These expressions match the coding of features during feature extraction by the `FeaturesProcessor` and `postResults` methods described in Chapter 4. The allowance for a 0 or 1 on the end of each feature name follows from the allowance of two levels of each feature in the classifier, and is consistent with the R `model.matrix` function’s output feature names.

## 5.4 Results and Analysis

Given the planned analysis for each research question through the metrics and expectations described above, below are the results of each metric after applying the steps in the Methodology presented in Section 5.2. Findings related to each research question are summarized in each section below. Chapter 6 summarizes all findings from the study.

### 5.4.1 Sanity Checks

Table 5.6 contains the sanity check metrics for each AUT in the study, with the exception of SANI. Figure 5.1 displays SANI, the feature count distributions of each AUT, as histograms.

#### 5.4.1.1 EFG Complexity

The EFG metrics of event count (SANA), initial event count (SANB), and edge count (SANC) all indicate a reasonable level of complexity for the EFG models obtained

AUT	SANA	SANB	SANC	SAND	SANE	SANF	SANG	SANH
JabRef	662	165	646	102102	23669577	45952	709	0
ArgoUML	480	219	515	15867	650425	23739	0	46
Buddi	239	42	407	3702	94757	8815	54	0
FreeMind	648	419	586	47127	4868853	37058	2907	0

Table 5.6: Sanity Check Metrics (all but SANI)

for each AUT. Events and edges number in the 100s for each application. While the Buddi application only identified 42 initial events in its EFG, these still represent a reasonable percentage of the 239 events in the overall EFG (17.5%).

The sequence length suite size metrics also speak to the complexity of the EFG of each AUT. For the length=2 suites (SAND), only Buddi has a full suite size of less than 15,000 test cases. The sizes of the full length=3 suites (SANE) further indicate the complexity of the AUTs. These suite sizes are in the millions for the more complex JabRef and FreeMind applications, and at nearly 100K (94,757 test cases) even for the much simpler Buddi application.

The metrics for sequence-length coverage suite size in general echo my original motivation for fully automated, predictive consideration of feasibility. The complexity of the length=3 suite also suggests that sampling a reasonable number of test cases from this space (such as the 10,000 test cases I sample for construction of the Training suite of each AUT in this study) provides a correspondingly broad feature space for use as training data.

I conclude that the AUTs chosen for this study, and the Ripper configurations used for their model construction, have led to sufficiently complex input spaces for the purposes of this experiment. The particular metrics of each AUT’s model complexity will also be relevant in additional considerations throughout the experiment.

### 5.4.1.2 Training Data Complexity

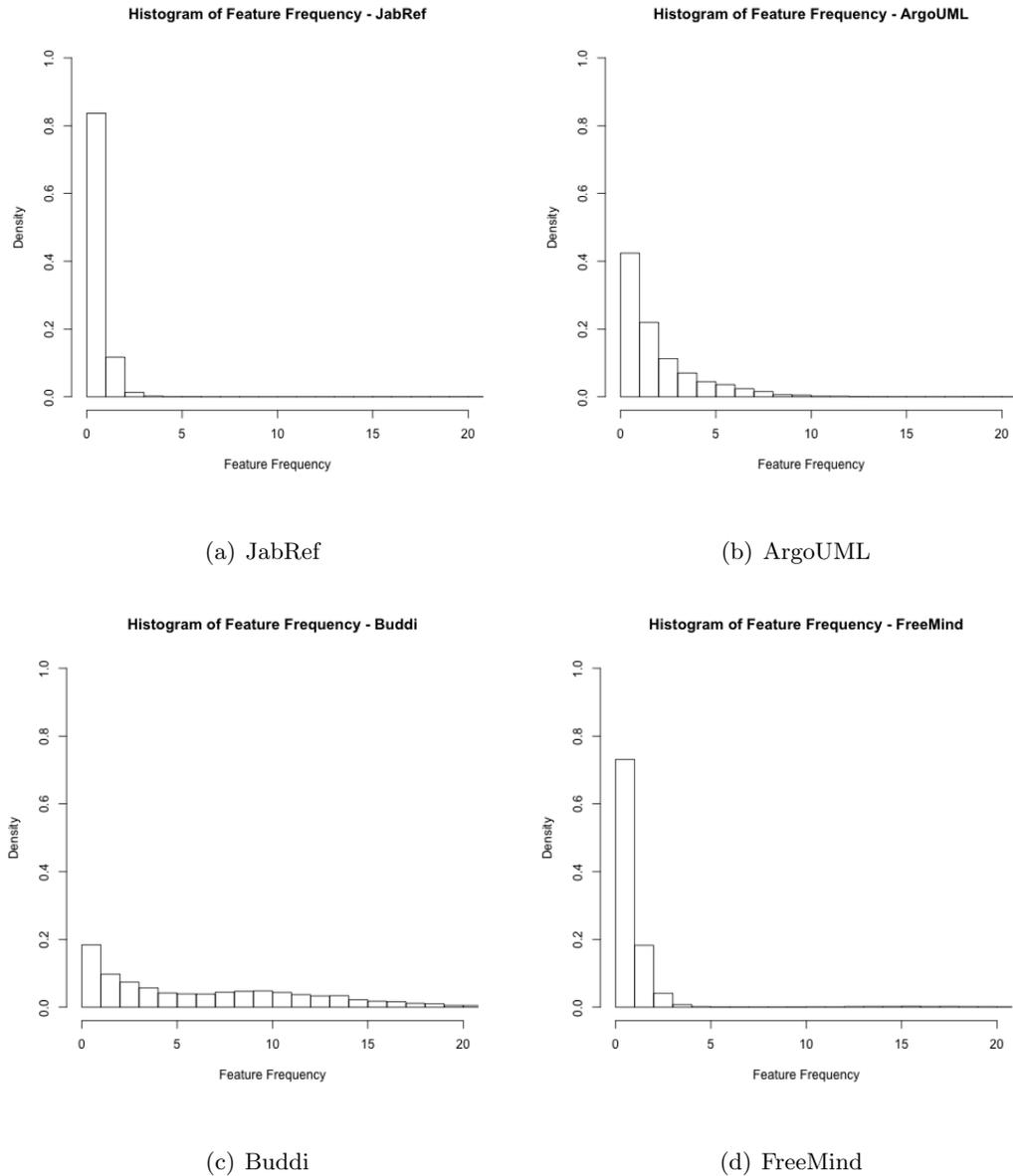


Figure 5.1: AUT Feature Frequencies

Metrics SANF and SANG in Table 5.6 further detail the Training data ultimately used by each AUT in the experiment. Each AUT starts with several thousand features available for the training phase. Note that this metric is distinct from the number of features *selected* by each model during training, which is given by metric RQ2B and

discussed during consideration of RQ2 below.

For purposes of sanity checking, a basis of several thousand possible features seems appropriate. The histograms shown in Figure 5.1 give a more complete picture of the available features. These plots show, for each AUT, the percentage of features which have a given number of occurrences in training data. I have kept the X and Y axes of these plots consistent, for comparison purposes.

Each bar of the histogram indicates the percentage of features who have counts “falling between” the counts on the X axis. Since the X axis is counting by 1s, each bar represents the number to its right (i.e., the first bar indicates features having a count of 1, the second bar the percentage of features having a count of 2, and so on).

The plots show that the two more complex AUTs (JabRef and FreeMind) have much larger percentages of features which occur only once in the training data. Additionally, both have very few features which occur more than 4 times. However, recall that these AUTs also have considerably higher overall number of features compared to ArgoUML and Buddi (Buddi only has roughly 8800 features, compared to JabRef’s >45,000).

Interestingly, the Buddi application, though much less complex in its EFG model and feature space, has a much more diverse distribution of feature occurrences. The distribution for ArgoUML falls between that of Buddi and those of JabRef and FreeMind in terms of this diversity.

Overall, I conclude that the feature spaces used in the Training suite of each AUT are once again sufficiently complex and diverse for this analysis. Even the less-diverse distributions have 100s of features with counts of 2 or higher.

However, the prevalence of features with lower counts (especially to the extent observed on JabRef and FreeMind) does lead to noise and potential overfitting during model

training. For example, no feature with a count of 1 should ultimately be selected by any classifier; but I do nothing in the current experiment to control for this possibility. For example, I do not explicitly throw out any features due to having low counts. I count on the regularized model to avoid selection of these features.

#### 5.4.1.3 Inconsistent Test Cases

Returning to the metrics from Table 5.6, the last two sanity check metrics indicate the prevalence of inconsistent test case execution in the Training (SANG) and Test (SANH) suites. The JabRef and FreeMind applications indicated potential issues with inconsistent execution of test cases from their Training suites, with JabRef having 709 out of 10,000 test cases behave inconsistently, and FreeMind having 2907 out of 10,000. While the JabRef percentage is within the “up to 10%” I assumed was reasonable, the nearly 30% of FreeMind Training test cases having inconsistent execution does indicate the possibility of an issue with Replayer or infrastructure configurations for FreeMind, or that an intermittent infrastructure issues (e.g., I/O or resource allocation issues) caused an error during one of the two bundle executions for FreeMind training test cases.

No AUTs exhibited prevalent issues with Test suite executions, with only ArgoUML having a small number of tests executing inconsistently (46).

For the purposes of the current experiment, the large number of inconsistent executions for the FreeMind Training suite mean that the Training suite will only have the remaining 7,093 test cases available for training. This smaller training suite could potentially have negative effects on the overall performance of a classifier for FreeMind. Ultimately, the performance of the FreeMind classifier was exceptional, as discussed below. Because Replayer and infrastructure configuration considerations are secondary to

<b>AUT</b>	<b>RQ1A</b>	<b>RQ1B</b>	<b>RQ1C</b>	<b>RQ1D</b>
JabRef	6307	2984 (32.1%)	705	295 (29.5%)
ArgoUML	4653	5347 (53.5%)	610	344 (36.1%)
Buddi	8977	969 (9.7%)	813	187 (18.7%)
FreeMind	5763	1330 (18.8%)	922	78 (7.8%)

Table 5.7: RQ1 Metrics

the classifier experiments in this study, I have not investigated the nature of the FreeMind or JabRef failures encountered any further as a part of this study. However, I do archive all artifacts (including Replayer and infrastructure configurations, and all Replayer artifacts) for consideration of these types of failures in future work.

#### 5.4.2 RQ1: Prevalence

For RQ1, I look to gather evidence of the prevalence of infeasibility in model-based test suites (and the sequence-length coverage suites used in this experiment and many prior studies, in particular). Table 5.7 shows the metrics gathered to address this research question, including the number of infeasible test cases as a percentage of the total number of test cases in the Training (RQ1B) and Test (RQ1D) suites.

The sampled test suites exhibited between the wide range of 7.8% and 53.5% infeasible test cases. As expected, the actual number of infeasible test cases depends on the nature of the AUT and its acquired model. I conclude that these results confirm my hypothesis that infeasibility is prevalent enough in model-based test suites such as sequence length coverage suites to warrant approaches such as the feasibility prediction I design and evaluate in this study.

AUT	RQ2B	RQ2C	RQ2D	RQ2E
JabRef	392	0.029	0.040	0.003
ArgoUML	614	0.046	0.003	0.122
Buddi	566	0.012	0.004	0.048
FreeMind	174	0.037	0.040	0.000

Table 5.8: RQ2 Metrics (all but RQ2A)

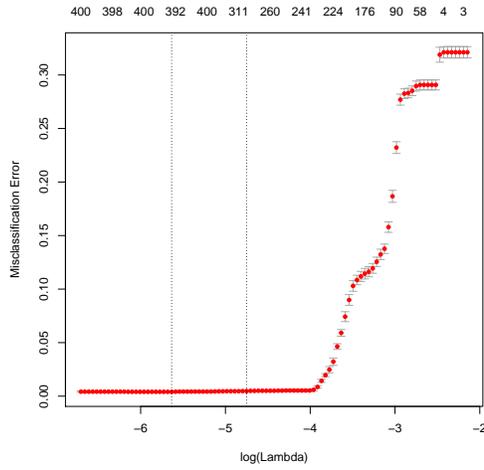
### 5.4.3 RQ2: Binary Classifier

For RQ2, I look to consider the general ability to train an accurate application-specific classifier for test case feasibility for model-based test suites, given the modeling approach detailed in Chapter 3. As planned, I evaluate the models in terms of fit and accuracy. While accuracy is certainly the most important measure of the model’s performance, its ability to fit as intended is also an important consideration for generalizability.

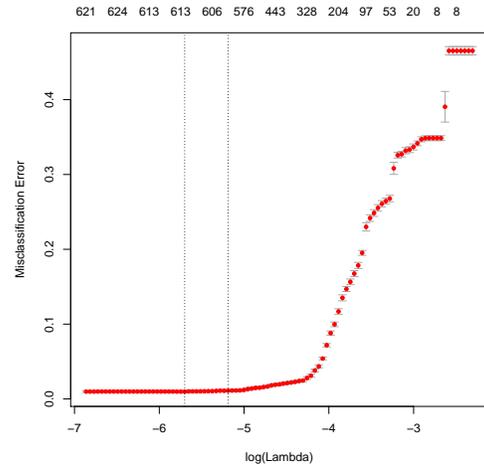
Table 5.8 shows the metrics gathered for investigating RQ2, with the exception of RQ2A, which is shown as a grid of AUT-specific plots in Figure 5.2.

#### 5.4.3.1 Model Fit

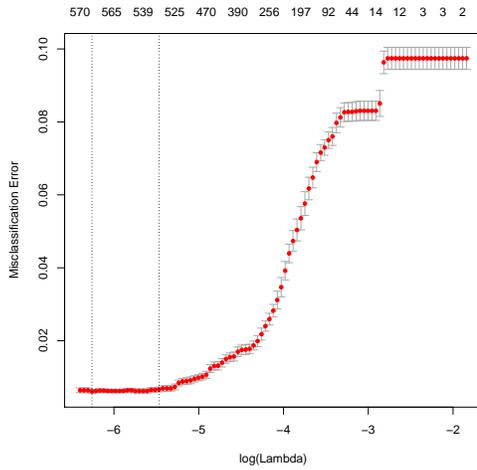
I first consider metric RQ2B, which shows the number of variables ultimately selected by the logistic regression procedure with the lasso term. Of the 1,000s and 10,000s of features available, the models are selecting between 174 and 614 features. Recall from the implementation discussed in Chapter 4 that the model actually has two levels of each feature available, corresponding to the presence or absence of each feature independently (i.e., the actual maximum number of features is twice the number of features explicitly provided in the training data). At a high level, this shrinkage of the number of variables considered by the model suggests that the lasso term is accomplishing its goals of shrinkage



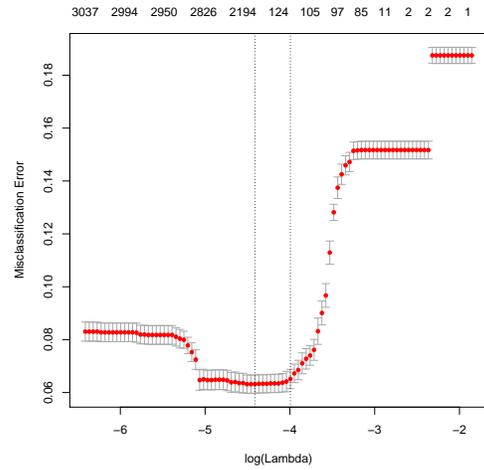
(a) JabRef



(b) ArgoUML



(c) Buddi



(d) FreeMind

Figure 5.2: RQ2A: Training Misclassification as Function of Lambda and Number of Variables and selection.

While the number of variables selected indicates that at least some shrinkage is happening, it does not say whether an appropriate amount of shrinkage is happening. The RQ2A plots in Figure 5.2, provided by `glmnet`, show the classification error on the training data for each value of  $\lambda$  considered when fitting the model. (Because I use cross

validation to fit the model, this classification error is an average over the 10 cross-validated subsets.)

Note that the X axis along the bottom of each plot indicates the log of  $\lambda$ , which increases as we move from left to right. Due to the construction of the minimization problem discussed in detail in Section 3.2, larger values of  $\lambda$  lead to a smaller number of coefficients. A second X axis along the top of each plot shows the number of non-zero coefficients in the model corresponding to each value of  $\lambda$ .

There are also two vertical dashed lines on each plot. The leftmost of these lines indicates the value of  $\lambda$  which achieves the minimum classification error on the training data. The rightmost line indicates a value within one standard error of the minimum training error. As I note in Section 3.2, I use the minimum value of  $\lambda$  for the classifiers in this study.

For the purposes of evaluating RQ2, I interpret the plots of Figure 5.2 as indicating that the selected values of  $\lambda$  are indeed global minima from the range of values considered. The plots also indicate, for each AUT, that moving to larger values of  $\lambda$  stands to increase the classification error; and that performance significantly degrades at much larger values of  $\lambda$ .

For the fit of models as a whole, I conclude from RQ2A and RQ2B that the logistic regression with lasso successfully converged a regularized model for the feasibility classifier. This evaluation of model convergence behavior also implies that my implementation of the supervised learning scenario and the use of `glmnet` for model fitting as a whole are reasonable.

#### 5.4.3.2 Classifier Performance

As evaluated so far, the trained model may still be significantly *overfitting*, though the use of cross-validation should counter this tendency as much as the training data allow. Metrics RQ2C, RQ2D, and RQ2E evaluate the feasibility classifier’s performance on completely unseen data, summarized in Table 5.8.

As an aside, this research (as discussed since the original problem formulation in Chapter 1) is not concerned with construction of cross-application models of feasibility. By overfitting, I mean that the model could potentially miss correlations present in the Test suite (or in general, any collection of test cases from the same application). I do not mean, at least within the context of this study, to protect against overfitting the features of a single AUT.

For RQ2, I originally hypothesized that a binary classifier for feasibility should at least be able to outperform a random “50/50” model. As each of the metrics for RQ2 indicate, the classifiers for the AUTs in this study actually do much better. Overall error rates, as well as false positive rates for each AUT, are at 5% or less on the Test suite considered for this study. False negative rates for all AUTs other than ArgoUML are also under 5%.

In the case of ArgoUML, the classifier predicts several false negatives, which are potentially troubling. False negatives could occur due to:

- Test case features which are causing infeasibility in the Test suite which are not present in the Training suite.
- Test case features which are causing infeasibility in the Test suite which are present in the Training suite, but are not correlated with infeasibility in the Training suite.

- Test case features which are causing infeasibility in the Test suite are correlated with infeasibility in the Training suite, but are not selected by the model.

Given the overall very good performance of the classifier, I consider the first two explanations above to be the most likely. Especially for event ID-related features, only events which are seen (and seen multiple times) in the training data will have the chance to be considered for inclusion in the model. This finding suggests that the use of more generic features (i.e., features likely to occur more often yet still related to event context and test case feasibility) would improve the general effectiveness of the classifier. Also, this evaluation could be strengthened by considering multiple randomly selected Test and Training suites rather than just one. I elaborate on this and other threats to validity from the experimental constructs in Chapter 6.

Even with the occurrence of a higher number of false negatives on one AUT, the overall performance of the classifier is very promising. I conclude that my proposed feasibility classifier effectively identified infeasible test cases, and it did so with overall error rates of less than 5% for each AUT considered.

#### 5.4.4 RQ3: Important Features

To address RQ3, I consider both the types (RQ3A) and magnitudes (RQ3B) of features selected by the classifier during training. Recall from the discussion of planned analysis for RQ3 above that by “magnitude,” I am measuring the cumulative magnitude of all model coefficients in each category of features.

Figure 5.3 shows the counts of each category of features, by AUT. Figure 5.4 shows the corresponding cumulative magnitude of model coefficients of each category of features, by AUT.

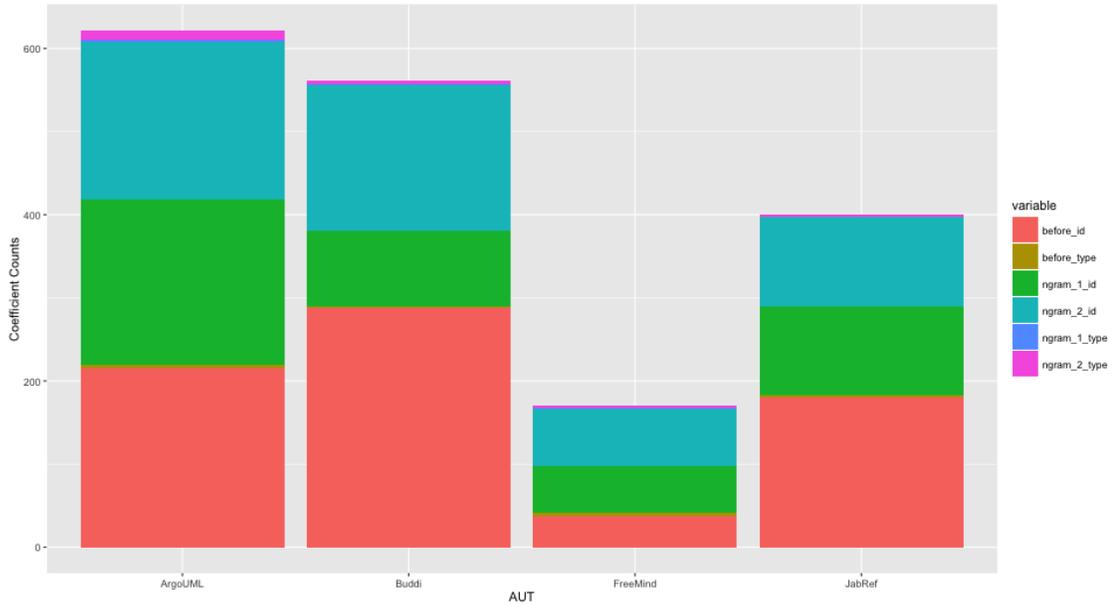


Figure 5.3: RQ3A: Coefficient Counts by AUT and Variable Category

#### 5.4.4.1 Event Types vs. Event IDs

Immediately from Figure 5.3, notice that the event type features are very rarely selected by the regularized model. The distinction of before-pairs vs. n-grams of these types also does not seem to make a difference. Event type features are selected only 39 times total, out of the several thousand possible features of each AUT’s model. Corresponding data in Figure 5.4 confirm that the cumulative magnitudes of event type coefficients likewise account for less than 1% of the total for each AUT.

Clearly, these findings suggest that the EFG’s event types - in isolation or in shorter-length contexts as considered in this study - are not closely correlated with infeasible test cases. I interpret this finding to also confirm that the Converter tool’s algorithm for inferring *may-follow* relations from the `GUIstructure` output of the Ripper is not a cause of infeasibility. Stated another way, test case infeasibility as observed in this study does

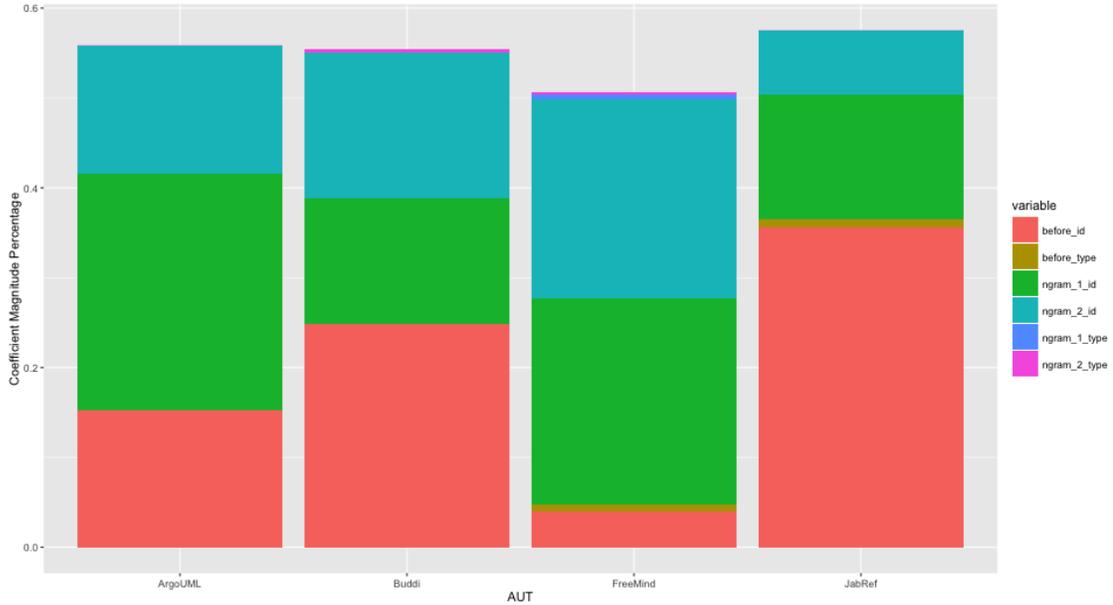


Figure 5.4: RQ3B: Cumulative Magnitudes by AUT and Variable Category

not appear to stem from any improper conversion of `GUIStructure` input to EFG by the Converter.

Features related to event IDs, then, take up the remaining vast majority of selected features and coefficient magnitudes in each AUT’s model.

#### 5.4.4.2 Most Frequently Selected and Highest Cumulative Magnitude Feature Type

As the bars in Figure 5.3 indicate, before-pairs of event IDs are the most frequently selected category of features in feasibility models. The only exception to this finding among the four AUTs is FreeMind, which had more n-grams of event IDs with length=2 selected than any other category of features. When considered as a single category, the N-gram category of features as a whole (regardless of length) is selected more often than before-pairs. The only exception to this among the four AUTs is Buddi, which still had more before-pairs selected.

In terms of coefficient magnitude, the results are a bit more mixed. For JabRef and Buddi, before-pairs of event IDs contribute the largest cumulative magnitude to the model (at 36% and 25%, respectively). For FreeMind and ArgoUML, n-grams of length 1 account for the largest cumulative magnitude (at 26% and 23%, respectively).

From these results, I cannot conclusively accept either of the two hypotheses suggested for RQ3. The evidence of a higher number of features from a category being selected does not support an interpretation of those features as “effective” on its own. These findings suggest that for some applications, test case infeasibility is highly correlated with the occurrence of a single event. Perhaps this single event blocks many (or all) potential subsequent events from executing, and this blocking is not detected by the Ripper during the creation of the `GUIStructure`.

One possible explanation for these results could be that counts and magnitudes for n-grams outnumber and outweigh those of before-pairs due to n-grams occurring more often. However, before-pairs actually occur more frequently in a typical test case than either class of n-grams. Consider a test case of four events:  $e1 \rightarrow e2 \rightarrow e3 \rightarrow e4$ . There would be 4 n-grams of length 1 for this test case, with one for each event. There would also be 4 n-grams of length=2:  $START \rightarrow e1$ ,  $e1 \rightarrow e2$ ,  $e2 \rightarrow e3$ , and  $e3 \rightarrow e4$ . But there would be 6 before-pairs:  $(e1, e2)$ ,  $(e1, e3)$ ,  $(e1, e4)$ ,  $(e2, e3)$ ,  $(e2, e4)$ , and  $(e3, e4)$ .

These results suggest a possible masking effect between before-pairs and n-grams of length=2. Notice from the example above that some before-pair features will always occur with a corresponding n-gram (e.g., the  $(e1, e2)$  before pair and the  $e1 \rightarrow e2$  n-gram). Lacking any additional information, the feasibility model could potentially select the n-gram to explain any corresponding infeasibility, when the before pair may be an equivalent (and more general) choice. In general, this finding proposes an interesting

follow-up question for future work: Can models trained solely on before-pairs of event IDs equal the performance of those trained with just n-grams, or those trained with a mix of before-pairs and n-grams as in this study?

## 5.5 Published Artifacts

In the interest of transparency and the repeatability of my experiments, I publish a number of experimental artifacts. As explained in detail in Chapter 4, I go to great lengths throughout this research to focus on scalable infrastructure and tooling. I hope that by making these tools and artifacts available, I can help to advance the model-based testing community of researchers and practitioners, and perhaps broader communities (where applicable).

Available artifacts include:

- **docker-jenkins-swarm**: This repo, available on Github at <http://github.com/bryantrobbins/docker-jenkins-swarm>, contains the Docker configurations used in my experimental infrastructure. This code is designed to be reusable for empirical studies of testing or other fields where a distributed execution of tasks is required. These tools require a Linux host or VM running Docker.
- **guitar2**: This repo, available at <http://github.com/bryantrobbins/guitar2>, contains my Maven-ized fork of GUITAR's Java tools, as well as the TestData tool for artifact persistence. The GUITAR toolchain should be compatible with any JFC application. The TestDataTool can be extended for the persistence of arbitrary binary or JSON-serialized artifacts associated with test cases, test suites, and their executions.
- **umd-dozen**: This repo, available at <http://github.com/bryantrobbins/umd-dozen>,

contains GUITAR tool configurations for each AUT used in my experiments, as well as an extensive set of Gradle and Bash wrapper scripts for more direct integration with automated jobs.

- **Master Archive:** This final artifact is a tarball of data volumes which, when used in combination with the docker-jenkins-swarm generic scripts, will restore a set of Master Docker containers (e.g., Jenkins Master, MongoDB, and Maven Repo) with the exact data collected and used in my experiments. Publication of this archive is pending. Please contact the author for access.

## Chapter 6: Conclusions and Future Work

In this chapter, I summarize the findings of Chapter 5 within the broader context of the research questions. I also summarize a number of understood threats to the validity of this research. I conclude with thoughts on future directions for this work.

### 6.1 Outcomes

In this research, I have developed a novel technique for the detection of infeasible test cases. I first investigate the prevalence of infeasible test cases in four Java desktop applications (RQ1). I then construct and evaluate a binary classifier for feasibility’s ability to detect infeasible test cases from a randomly sampled suite of model-based test cases designed to meet a sequence-length coverage criterion (RQ2). Finally, I inspect the models learned by the classifier for insights into the most effective predictors of infeasibility.

#### 6.1.1 RQ1: Prevalence

Regarding the overall prevalence of infeasible test cases in model-based test suites, I conclude that:

- O1: Infeasibility in model-based test suites is a significant problem, with infeasible test cases generated at rates of 7.8% or higher for every test suite I considered, and as high as 53.5%.
- O2: The tendency to generate infeasible test cases varies widely across application.

These findings confirm those observed from previous studies such as Nguyen and Memon [17] and Bae, et al. [38].

### 6.1.2 RQ2: Classification

Regarding the ability of feasibility prediction to be learned by a binary classifier, I conclude that:

- O3: Binary classifiers of feasibility with overall error, false positive, and false negative rates under 5% can be constructed by observing the executions of randomly sampled model-based test cases.

To my knowledge, this study represents the first of its kind in its approach to detecting feasibility. My approach contrasts with the fully observed paradigms of Nguyen and Memon [17], and Huang, et al. [18]. However, I focus on only the detection of feasibility while their existing techniques also consider the generation of feasible test cases from infeasible.

### 6.1.3 RQ3: Classifier Features

Regarding the relative importance of categories of the considered classifier features, I conclude that:

- O4: The event types of the the GUITAR Ripper’s `GUIStructure` output artifact do not correlate with infeasibility, confirming that initial model imprecision (rather than the inferences of the Converter and Test Case Generation algorithms) is the dominant source of infeasibility in model-based testing workflows.
- O5: Event IDs formed by creating hashes of GUI properties during model construction served as effective features for the binary classifiers constructed in this study.

- O6: I identified three types of features formed from event IDs that serve as effective inputs for the feasibility classifier: loosely ordered before-pairs of event IDs, and n-grams of event IDs of length=1 and length=2.
- O7: While all three of the feature types identified in O6 contributed to feasibility classifiers, no single type of the three appeared unilaterally more effective across the four AUTs considered in my experiments.

This study applies the earlier assumptions of work on test case feasibility from Memon [46], Huang et al. [18], and Nguyen and Memon [17] to the construction of a binary classifier. I have shown, for the first time, that the same features captured in Memon’s EFG model of GUIs for model-based testing can, at least in this use case, be effectively represented within the much more general class of linear models trained by logistic regression.

#### 6.1.4 Additional Outcomes

In addition to the experimental outcomes above, this research effort as a whole also contributes the following:

- A general framework for the serialization and persistence of arbitrary experimental artifacts, through the TestData tool.
- A general approach for the extraction and persistence of arbitrary test case features for classifier construction.
- A scalable approach for isolated test execution using state-of-the-art container and continuous integration technologies.

As mentioned from Chapter 1 and detailed in Chapter 4, I placed a very large emphasis on tool and infrastructure support throughout this research. As a result, every

outcome I have gleaned from experiments is repeatable on a wide range of hardware configurations (i.e., Linux hosts running Docker). Of the tools and infrastructure components I developed, very few are platform or AUT specific, and all are easily extended for a wide range of research and practical applications.

## 6.2 Threats to Validity

As with any experimental outcomes, the outcomes of my experiments in this study are subject to both internal and external threats to validity. In this section, I discuss what I consider to be the biggest threats to internal and external validity.

### 6.2.1 Threats to Internal Validity

In my studies, I am assuming that GUI tests are repeatable, given the same input and AUT configurations. As the inconsistent test case executions indicate, there are scenarios when general “flakiness” in application and testing tool configurations can cause inconsistent results. I chose to throw out altogether any test cases which exhibited inconsistencies across two separate executions as one way of addressing this threat.

In forming Training and Test suites for my experiments, I assume that the selection of test cases from a much larger available pool is truly random, and that the randomly sampled suites are representative of the broader population of model-based test suites from which they are drawn. During “sanity checking,” I confirmed that the pools from which each suite is sampled are of very different sizes (e.g., with the training pool 1-2 orders of magnitude larger than the test pool for most AUTs). I also manually confirmed that the random sampling logic in the Java code constructing the randomly sampled suites is using an appropriate approach.

The selection of AUTs is also a possible source of bias in my experiments. I chose to use the Java JFC platform, which does not have a very large pool of robust applications available, in comparison to the more contemporary Web and mobile platforms. However, the Java platform of GUITAR has been used as the basis for many studies with GUITAR and model-based testing in general, and the Java tools are the most robust of those in the framework. Also, I assert that the choice of platform has little effect on the outcomes of this study: a model such as the feasibility classifiers I construct is platform-independent, though the specifics of features (e.g., the algorithm for computing a unique event ID) may vary across platforms. Additionally, I consider the risk of using newer, less established tools to be much greater than that of using older, less familiar AUTs.

Finally, as discussed in Chapter 3 regarding model construction, I chose not to spend a great deal of time on the customization of GUITAR tool configurations for this study. It is possible that more thorough configuration of tools, and the more accurate models obtained from those configurations, would lead to different outcomes due to more variety being present in the EFG and considered event IDs.

### 6.2.2 Threats to External Validity

The majority of the outcomes of my experiment must be considered within a fairly specific context. Throughout this research, I am considering the construction of application-specific classifiers of feasibility which are applicable to model-based test cases. While these assumptions understandably limit the generalizability of my results, I made these choices with the novelty of the technique in mind. I chose to limit possible sources of complexity as a tradeoff for some generalizability.

The use of four AUTs from the same technology stack also limits the generalizability

of my results. The four selected AUTs may have unknown properties which cause them to perform favorably with model-based testing in general or my feasibility prediction approach.

Research Question RQ3 also presents some challenges with generalizability. In analyzing the fit of models for important features, I do not control for the overlap and correlation between features. For example, the presence of an n-gram  $e1 \rightarrow e2$  implies the presence of a before pair  $(e1, e2)$  and the presence of n-grams  $e1$  and  $e2$  of length 1. I assume that the model will choose the n-gram  $e1 \rightarrow e2$  only when that particular feature is significant, but there may not be enough data on each possible feature for the model to make a proper distinction in these cases.

As with any machine learning experiments, more data and more subjects (AUTs in this study) would only stand to improve the generalizability of the results.

### 6.3 Future Work

As mentioned, this study was intended as the first in a much longer line of research applying more general models from machine learning to software engineering challenges. In this study, I have focused on feasibility as a predicted variable; but there are a number of other interesting variables in the model-based software testing space alone:

- Coverage of events from various domains (e.g., GUI, HTTP, DB, and other layers in modern event-driven systems)
- Coverage of code and faults in code
- Security threats
- Sources of performance and other non-functional issues in an application

For the feasibility classifier in particular, there are a number of possible extensions.

First, genetic algorithms such as that proposed by Huang, et al. [18] could benefit from the availability of a reliable feasibility classifier trained by a seed suite. More generally, the classifier can be leveraged by any number of test case generation approaches which use seed suites in general, including test suite reduction methods which may need to control for feasibility.

Another angle of research on the feasibility classifier would be exploring the utility of cross-application models. Such models would require the identification of application-agnostic features, perhaps from the widget and window properties in the `GUIStructure`.

Finally, there are a number of follow-up questions from the current outcomes of this study which could benefit from further exploration:

- What is the effect of training data size on classifier performance?
- Would the use of random sequences of events gained by walking the AUT (rather than randomly selected test cases from sequence-length coverage suites) impact classifier performance?
- Of the three commonly selected categories of features identified in this study (see Outcome O6 above), could any one of these classes replace the need for one or both of the others?

And finally, I believe that the infrastructure and tools developed as a part of this work may have some utility as a more generalized framework for empirical studies of software. One angle of future research would be to apply this framework to additional GUITAR platforms and other toolsets, such as that provided by Bae, et al. [38], and evaluate its utility as a generic approach for persistence and infrastructure configuration in empirical studies of software. Use of a standard framework, and especially the establishment of requirements for reproducible configurations and other concerns (such as the work

done by COMET [45], could stand to greatly improve the quality of empirical studies of software engineering in general.

## 6.4 Conclusions

In this study, I have proposed and evaluated a novel technique for the detection of feasibility in model-based test cases of event-driven software. This technique uses logistic regression with a lasso penalty to fit a linear model to training data. I identify and extract many features from each test case, and found that event IDs serve as an important features in the classifiers. Event types from the EFG, on the other hand, did not seem to correlate with prediction. Interestingly, n-grams of length=1 and length=2 performed just as well or better than the more loosely ordered before-pairs considered.

In the future, I hope to extend this work to consider additional predicted variables from across software engineering and the SDLC, and better features for cross-application models of feasibility in particular. I would also like to separately investigate the utility of the persistence tools and other infrastructure components developed as a part of this research.

-

## Appendix A: Docker Infrastructure Scripts

Listing A.1: Jenkins GUITAR Slave

```
FROM ubuntu:14.04
MAINTAINER Bryan Robbins <bryantrobbins@gmail.com>

# Get latest packages
RUN apt-get clean
RUN apt-get update

# Needed for Jenkins
RUN apt-get install --no-install-recommends openjdk-7-jdk -y

# Needed for Jenkins jobs
RUN apt-get install ant wget unzip bzip2 xvfb curl git -y

# Install Gradle
ADD https://services.gradle.org/distributions/gradle-2.6-bin.zip /
RUN unzip gradle-2.6-bin.zip
RUN mv /gradle-2.6 /gradle
ENV PATH $PATH:/gradle/bin

# Create slave user
RUN groupadd -r jslave && useradd -r -g jslave jslave

# Stage sources
ADD sources.tar /
RUN mkdir -p /home/jslave/.gradle
RUN mv /sources/artifacts/* /home/jslave/.gradle

# Install swarm agent and start script wrapper
ADD http://maven.jenkins-ci.org/content/repositories/releases/org/jenkins-ci/
  plugins/swarm-client/2.0/swarm-client-2.0-jar-with-dependencies.jar /home/
  jslave/swarm.jar
ADD start.sh /home/jslave/start.sh
RUN chmod 700 /home/jslave/start.sh

EXPOSE 22
EXPOSE 5005

# Prepare for jslave user
RUN chown -R jslave:jslave /home/jslave
USER jslave
WORKDIR /home/jslave
```

```
# Prepare command for launching swarm client
CMD  ["/start.sh"]
```

### Listing A.2: Jenkins R Slave

```
FROM ubuntu:14.04
MAINTAINER Bryan Robbins <bryantrobbins@gmail.com>

# Get latest packages
RUN echo "deb http://cran.rstudio.com/bin/linux/ubuntu trusty/" >> /etc/apt/
    sources.list
RUN apt-key adv --keyserver keyserver.ubuntu.com --recv-keys E084DAB9
RUN apt-get update && apt-get clean

# Needed for Jenkins
RUN apt-get install --no-install-recommends openjdk-7-jdk -y

# Needed for Jenkins jobs
RUN apt-get install ant wget unzip bzip2 xvfb git -y

# Needed for R
RUN apt-get install r-base r-base-dev libxml2-dev libcurl3-dev -y
ADD install.r /install.r
RUN Rscript install.r

# Install Gradle
ADD https://services.gradle.org/distributions/gradle-2.6-bin.zip /
RUN unzip gradle-2.6-bin.zip
RUN mv /gradle-2.6 /gradle
ENV PATH $PATH:/gradle/bin

# Create slave user
RUN groupadd -r jslave && useradd -r -g jslave jslave

# Install swarm agent and start script wrapper
ADD http://maven.jenkins-ci.org/content/repositories/releases/org/jenkins-ci/
    plugins/swarm-client/2.0/swarm-client-2.0-jar-with-dependencies.jar /home/
    jslave/swarm.jar
ADD start.sh /home/jslave/start.sh
RUN chmod 700 /home/jslave/start.sh

EXPOSE 22
EXPOSE 5005

# Prepare for jslave user
RUN chown -R jslave:jslave /home/jslave
USER jslave
WORKDIR /home/jslave

# Prepare command for launching swarm client
CMD  ["/start.sh"]
```

## Appendix B: Custom R Scripts

Listing B.1: R Script: Common Utilities

```
library(data.table)

loadData <- function(file) {
  # Load massive data file from csv
  data=fread(file, stringsAsFactors=TRUE)
  cat('Before filtering:', length(names(data)), 'features in dataset', '\n')

  # Uncomment this line (and edit the first arg to grep) to drop features that
  # match some pattern
  #drop = grep("e[1-9]+", names(data), value=TRUE)
  #data=.data[, (drop):=NULL]

  # Drop any factors which do not have more than 1 level
  # ie, these are factors which appeared in test data
  filter = c('V1')
  for (ix in names(data)) {
    str <- sprintf("data$\"%s\"", ix)
    if(nlevels(eval(parse(text=str))) < 2) {
      filter <- c(filter, ix)
    }
  }
  data=data[, (filter):=NULL]
  cat('After filtering:', length(names(data)), 'features in dataset', '\n')

  # Split into sep. training and test sets
  train.data=data[isTraining=="1"]
  test.data=data[isTraining=="0"]

  # Drop filter var
  train.data=train.data[, isTraining:=NULL]
  test.data=test.data[, isTraining:=NULL]

  # Prepare training matrix
  cat('Creating training matrix', '\n')
  xm=model.matrix(isInfeas~. - 1, data=train.data, contrasts.arg = lapply(train.
    data[sapply(train.data, is.factor)], contrasts, contrasts=FALSE))
  x=apply(xm, 2, as.numeric)
  y=as.numeric(train.data$isInfeas)

  # Prepare test data
  cat('Creating test matrix', '\n')
```

```

xtm=model.matrix(isInfeas~. - 1, data=test.data, contrasts.arg = lapply(test.
  data[sapply(test.data, is.factor)], contrasts, contrasts=FALSE))
xt=apply(xtm, 2, as.numeric)
actual=as.numeric(test.data$isInfeas)

cat('Data loading complete.', '\n')

ret = list( trainData=train.data,
            testData=test.data,
            trainMatrix=x,
            testMatrix=xt,
            trainY=y,
            testY=actual
            )

return (ret)
}

loadAndWriteData <- function(fileIn, fileOut) {
  dd <- loadData(fileIn)
  saveRDS(dd, file = fileOut)
  return(dd)
}

getFromS3 <- function(accessKey, secretKey, objectKey) {
  bucket <- 'com.btr3.research'
  S3_connect(accessKey, secretKey)
  S3_get_object(bucket, objectKey, objectKey)
}

```

Listing B.2: R Script: Prepare Data

```

# I want commands printed in output
options(echo=TRUE, warning.length=8170)

# Load libs
library("rmongodb")
library("RS3")

# Load common code
source('common.r')

# Connect to mongo
m <- mongo.create(host = "guitar05.cs.umd.edu:37017")

# Verify connectivity
mongo.is.connected(m)

# LOAD THESE VALUES FROM COMMAND LINE
args <- commandArgs(trailingOnly = TRUE)
groupDb <- args[1]
groupId <- args[2]
trainingDb <- args[3]

```

```

trainingId <- args[4]
testDb <- args[5]
testId <- args[6]
accessKey <- args[7]
secretKey <- args[8]

cat("Group Object", "\n")
cat(groupDb, "\n")
cat(groupId, "\n")

cat("Training Object", "\n")
cat(trainingDb, "\n")
cat(trainingId, "\n")

cat("Test Object", "\n")
cat(testDb, "\n")
cat(testId, "\n")

#####
# IF YOU EDIT SOMETHING BELOW THIS LINE YOU BETTER #
# HAVE A REALLY GOOD REASON #
#####

# Collections
trainingCollection <- sprintf('%s.results', trainingDb)
testCollection <- sprintf('%s.results', testDb)
groupsCollection <- sprintf('%s.groups', groupDb)

# Get global features
cat('Loading group object\n')
group.query <- sprintf('{"groupId": "%s"}', groupId)
bson <- mongo.bson.from.JSON(group.query)
value <- mongo.findOne(m, groupsCollection, bson)
list <- mongo.bson.to.list(value)
featureKey <- sprintf('testCaseFeatures_n_%s', list[['maxN']])
input.suite <- list[['suiteId']]
global.features <- list[['featuresList']]
input.suite
length(global.features)

# Get training data
train.query <- sprintf('{"resultId": "%s"}', trainingId)
cat('Loading training data', '\n')
bson <- mongo.bson.from.JSON(train.query)
value <- mongo.findOne(m, trainingCollection, bson)
rlist <- mongo.bson.to.list(value)
train.passing <- rlist[['results']][['passingResults']]
train.failing <- rlist[['results']][['failingResults']]
train.all <- c(train.passing, train.failing)

# Get test data
test.query <- sprintf('{"resultId": "%s"}', testId)
cat('Loading test data', '\n')
bson <- mongo.bson.from.JSON(test.query)

```

```

value <- mongo.findOne(m, testCollection, bson)
rlist <- mongo.bson.to.list(value)
test.passing <- rlist[['results']] [['passingResults']]
test.failing <- rlist[['results']] [['failingResults']]
test.all <- c(test.passing, test.failing)

global.all <- c(train.all, test.all)
length(global.all)

# Build data frame for all examples
cat('Initializing global data frame\n')
cna <- c(list('isInfeas', 'isTraining'), global.features)
mm = matrix("0", length(global.all), length(cna), dimnames=list(global.all, cna))

for (tid in global.all){
  if(tid %in% train.failing){
    mm[tid, 'isInfeas'] = "1"
  }

  if(tid %in% test.failing){
    mm[tid, 'isInfeas'] = "1"
  }

  # Set 'isTraining' value for splitting later
  # init artifactsCollection for loading of features
  if(tid %in% train.all){
    mm[tid, 'isTraining'] = "1"
    artifactsCollection <- sprintf('%s.artifacts', trainingDb)
  } else {
    artifactsCollection <- sprintf('%s.artifacts', testDb)
  }

  # Load features
  features.query <- sprintf('{"artifactType": "%s", "ownerId": "%s"}', featureKey
    , tid)
  cat('Loading features for', tid, '\n')
  queryBson <- mongo.bson.from.JSON(features.query)
  resultBson <- mongo.findOne(m, artifactsCollection, queryBson)
  resultList <- mongo.bson.to.list(resultBson)
  for (feat in resultList[['artifactData']] [['features']]){
    if(feat %in% global.features){
      mm[tid, feat] = "1"
    }
  }
}

# Convert to df
cat('Converting to data frame')
mm = unname(mm)
global.df = data.frame(mm)
rownames(global.df) <- global.all
colnames(global.df) <- cna
okey <- sprintf('data/%s_%s_data', input.suite, featureKey)
output.file <- sprintf('%s.csv', okey)

```

```

model.file <- sprintf('%s.dat', okey)

# Write out df
cat('Writing out data frame\n')
write.csv(global.df, file = output.file)

# Convert to model
cat('Loading model structures', '\n')
data <- loadAndWriteData(output.file, model.file)

# Upload files to S3 location
bucket <- 'com.btr3.research'
S3_connect(accessKey, secretKey)
S3_put_object(bucket, output.file, output.file, "text/csv")
S3_put_object(bucket, model.file, model.file, "application/octet-stream")

# Re-print any warnings
warnings()

```

Listing B.3: R Script: Predict

```

# I want commands printed in output
options(echo=TRUE, warning.length=8170)

# Library loading
library('data.table')
library('glmnet')
library('methods')
library('mda')
library("RS3")

# Grab arguments
args <- commandArgs(trailingOnly = TRUE)
model <- args[1]
accessKey <- args[2]
secretKey <- args[3]

# Get data from S3
data.key <- sprintf('data/%s.dat', model)
bucket <- 'com.btr3.research'
S3_connect(accessKey, secretKey)
S3_get_object(bucket, data.key, data.key)

# Load model from exported file
data <- readRDS(data.key)

# Run the lasso
cvfit = cv.glmnet(data$trainMatrix, data$trainY, family = "binomial", type.measure = "class")

# Plot the fit
plot(cvfit)

# Run predictions, printing confusion matrix of t1 and t2 errors

```

```
confusion(predict(cvfit, newx = data$testMatrix, type = "class", s = c(cvfit$
  lambda.min)), data$testY)

# Inspect fit
# Inspect fit
lix <- which(cvfit$lambda == cvfit$lambda.min)
cfs <- coef(cvfit, s = "lambda.min")
sorted <- sort(cfs[which(cfs != 0)], decreasing=TRUE, index.return=TRUE)
cat('Lambda is', cvfit$lambda.min, '\n')
cat('Lambda index is', lix, '\n')
cat('Model has', cvfit$nzero[lix], 'coefficients\n')
cat('Coefficients are:\n')
for (c in rownames(cfs)[sorted$ix]) {
  cat(c, '\n')
}

# Re-print any warnings
warnings()
```

## Bibliography

- [1] Ieee standard classification for software anomalies. *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, pages 1–23, Jan 2010.
- [2] N. Ayewah, D. Hovemeyer, J.D. Morgenthaler, J. Penix, and William Pugh. Using static analysis to find bugs. *Software, IEEE*, 25(5):22–29, Sept 2008.
- [3] VictorR. Basili, Scott Green, Oliver Laitenberger, Filippo Lanubile, Forrest Shull, Sivert Srungrd, and MarvinV. Zelkowitz. The empirical investigation of perspective-based reading. *Empirical Software Engineering*, 1(2):133–164, 1996.
- [4] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [5] BaoN. Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. Guitar: an innovative tool for automated testing of gui-driven software. *Automated Software Engineering*, pages 1–41, 2013.
- [6] X. Yuan and A.M. Memon. Using GUI run-time state as feedback to generate test cases. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 396–405, 2007.
- [7] Xun Yuan and Atif M. Memon. Generating Event Sequence-Based Test Cases Using GUI Runtime State Feedback. *IEEE Transactions on Software Engineering*, 36:81–95, 2010.
- [8] Xun Yuan and Atif M. Memon. Iterative execution-feedback model-directed GUI testing. *Information and Software Technology*, 52(5):559–575, 2010.
- [9] Penelope A. Brooks and Atif M. Memon. Automated GUI testing guided by usage profiles. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 333–342, New York, NY, USA, 2007. ACM.
- [10] Scott McMaster and Atif Memon. Call-Stack Coverage for GUI Test Suite Reduction. *IEEE Transactions on Software Engineering*, 34:99–115, 2008.
- [11] T. S. Chow. Testing Software Design Modeled by Finite-State Machines. *IEEE trans. on Software Engineering*, SE-4, 3:178–187, 1978.

- [12] James M. Clarke. Automated Test Generation from a Behavioral Model. In *Proceedings of Pacific Northwest Software Quality Conference*. IEEE Press, May 1998.
- [13] Philip J. Bernhard. A Reduced Test Suite for Protocol Conformance Testing. *ACM Transactions on Software Engineering and Methodology*, 3(3):201–220, July 1994.
- [14] Sandra Rapps and Elaine J. Weyuker. Data flow analysis techniques for test data selection. In *Proceedings of the 6th international conference on Software engineering, ICSE '82*, pages 272–278, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.
- [15] Phyllis G. Frankl and Elaine J. Weyuker. An Applicable Family of Data Flow Testing Criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988. Special Section on Software Testing.
- [16] Atif M. Memon. Automatically repairing event sequence-based GUI test suites for regression testing. *ACM Trans. on Softw. Eng. and Method.*, 2008.
- [17] Bao N Nguyen and Atif Memon. An observe-model-exercise\* paradigm to test event-driven systems with undetermined input spaces. *IEEE Transactions on Software Engineering*, 99(Preliminary):1, 2014.
- [18] Si Huang, Myra Cohen, and Atif M. Memon. Repairing gui test suites using a genetic algorithm. In *ICST 2010: Proceedings of the 3rd IEEE International Conference on Software Testing, Verification and Validation*, Washington, DC, USA, 2010. IEEE Computer Society.
- [19] Z. Gao, Z. Chen, Y. Zou, and A. Memon. Sitar: Gui test script repair. *Software Engineering, IEEE Transactions on*, 2015.
- [20] S. Fujiwara, G. v.Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *Software Engineering, IEEE Transactions on*, 17(6):591–603, 1991.
- [21] H. Ural and B. Yang. A test sequence selection method for protocol testing. *Communications, IEEE Transactions on*, 39(4):514–523, 1991.
- [22] Kwang Ting Cheng and A. S. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *Proceedings of the 30th international Design Automation Conference, DAC '93*, pages 86–91, New York, NY, USA, 1993. ACM.
- [23] R. K Shehady and D. P. Siewiorek. A Method to Automate User Interface Testing Using Variable Finite State Machines. In *Proceedings of The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing (FTCS'97)*, pages 80–88, Washington - Brussels - Tokyo, June 1997. IEEE Press.
- [24] Lee White and Husain Almezen. Generating test cases for GUI responsibilities using complete interaction sequences. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 110–121, October 8–11 2000.

- [25] Lee White, Husain Almezen, and Nasser Alzeidi. User-Based Testing of GUI Sequences and Their Interactions. In *ISSRE '01: Proceedings of the 12th International Symposium on Software Reliability Engineering*, page 54, Washington, DC, USA, 2001. IEEE Computer Society.
- [26] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. Hierarchical GUI Test Case Generation Using Automated Planning. *IEEE Transactions on Software Engineering*, 27(2):144–155, February 2001.
- [27] Atif M. Memon. An event-flow model of GUI-based applications for testing. *Softw. Test. Verif. Reliab.*, 17:137–157, September 2007.
- [28] Atif M. Memon, Mary Lou Soffa, and Martha E. Pollack. Coverage criteria for GUI testing. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, volume 26 of *ESEC/FSE-9*, pages 256–267, New York, NY, USA, September 2001. ACM.
- [29] A. Memon, I. Banerjee, and A. Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *Reverse Engineering, 2003. WCRE 2003. Proceedings. 10th Working Conference on*, pages 260–269, 2003.
- [30] Xun Yuan, Myra B. Cohen, and Atif M. Memon. GUI Interaction Testing: Incorporating Event Context. *IEEE Trans. Software Eng.*, 37(4):559–574, 2011.
- [31] Ishan Banerjee, Bao Nguyen, Vahid Garousi, and Atif Memon. Graphical user interface (gui) testing: Systematic mapping and repository. *Information and Software Technology*, 2013.
- [32] Atif Memon, Ishan Banerjee, Nada Hashmi, and Adithya Nagarajan. DART: A Framework for Regression Testing "Nightly/daily Builds" of GUI Applications. In *Proceedings of the International Conference on Software Maintenance, ICSM '03*, pages 410–, Washington, DC, USA, 2003. IEEE Computer Society.
- [33] D. J. Richardson, S. Leif Aha, and T. O. OMalley. Specification-based Test Oracles for Reactive Systems. In *Proceedings of the 14th International Conference on Software Engineering*, pages 105–118, May 1992.
- [34] Debra J. Richardson. TAOS: Testing with Analysis and Oracle Support. In Thomas Ostrand, editor, *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA): August 17–19, 1994, Seattle, Washington, USA*, ACM Sigsoft, pages 138–153, New York, NY 10036, USA, 1994. ACM Press.
- [35] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. Automated Test Oracles for GUIs. In *Proceedings of the ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering (FSE-8)*, pages 30–39, NY, November 8–10 2000.
- [36] Qing Xie and Atif M. Memon. Designing and comparing automated test oracles for GUI-based software applications. *ACM Trans. Softw. Eng. Methodol.*, 16(1):4, 2007.

- [37] Xun Yuan and Atif M. Memon. Alternating GUI Test Generation and Execution. In *Proceedings of the Testing: Academic & Industrial Conference - Practice and Research Techniques*, pages 23–32, Washington, DC, USA, 2008. IEEE Computer Society.
- [38] Gigon Bae, Gregg Rothermel, and Doo-Hwan Bae. Comparing model-based and dynamic event-extraction based gui testing techniques. *J. Syst. Softw.*, 97(C):15–46, October 2014.
- [39] A. Memon, I. Banerjee, B.N. Nguyen, and B. Robbins. The first decade of gui ripping: Extensions, applications, and broader impacts. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 11–20, Oct 2013.
- [40] R. W. M. Wedderburn J. A. Nelder. Generalized linear models. *Journal of the Royal Statistical Society. Series A (General)*, 135(3):370–384, 1972.
- [41] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58(1):267–288, 1996.
- [42] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Regularization paths for generalized linear models via coordinate descent. *Journal of Statistical Software*, 33(1):1–22, 2010.
- [43] Daniel Jurafsky and James Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Pearson Prentice Hall, 2nd edition, 2009.
- [44] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.
- [45] Zebao Gao, Yalan Liang, Myra B. Cohen, Atif M. Memon, and Zhen Wang. Making system user interactive tests repeatable: When and what should we control? In *Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15*, pages 55–65, Piscataway, NJ, USA, 2015. IEEE Press.
- [46] Atif M. Memon. Automatically repairing event sequence-based GUI test suites for regression testing. *ACM Trans. Softw. Eng. Methodol.*, 18:4:1–4:36, November 2008.