UNIVERSITY OF CALGARY

Enhancing Exploratory Testing with Rule-Based Verification

by

Theodore D. Hellmann

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

August 2010

UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "Enhancing Exploratory Testing with Rule-Based Verification " submitted by Theodore D. Hellmann in partial fulfilment of the requirements of the degree of Master of Computer Science.

Supervisor, Dr. Frank Oliver Maurer, Department of Computer Science

---

Dr. Jörg Denzinger, Department of Computer Science

Professor Denis Gadbois, Faculty of Environmental Design

Dr. Mario Costa Sousa, Department of Computer Science

Date

Abstract

Testing graphical user interfaces (GUIs) is difficult – they're enormously complex, difficult to verify, and very likely to change. This thesis presents an overview of these challenges, followed by an overview of previous attempts to make GUI testing practical that categorizes each approach based on the challenge it addresses. A new tool for GUI testing, LEET, was created to apply automated rule-based verifications to manual exploratory test sessions that had been recorded. Five main research questions are identified:

1) Can rule-based exploratory testing be used to catch high-level, general bugs?
2) Can rule-based exploratory testing be used to catch low-level, specific bugs?
3) Can rules be reused in tests for different applications?
4) How often is it possible to use keyword-based testing on GUIs?
5) Is rule-based exploratory testing less effort than writing equivalent tests by using a capture/replay tool and inserting verifications manually?

A preliminary evaluation of rule-based exploratory testing with LEET is performed. This pilot study was able to provide some answers to the first four questions. It was found that rules can be used to detect both general and specific bugs, but rules for general bugs were easier to transfer between applications. It was also suggested that, despite the advantages that keyword-based testing presents to human testers, it interferes with the process of creating automated test oracles and the process of transferring rules between applications. The fourth question, however, was unanswerable based on the results of these pilot studies, so instead a set of recommendations for future work on the subject is presented.

Acknowledgements

This thesis was not a work in isolation. It is the culmination of the guidance and support I received from many people – and I'd like to thank a selection of them here.

- To my especially influential undergraduate professors: Lynn Lambert, Anton Riedl, and especially Roberto Flores. Without your tolerance, good humour, and support, I would never have made the leap from Christopher Newport University to the University of Calgary.
- For their feedback, guidance, and, especially, reality checks: Frank Maurer and Jorg Denzinger. Thanks for putting me on the right track, keeping me there, and seeing this thing through to the finish.
- To all my friends – especially Ali Hosseni Khayat, without whom LEET would have been impossible. Thanks for your friendship, support, and tolerance throughout the years.
- To my family, for allowing my interest in the sciences to develop from such an early age: thanks for always believing in me.
- To my kind and loving wife, Christin, who helped me through all the migraines and the self-doubt that come with doing graduate work. Without you, I would simply have given up on this whole thing long ago.

Table of Contents

List of Tables

List of Figures and Illustrations

List of Symbols, Abbreviations and Nomenclature

Symbol	Definition
AKG	Actionable Knowledge Graph
AKM	Actionable Knowledge Model
APDT	Agile Planner for Digital Tabletops
CWE	Common Weakness Enumeration
EFG	Event-flow Graph
EIG	Event-interaction Graph
GUI	Graphical User Interface
LEET	LEET Enhances Exploratory Testing
MEEC	Minimal Effective Event Context
SGL	Sub-Goal List
TDD	Test-Driven Development
TWOM	Testing with Object Maps
UI	User Interface

Epigraph

A process cannot be understood by stopping it. Understanding must move with the flow of the process, must join it and flow with it.

Frank Herbert, *Dune*

Chapter One: Introduction

Nearly every modern application uses a graphical user interface (GUI) as its main means of interacting with users. However, because GUIs allow users so much freedom of interaction, they are very difficult to test. This thesis discusses a new approach to GUI testing that enhances the process of manual exploratory testing with automated, rule-based verifications.

Previous research has mainly focused on how to automate the process of writing tests, as seen in Chapter 2. Automating GUI tests is difficult because GUIs are very complicated, writing automated test oracles for GUIs is difficult, and changes to GUIs that do not change the functionality of an application can still break automated GUI tests, as seen in Section 1.3. Rule-based testing, however, is a form of automated testing which reports erroneous application behaviour instead of verifying proper behaviour, which makes creating automated test oracles easier and reduces the chances that a test will break when a GUI is changed. Exploratory testing is a way of testing GUIs manually, but it too is hampered by the complexity of GUIs. Combining exploratory testing and rule-based testing makes use of the strengths of both of these approaches and provides a new, promising approach to GUI testing.

The first step to combining these two methods is to record the interactions performed during an exploratory test session as a replayable script. Next, a human test engineer defines a set of rules that can be used to define the expected (or forbidden) behaviour of the application. The rules and script are then combined into an automated regression test that can be used to expand the amount of the system that was originally investigated by exploratory testing. This approach allows a human tester to select specific parts of the

system that should be tested further, which, in effect, decreases the amount of manual testing that needs to be done. At the same time, this subset of the application is tested thoroughly using automated rule-based testing in order to verify more properties of more parts of the application than would be possible with exploratory testing alone.

This thesis provides a review of literature detailing previous attempts to automate GUI testing. It also discusses the method of creating a system to combine exploratory testing with automated rule-based testing. This system was implemented as LEET (LEET Enhances Exploratory Testing), and a pilot evaluation was used to determine that rule-based exploratory testing is a practical approach to GUI testing that deserves further study.

This chapter provides background material necessary to understanding the contributions of this thesis. First, the structure of GUIs is discussed. An overview of testing is provided next, followed by a discussion of the benefits and liabilities of manual and automated testing techniques. Finally, the difficulties involved with testing GUIs are discussed. Ways in which this rule-based exploratory approach can help address these difficulties are discussed. Research questions and goals are also stated.

1.1 Graphical User Interfaces

GUIs allow users to interact with applications via text input, mouse movement, and mouse clicks, which makes interacting with computers more natural and intuitive. GUIs are now an essential part of most modern applications, and very few do not contain some form of GUI. Of these GUI-based applications, 45-60% of the total code of the application can be expected to be dedicated to its GUI [1].

In modern, GUI-based operating systems, the root of a GUI must be a *window*. Windows are special containers of *widgets*, or *user interface elements* (UI elements), that are grouped hierarchically. Widgets receive a user's interactions, such as mouse clicks or text input. These interactions are then received by the application containing the widget, which should respond to the user's interaction by updating its GUI. GUIs are composed of one or more windows, each of which contains of a set of one or more widgets.

This thesis focuses specifically on GUIs running on Microsoft's Windows line of operating systems. The operating system used for evaluations was the 32-bit version of Windows 7. In other operating systems, Linux variants in particular, GUIs usually represent an interface with an application running from the command line, a *text-only interface*. Text-only interfaces only allow for interactions based on textual input, and are less complicated (and therefore easier to test) than GUI-based applications. However, in Windows, GUIs are tightly integrated with applications, and therefore testing an application through interactions with its GUI, rather than testing an application and its GUI separately, is an important topic.

Additionally, the approach to testing GUIs as presented in this thesis focuses on *event-driven* applications. In event-driven applications, every function performed by the application is a response to a user interacting with the application. For example, when a user clicks a button, some function of the application is invoked. Most desktop applications and most websites fall into the category of event-driven applications. Another class of applications can be described as *loop-driven*. Loop-driven applications perform functions primarily in response to internal computations or the passage of time, though user events also invoke functionality. Computer games, for example, are

primarily loop-driven in nature. In this thesis, an event-driven approach to testing is explored, and so only desktop and web-based applications are considered.

1.2 Testing – An Overview

It is a basic fact of software development that applications will contain *defects*, or errors in the code of the application. Defects may or may not cause *faults*, or discrepancies between the desired behaviour of the system and its actual behaviour when run. *Failures* occur when the faulty state of the system causes a noticeable error. The term *bug* is slang, and may be used informally to refer to any of the above – but, in this paper, "bug" will always refer to a software failure. There is not necessarily a one-to-one relationship between defects, faults, and failures. The same fault can be caused by multiple defects in the code of an application, and a single failure can be caused by different faults.

Testing is the process of interacting with an application in an attempt to find bugs so that they can be fixed. Thus, the purpose of testing is to increase the odds of an application functioning properly for users. Tests can be used to find bugs in an application, but they are not able to show that an application has no bugs.

A test consists of two parts: a *test procedure* and a *test oracle*. A test procedure is an ordered set of interactions that explore an application's functionality. This can be envisioned as a script of interactions to take in order to execute a test. The term *state* refers to the values of all of the properties of different parts of an application at a given point in time, and a test oracle describes the expected state of an application during execution of the test procedure. Bugs are exposed when the execution of a test procedure triggers the code in which they are contained and the test oracle notices the discrepancy between the expected state of the application and its actual state.

Testing can be performed from two perspectives: *black box* or *white box* (sometimes known as *glass box*). In white box testing, an application's code is visible to the test procedure. White box testing is used to determine the correctness of the internal structures of an application: variables can be set, classes can be created, and individual methods within the application can be called directly by the test. Black box testing, on the other hand, tests the functionality of an application from a user's point of view. The application is interacted with through its interface, and its source code is not used. White box testing is useful for detecting whether parts of an application are working correctly, while black box testing is useful for determining whether the application provides the desired functionality to the user.

Testing can also be performed at a variety of levels and for a variety of purposes. *Unit testing* tests individual classes and methods of an application, and is used to determine if individual parts of the system work correctly in isolation. *Integration testing* tests different subsystems of the application in order to determine whether different parts of the system will function correctly when interacting with each other. *System testing* tests the application as a whole, and is used to determine if the application as a whole is working correctly. *Acceptance testing* is similar to system testing in that it tests the entire application, but the purpose of acceptance testing is to determine if the application provides the functionality it was originally intended to. *Regression testing* is the re-running of previous tests on an application after changes have been made to it, and is important in ensuring that changes to one subsystem of an application don't end up breaking a seemingly-unrelated subsystem.

Manual testing is a form of testing in which human testers execute tests on an application through its user interface. Generally, these tests are defined beforehand. However, *exploratory testing* is a form of manual testing that does not use pre-defined tests, and instead relies on the skills and intuition of individual testers. *Automated testing* is performed by specifying tests in terms of executable code, which a computer then runs. Each of these techniques is described in detail in the following subsections.

The subject of this thesis is a new approach to testing graphical user interfaces by enhancing exploratory testing with rule-based verification. This approach leverages manual exploratory testing to provide test procedures and leverages the advantages of automated testing through use of rule-based test oracles. This approach is a black box, system-level approach to testing the functionality of an application through its graphical user interface. While the exploratory testing involved is manual, recording these tests and enhancing them with rule-based verification leverages the advantages of automated testing.

1.2.1 Manual Testing

In manual testing, a human will interact with a system in order to attempt to uncover bugs. Manual testing is generally understood as scripted manual testing, in which a tester follows a predefined script when interacting with the system. These scripts are usually created by test engineers, and then run later by software testers. Scripts for manual testing can be viewed as test programs written to be executed by humans rather than by machines. Scripted manual testing is a tedious process when done repetitively (as for regression testing).

A linear expenditure of effort is required to perform manual testing on new features as well as regression tests of existing features. The graph shown in Figure 1 shows a hypothetical example of the amount of effort required to perform this degree of testing over several iterations of software development.

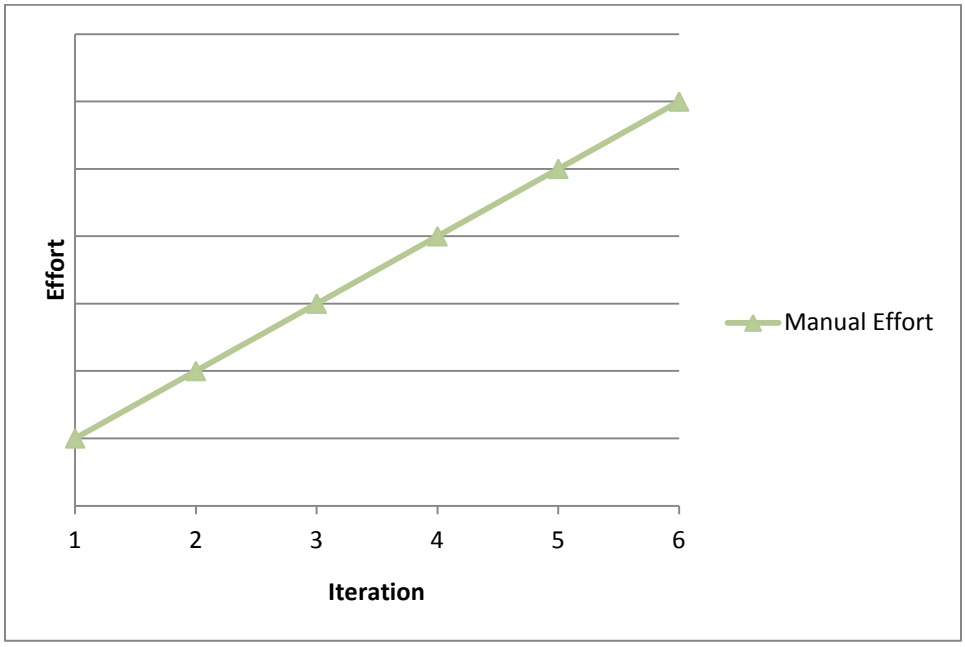

Figure 1: Effort required to manually test an application as features are added

1.2.1.1 Exploratory Testing

Exploratory testing is a form of manual testing, but it is significantly different from scripted manual testing. Exploratory testing has been defined as "simultaneous learning, test design, and test execution" [2]. In exploratory testing, no predefined test cases are used, which allows testers to perform ad-hoc interactions with the system based on their knowledge, experience, and intuition in order to expose bugs.

Scripted testing has the advantage of consistency, while exploratory testing benefits from its reliance on human ingenuity. Both avoid the difficulties involved with creation of an

automated test oracle, as described in Section 1.3.2, by relying on a human's ability to judge whether or not a system meets expectations.

Despite its unplanned, freeform nature, exploratory testing has become accepted in industry, and is felt to be an effective way of finding defects [3]. Practitioner literature argues that exploratory testing also reduces overhead in creating and maintaining documentation, helps team members understand the features and behaviour of the application under development, and allows testers to immediately focus on productive areas during testing [3] [4]. Further, a recent academic study shows that exploratory testing is at least as effective at catching bugs as scripted manual testing, and is less likely to report that the system is broken when it is actually functioning correctly [4].

However, there are several major difficulties involved with exploratory testing. First, exploratory testing is limited in that human testers can only test so much of an application in a given time. This virtually ensures that parts of the application will be insufficiently tested if exploratory testing is the only testing strategy used to test an application. Figure 2 shows a brief example of the path through the states of an application that might be explored in an exploratory test session. Many states of the application are left unexplored in this example. Second, it is often difficult for practitioners of exploratory testing to determine what sections of an application have actually been tested during a test session [3]. This makes it difficult to direct exploratory testing towards areas of an application that have not been tested previously, and runs the risk of leaving sections of

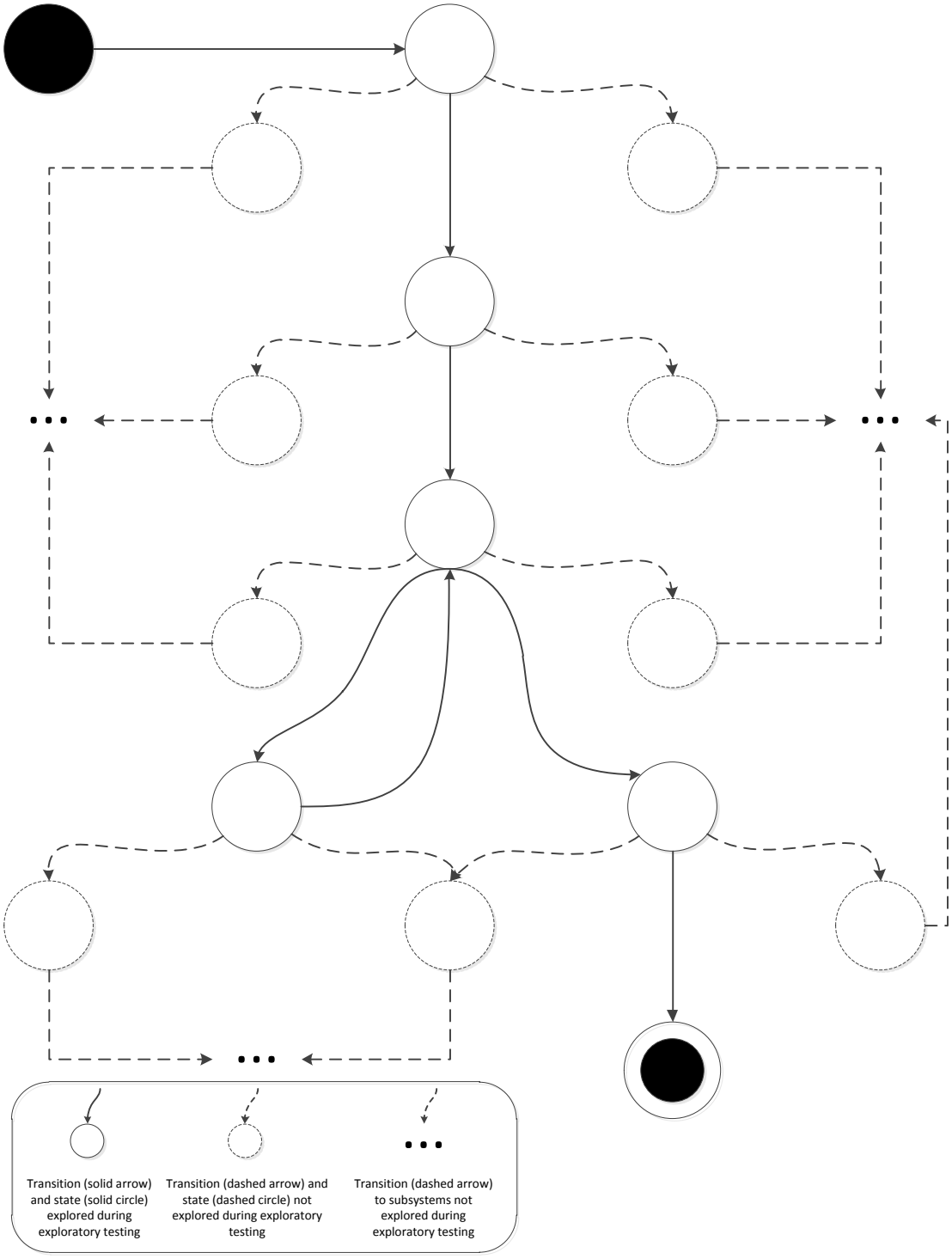

Figure 2: State diagram of manual test session

an application untested. The difficulties encountered in testing modern software systems are explored in detail in Section 1.3.

Because of this, practitioners of exploratory testing argue for a diversified testing strategy, including exploratory, scripted, and automated testing [5]. Such a testing strategy would combine the benefits of exploratory testing with the measurability, thoroughness, and other advantages of automated testing described in the following subsection. However, there is a lack of tool support that would augment exploratory testing with scripted and automated testing techniques. One of the contributions of this thesis is the creation of such a tool and a pilot evaluation of its abilities.

1.2.2 Automated Testing

In automated testing, both the test procedure and test oracle are defined in terms of executable code. These automated tests are then used to verify that the application functions as expected throughout development. The main advantage of automated testing is that the effort required of human test engineers is nearly constant. Because executable tests can be rerun without effort on the part of human testers, regression testing is much simpler in automated testing situations. Humans are still required to write new tests and to fix broken tests, but tests are run automatically. This resulting necessary effort to write new tests for each iteration can be seen in Figure 3.

There are many additional advantages to automated testing. First, automated testing is highly repeatable. If a bug is detected during test execution, the failing test can simply be rerun to determine if the bug is reproducible. This can make it easy to isolate the specific interaction that triggers a bug. Second, many code coverage tools exist that work well in tandem with the execution of automated tests. This makes it easier to determine which

areas of an application need additional testing. Third, many techniques exist for automatically generating automated tests that thoroughly explore an application under test. Some of these are explored in detail in Section 2.1.2.

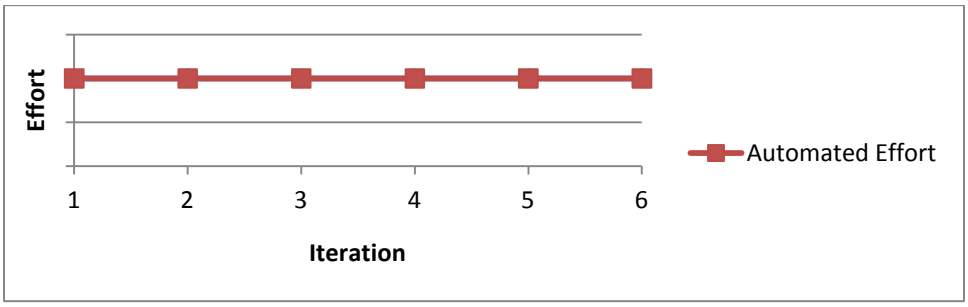

Figure 3: Effort required to implement automated tests for an application as features are added

Even if coding an automated test is assumed to be more difficult than running a manual test, over time automated testing can be expected to save a significant amount of effort provided that regression testing is thoroughly conducted, as in the conceptualization shown in Figure 4. However, this is not always the case. For example, if an automated test requires frequent maintenance, or if only a few iterations of software development are being performed, then automated testing may well require more effort than a manual approach. For reasons explained in the Section 1.3, the effort involved in writing automated tests is highly variable, and can negate the benefit of not requiring a human to execute tests.

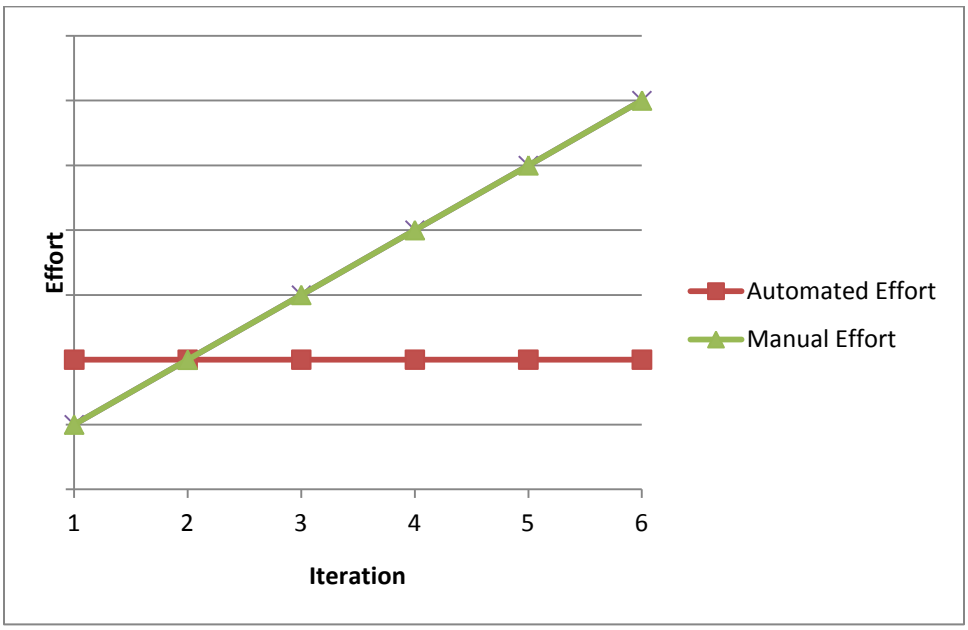

Figure 4: Comparison of expected effort involved in manual and automated testing

1.2.2.1 Rule-Based Testing

A *rule* is a way of expressing the permissible states of an application. Rules can be described using an "if... then...so..." formulation. For example, "**if** you are under the age of 21, and **if** you are in the United States, and **if** you are not a member of the armed forces, **then** you cannot legally purchase alcohol, **so** if you do a crime has been committed" is a rule describing alcohol regulations in the United States.

In this example, the "if..." part of a rule is known as a *precondition.* A precondition is a simple check to see whether a rule should apply given the current state of the application under test. If rules have more than one precondition, all preconditions must be met in order for the rule to apply. If the preconditions are not met, then the rule is finished executing. Preconditions can be shared among rules, and this fact can be used with algorithms like the Rete Algorithm in order to reduce the overhead involved with verifying preconditions in rule-based testing [6].

If a rule's preconditions are met, then its *action* should be run. A rule's action is used to ensure that a piece of the functionality of the system is working as expected within a given state, as determined by the precondition. A rule's action must at least perform a verification about the state of the system, but, in this thesis, it can also be used to drive test execution in order to perform verifications. The action associated with a rule should not be confused with actions used in test scripts, which signify interactions with the application under test only and do not perform verifications. For example, in #Section, rule actions are used to interact with the system under test so that further verifications, also carried out as part of the rule action, will be possible. While this definition is currently used in the implementation discussed in this thesis, it will be advisable for future work to split rule actions into a discrete action and a postcondition in order to intellectually separate each part of the rule.

If a rule's verification is not met, there will be a *consequence* for the failure of the application to meet expectations. Consequences can be either fatal or nonfatal. Fatal consequences indicate that a feature of the system does not meet expectations, while nonfatal consequences are used to draw a tester's attention to suspicious system behaviour. In *rule-based testing*, a set of rules for an application is paired with a test procedure and used as a test oracle.

This thesis presents a combination of manual and automated testing techniques that leverages the advantages of both techniques. Manual exploratory testing is paired with automated rule-based testing, and the pilot evaluation of this approach, presented in Chapter 5, shows that this pairing is promising and that further investigation should be performed.

1.3 Major Challenges of GUI Testing

This section explores the issues encountered when testing modern software systems. The underlying difficulty is that users interact with nearly all applications through a GUI. GUIs allow users to interact with applications via text input, mouse movement, and mouse clicks. Because GUIs offer users great freedom of interaction, the number of possible interactions with a GUI is very large, which makes GUIs enormously complicated from a testing perspective. This makes interacting with computers more natural and intuitive than using a command line interface and allows users to interact with more than one application at a time. GUIs are now an expected part of most applications. This thesis addresses the problem of testing an application with a GUI. It is understood that tests will interact with an application through its GUI, as users would. This is easily accomplished through an exploratory testing approach, but difficult to do with automated tests. Because this thesis explores an approach that makes use of a form of automated testing, it is important to understand the reasons that automated GUI testing is difficult. In this subsection, the issues involved in performing automated testing of an application with a GUI are broken down in to three categories, and each is explored individually.

1.3.1 Complexity

The more freedom a user is allowed when interacting with an application's GUI, the larger the application's interaction space becomes. Even a relatively simple GUI can present an alarmingly large number of different ways in which a user can interact with it. In terms of testing, this means that the number of possible sequences of events increases exponentially with the length of a test script [7]. Further, the potential a suite of GUI tests has to find bugs in an application is determined by two factors: the number of

possible events that have been triggered; and the number of different states from which each of these events is triggered [8].

This means that, in order for a suite of tests to have a reasonable chance of detecting bugs in an application, its tests will need to trigger as many events as possible from as many different states as possible. In other words: there is an impractically huge amount of testing that could be done on an application through its GUI, and it's hard to detect bugs without doing a significant portion of this testing.

The complexity of GUIs is a problem for manual and automated testing alike. For manual testing, the main problem is that the human effort required to test a complex system is prohibitive. While exploratory testing shows promise in terms of efficiently identifying aberrant behaviour in GUI-based applications [3] [4], it remains impractical to thoroughly or repeatedly test an application using only an exploratory approach. With automated testing, it is possible to run a large number of tests cheaply, so if automated tests are generated automatically, as in Section 2.1.2, it may be possible to thoroughly test an application or to run tests repeatedly. However, doing so takes time and it may not be practical to run such a test suite on a regular basis [9].

The approach described in this thesis leverages the ability of exploratory testing to focus on an "interesting" part of an application under test and to determine correctness or incorrectness of the application's features. By focusing effort on testing one part of a system over another, or testing one part of a system in detail, human exploratory testers are implicitly exercising their judgement to identify interesting parts of an application. Automated rule-based testing can then be used to perform further testing on this subsystem. In this way, a only a subset of an entire application is tested, but this

subsystem has been deemed important due to the focus placed on it through exploratory testing and will be tested thoroughly through rule-based verifications.

1.3.2 Test Oracle Problem

In exploratory testing, a human tester's expectations are used in place of a test oracle to determine if a test fails or succeeds. Creating automated test oracles to replace human intuition is a significant problem. The effectiveness of automated testing is limited by the ability of test engineers to define automated test oracles [10]. In automated testing, test oracles consist of sets of values that properties of the components of an application should have after a given step in the test procedure. These expected values are then automatically compared to the actual values of those properties in the application during the execution of a test. The more values a GUI test oracle is comparing, the stronger its bug-detection ability [11]. For example, an GUI test oracle that verifies many properties of many different widgets after each step of a test script is more likely to notice bugs than an oracle that verifies only properties of fewer widgets, or one that only performs verifications once at the end of a test. Even if the procedure exposes a bug in all of these cases, if the automated oracle isn't sufficiently-detailed, the bug could go uncaught.

However, detailed oracles are difficult to create and take much processor time to run [11]. In the previous section, it was mentioned that suites of tests for GUI-based applications needed to trigger many events from many different states. In addition, this section shows that, when relying on automated testing techniques alone, detailed test oracles would need to be created to verify that each of these individual interactions is correct.

The approach described in this thesis focuses on using rules as automated test oracles. The rules themselves, however, are defined by a test engineer to specify details about the

behaviour of only certain parts of an application. In this way, portions of an application that are judged to be likely sources of bugs, or have proven to be buggy in the past, are singled out for detailed scrutiny.

1.3.3 The Impact of Changes to the GUI on GUI tests

A problem specific to automated testing of GUI-based applications is that changes to the GUI are able to break tests even when the underlying application is not changed in any important sense. This is due to the fact that GUI testing frameworks interact with GUIs by looking up specific widgets and then sending events to them. The process of looking up widgets for interaction is what makes GUI tests fragile. GUI testing frameworks tend to interact with a GUI from another process in order to interact with a GUI in a similar way to a user. This fragility would not be reduced by executing GUI tests from within the same process as the GUI itself is running, as it would still be necessary to look up a widget in the running GUI from its representation within the code of the application. When interacting with a GUI from another process, as shown in Figure 5, these interactions are performed through the mediation of the operating system on which the test and application are running. Clicking a button, for example, is achieved either through moving the mouse over the button and sending the operating system a mouse click event or through the use of an accessibility framework that is able to directly call all of the events that would be invoked through this mouse click. After receiving these messages, the operating system will then pass appropriate events along to the application.

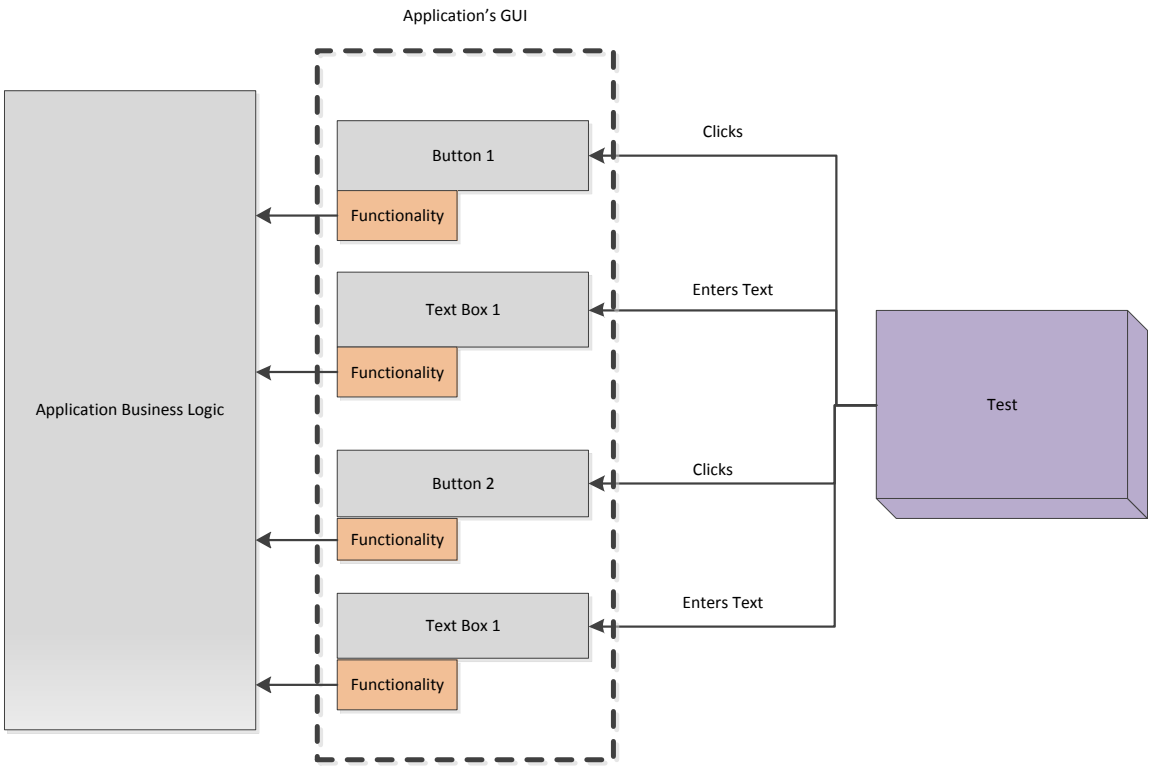

Figure 5: Testing a GUI-Based Application Through OS-Based Actions

A GUI can also be tested from within the same process, either by including tests in the same project so that they are compiled and run in the same process as the application itself or by injecting tests into a running application using aspect-oriented approaches. The issue with this is that additional events may be called by clicking a widget that would not be called by directly calling the most relevant method within an application from test code. For example, calling a button's "OnClicked" method from a test will invoke that method, but would not perform additional interactions that would be raised if a user had clicked the button – selecting the button, for instance. These additional events will not be invoked through in-process testing, even though they may be important to the way the application responds.

On the other hand, running test code from a separate process usually forces tests to interact with an application in a very similar way to the way in which a user would, the exception to this being when reflection is used to access the internals of a process. Out of process testingdoes, however, introduce the complication of locating specific widgets within the application's process so that they can be interacted with.

The first method of interacting with an application's GUI from an out-of-process test used absolute screen coordinates. For example, if one step of a test procedure was to click a close button, this button was assumed to be in a certain absolute location on the screen. The mouse cursor would then be moved to this location, and a mouse click event raised. Using a coordinate system relative to the window of the application's GUI rather than absolute screen coordinates avoids the obvious problem of windows opening at different locations at the screen. However, even using relative coordinates will cause a test procedure to break when a button is moved during the redesign of a GUI. Even though the functionality of the application will be unchanged in this case, a test involving the moved button will falsely indicate a bug in the application's functionality.

One solution is to use an *accessibility framework* to locate and interact with widgets. Accessibility frameworks provide information about the individual elements that make up a GUI. When using an accessibility framework, widgets of a GUI implement an AutomationPattern – an interface through which tests and other applications can easily access properties and functionality of widgets. This information can include the name of a widget, a unique identifier for it, its type, values of its different properties, and ways of interacting with it. Interactions from a test with a GUI-based applications through the mediation of an accessibility framework can be seen in Figure 6. These tools are often

used to enable users with disabilities – for example blindness – to use GUI-based applications with the support of assistive technologies – like screen readers. However, by using an accessibility framework to interact with elements of a GUI, the additional events that are not raised through in-process testing can be raised. Examples of accessibility framework include Microsoft's Active Accessibility [12] and Windows Automation API [13].

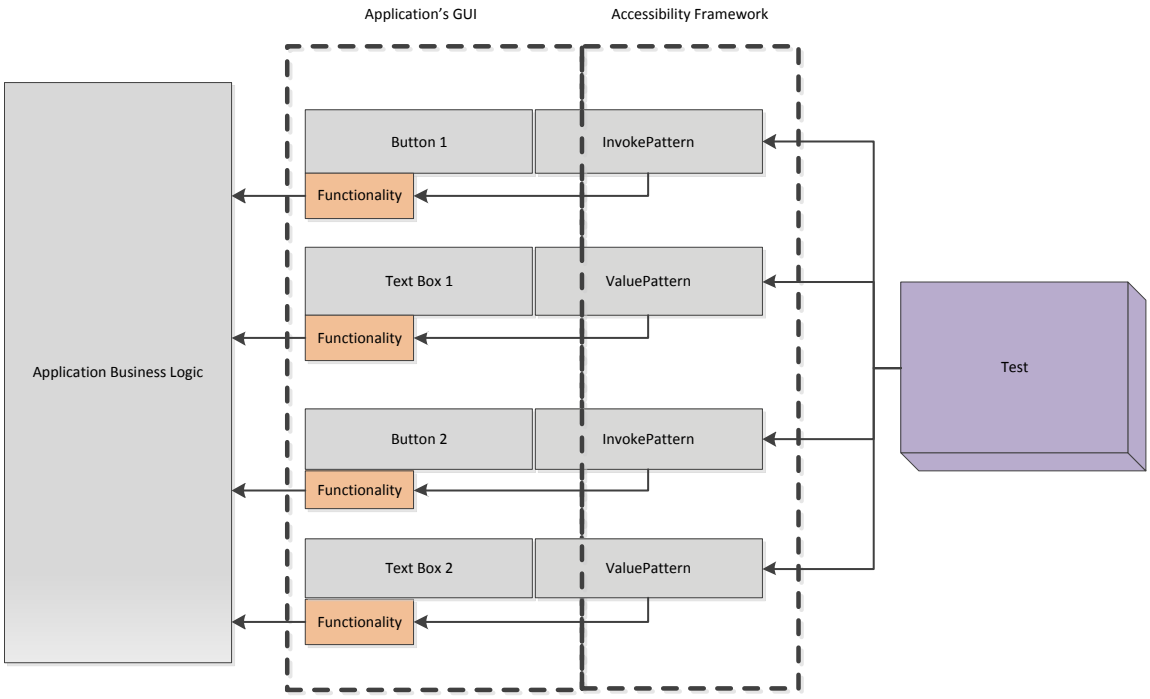

Figure 6: Testing a GUI-Based Application via an Accessibility Framework

Since accessibility frameworks can provide information that can be used to identify widgets uniquely, it is possible to find widgets of an application that match given parameters while running a test. There are two main schools of thought on how this should be done. In the first, called *testing with object maps*, as much information as possible about a widget is recorded when the test is written. This information can include

the name of a widget, its position on screen, its background color, and so on. Then, when a test is run, the widget most closely matching this set of information can be located. In the second approach, called *keyword-based* testing, each widget in an application is required to have a unique identifier that can be used to locate it when a test is run. In this way, only a single piece of information is required in order to locate a specific widget. While both of these methods decrease the risk of breaking tests when changing an application's GUI, keyword-based testing is easier for human testers to use. Because of this, LEET was implemented using keyword-based testing. However, as was discovered in Chapter 5, keyword-based testing is difficult to use with automated tests, so future work involving rule-based exploratory testing should make use of testing with object maps.

It is still possible to change a GUI without changing the underlying application's functionality in ways that will cause a test to break. For example, imagine a login screen like that found on many web pages. It consists of two text boxes: one for a username and one for a password. Before revision, entering an incorrect username/password pair will cause a new page to load which informs the user of the problem. After revision, a popup window might be created to display the same information, without changing the content of the current page. While the basic functionality of this feature, "inform the user that their name and/or password are incorrect," has not changed, the way the GUI displays this information has changed sufficiently to break tests.

This sort of scenario is very common, with up to up to 74% of test cases rendered unusable after modifications to an application's GUI [14] [15]. This means that broken test scripts will need to be repaired or that equivalent tests that will work on the new

version of the GUI will need to be created. When tests break too often, there's a risk that developers will start to dismiss failing tests as simply the test's fault, in which case developers will be less likely to rely on the test suite to catch errors, which defeats the purpose of automated testing in the first place [16]. Research on this topic is discussed in Section 2.2.

In order to minimize the risk of tests erroneously reporting bugs in an application, the approach proposed in this thesis leverages rule-based testing. Since rule-based testing is used to specify forbidden application states, rather than specific expected states, use of this approach reduces the odds of false bug reports at the risk of not detecting bugs in cases where preconditions of a rule are not met. However, since tests do use script-based test procedures, it is still possible for tests to break based on errors locating widgets to interact with.

1.3.4 Real-World Cost of GUI Bugs

Perhaps the most challenging aspect of GUI testing is that GUI-based bugs do have a significant impact on an application's users. Studies done at ABB Corporate Research support this concern [17]. 60% of defects can be traced to code in the GUI, which is directly in line with what is expected from the percentage of GUI code in an application. While it might be tempting to think that defects arising from a GUI might be trivial, these studies showed the opposite: 65% of GUI defects resulted in a loss of functionality. Of these important defects, roughly 50% had no workaround, meaning the user would have to wait for a patch to be released to solve the issue.

This means that, regardless of the difficulties involved, GUI testing is still a vital part of the development of an application. GUIs need to be well-tested in order to reduce the

instances of bugs being discovered not before release of an application by testers, but after release by customers. Despite the usefulness of automated testing in finding bugs, it is currently common for industrial testers to bypass automated GUI testing by interacting with the GUI through a specific test interface [18] or to skip automated GUI testing entirely [19]. In this second case, an attempt is made to keep the GUI code too simple to possibly fail, and the GUI is then tested manually while the rest of the application is tested with automated test suites.

While this approach may be sufficient in certain situations, the rule-based exploratory testing proposed in this thesis addresses the issues that prevent automated GUI testing from being widely-used, and provides a more rigorous method for finding bugs than manual testing alone.

1.4 Research Questions

This thesis presents a new approach to rule-based exploratory testing of GUI-based applications that leverages keyword-based testing in order to locate widgets for testing. Because this approach has not been attempted in detail previously, it is important to discover what kind of testing can take advantage of this kind of approach. Two of the research goals of this thesis can be derived directly from this fact:

1) Can rule-based exploratory testing be used to catch general bugs?
2) Can rule-based exploratory testing be used to catch specific bugs?

In these questions, general bugs are defined as bugs that could occur in many sorts of interfaces. For example, one general bug discussed in Chapter 5 is that widgets can respond to events even if they are not in the visible area of the computer's screen. This situation could occur in any GUI-based application, so it is considered to be a general

bug. Specific bugs, on the other hand, are defined as bugs that could only occur in specific types of interfaces. For example, the second bug discussed in Chapter 5 deals with a bug that can only occur in web-based applications that make use of a specific interface through which a user can enter his or her age. This bug is considered to be specific because it can only be encountered in very specific interfaces that are used in very few GUI-based applications. A tangential question to these first two is:

3) Can rules be reused in tests for different applications?

This third question is addressed in Sections 5.2.1 and 5.2.2, by creating rules based on general and specific bugs and attempting to apply these rules to various GUI-based applications. Rules are considered reusable if it is possible to apply them to different applications without changing them.

A fourth question was raised during the course of the preliminary evaluation:

4) How often is it possible to use keyword-based testing on GUIs?

This question was raised in response to various difficulties in conducting the investigations in Sections 5.2.1 and 5.2.2. Since there is an alternative to keyword-based testing – testing with object maps – this is a particularly relevant question.

Finally, an attempt was made to address the underlying question of this thesis:

5) Is rule-based exploratory testing less effort than writing equivalent tests by using a capture/replay tool and inserting verifications manually?

This fifth research question is intended to discover whether rule-based exploratory testing is more efficient than existing approaches to GUI testing. However, this is a very large question, and could conceivably take an entire thesis of its own to answer, so the purpose

of this investigation is included simply to see if, in a small data set, rule-based testing is always or never more efficient than creating equivalent tests with the traditional method.

1.5 Goals

There are two main goals of this thesis. The first is the development of a tool that can support manual exploratory testing of GUI-based applications with automated rule-based verifications. This is discussed in Chapter 3, which discusses LEET, the tool created for this thesis to make rule-based exploratory testing possible.

The second goal is to attempt to answer the research questions listed above in order to come up with a more focused list of recommendations for further work on rule-based exploratory testing. If future work is to prove whether this method is more effective than existing methods, it should be possible after attempting to answer each research question to come up with a concrete hypothesis for use in future evaluations.

1.6 Conclusion

In this chapter, an approach to testing applications through their GUIs by enhancing exploratory testing with rule-based verification is presented. The structure of GUIs was explained. Basic terminology for testing was presented in order to better contextualize rule-based exploratory testing. Manual and automated testing techniques are evaluated, and their strengths and weaknesses are explained so that it can be seen how adding rule-based verifications to exploratory testing can improve the process of GUI testing. The difficulties involved with GUI testing were explained in detail so that the benefits of enhancing exploratory testing with rule-based verification could be clarified. Finally, research questions to investigate and goals of this thesis were stated.

Chapter Two: Related Work

This section presents an overview of experiences with, techniques for, and tools that support testing an application through its GUI. There have been many attempts in academia to deal with the problem of GUI testing, and these focus in large part on an automated approach. Because of this, the following section will deal with publications by area of contribution, rather than individually. Contributions are grouped into those that deal primarily with complexity (Section 2.1), those that deal with primarily with change (Section 2.2), and those that deal primarily with verification (Section 2.3) in GUI testing. These areas of contribution are further subdivided where appropriate in the following subsections.

2.1 Dealing with Complexity

One of the three major challenges involved in GUI testing is dealing with the overwhelming complexity of the GUI itself, as originally introduced in Section 1.3.1. In the following subsections, various approaches to reducing the amount of complexity involved in GUI testing are summarized.

2.1.1 Avoiding the GUI

The simplest approach to testing a GUI-based application is to ignore its GUI. In this subsection, several ways of testing a GUI-based application while minimizing the focus that must be placed on its GUI are summarized.

2.1.1.1 Testing Around the GUI

The first option for avoiding the complexity associated with GUI testing is to circumvent the GUI. This can be achieved by adhering to a software design pattern along the lines of the Mode-View-Controller pattern, in which the GUI code is kept too simple to possibly

fail [20] [21]. It is also possible to design a specific testing interface through which to interact with the underlying application without the additional complications of the GUI [18]. It is worth noting, however, that neither of these approaches will uncover errors that, in spite of the design process, occur in GUI code. However, it is possible that accessibility frameworks, as described in Section 1.3.3, interact with the GUI in a realistic-enough fashion that they can be used as a testing interface for GUI-based applications.

2.1.1.2 Symbolic Execution

It is possible to use *symbolic execution* of a model of a GUI to derive optimal inputs to use during testing [22]. Symbolic execution is a way of figuring out which inputs will result in a specific path through a system being taken, and this process can be automated. For instance, out of the many possible inputs to a text box that takes a given number of characters, it's highly likely that only a small number will trigger interesting or different system behaviour [22]. By combining symbolic execution with a white-box approach to test execution, it's possible to identify a set of inputs that will statistically exercise more of the code related to a widget than would be probable by selecting input values randomly. This technique can be used to drastically reduce the number of test procedures needed to test a section of a GUI, as only the most interesting input is used. However, this technique has only been shown to be applicable to text input to GUI widgets, and has only been shown to be able to identify high-level errors such as uncaught exceptions [22]. In other words, it has only been used to expose a small subset of possible bugs.

2.1.2 Automated Generation of Tests

It is possible to use AI systems to automatically generate test suites for GUI-based application. These test suites tend to have *weak* test oracles despite having comprehensive test procedures. A weak test oracle is one that has a low probability of detecting a fault even if one is triggered. While it is possible to automatically generate test procedures for a GUI, it is very difficult to automatically generate meaningful test oracles. This subsection summarizes attempts to automatically generate test oracles capable of detecting faults in user interfaces.

2.1.2.1 Automated Planning

Automated planning makes use of AI systems to automatically generate a path between a given initial state in a GUI and a given goal state. This is useful in that if it is not possible to reach the goal state, then it can be inferred that the GUI is broken. While this kind of oracle requires no additional effort to create, it is very weak for detecting bugs. An obvious problem with these systems is that they rely on a human tester to define the expected behaviour of each of the widgets used in an application's GUI for use in the planning process – a tedious task [23] [24].

2.1.2.2 Evolutionary Testing

In *evolutionary computation,* a population is created by generating a set of algorithms based on a template. Algorithms generated in this way are similar to the template used to create them, but differ in small, randomized ways. This population is evaluated based on a set of fitness criteria, and a new population is generated using the fittest algorithms in the current population as the template. This process can be done iteratively in order to produce successively fitter populations.

Similarly, in *evolutionary testing*, populations are composed of test cases, and the fitness function used is based on factors like the number of bugs detected by the test, the amount of the system that the test covers, and so on. It should be noted that, in this work, the possibility that a bug report could be wrong is not considered. Using this sort of fitness function for evaluating and evolving test cases makes it is possible to increase test coverage in areas of the GUI that have been shown to include faults previously, and at the same time to weed out infeasible test cases – test cases that are not executable [25] [26]. This approach has also been applied to interactions themselves by prioritizing those that have led to faults in previous generations of tests [27].

While evolutionary testing makes it possible to increase the amount of an application that a test covers by using this axis as part of the fitness function, it is difficult to pair these procedures with meaningful test oracles. Oracles that check for faults like crashing programs and uncaught exceptions can be used, but oracles that verify the correctness of an application's functionality cannot be generated automatically, and are not have not been used in this method of testing. As with rule-based exploratory testing, it is possible to define strong oracles manually and then incorporate them into the testing process, but this has not yet been done with the fitness functions used in evolutionary GUI testing.

2.1.2.3 Randomized Interaction

Another solution to the complexity of testing a GUI-based application is to simply interact with its GUI randomly [28] [29]. Interactions can be chosen at random from those available at each state of the GUI, meaning that no formal definition of a test is actually recorded. Additionally, if each sequence of interactions is recorded, it is possible to generate a test suite made up of tests that increase overall code coverage. If two

sequences cover the same sections of the system, then it is possible also to select the shorter of the two for inclusion in the test suite [28]. In this way, it is possible to generate and refine a suite of tests geared toward increasing test coverage.

However, again, this approach is constrained by the difficulty of detecting faults through automatically-generated tests. While it is entirely possible to detect application crashes, uncaught exceptions, and the like, it is much more difficult to use this approach to verify that the correctness of the application under test's response to a specific interaction.

2.1.3 Model-Based Testing

One approach to simplifying the testing of a system is to first create a model of the system itself. Creating such models is a tedious process, but tools for automating it do exist. After a model has been created, it can be used to simplify the process of using one of the approaches to automated test case generation from Section 2.1.2. However, the basic issue with model-based testing is that these models are linked directly to the complexity of the system under test, which, in the case of GUI-based applications, means the models themselves can be quite complex. Further, if model-based testing simply confirms that a GUI conforms to its model, then only verification is being done, and validation – ensuring that a system meets its intended requirements – is omitted. In this section, the several types of models that have been applied to the testing of GUI-based applications are explained.

Model-based testing has several advantages, including the ability to select and optimize test cases intelligently and the ability to thoroughly test a system as represented in its model. However, test suites generated from models can be too large to run in a timely manner [9].

2.1.3.1 Finite State Machines

It has always been tempting to try to formally specify software applications in terms of *finite state machines*, or *finite state automata* (FSA). FSA can be used to represent an application as a set of states and interactions that cause transitions between states. This makes it possible to use FSA in the automated generation of test cases. The problem with using this approach on a GUI-based application is that a FSA representation of a GUI will have a enough states and transitions that creating and running tests from the FSA will take a prohibitively long time. Many of these states will not be detectable through an application's GUI, and the effect of many transitions will be difficult to determine. Moreover, FSA need to be entirely re-generated in the event of changes to the GUI [30]. Because of these factors, FSA remain difficult to apply effectively to testing applications through their GUIs.

However, some specialized forms of FSA have been used for GUI testing with some success. Variable finite state machines (VFSM) are FSA with additional global variables, which allow for the same state to respond to the same input in different ways based on previous input [31]. This makes it possible to decrease the number of states and transitions that must be included in the VFSM drastically [31]. While VFSMs are a significant improvement over FSA, they are still vulnerable to the same problems mentioned above.

2.1.3.2 Event-Flow Graphs

Event-flow graphs (EFGs), sometimes known as *event-sequence graphs* [32] or *complete interaction sequences* [33], are forms of FSA that can be used to model all possible sequences of interactions that can be executed on a specific GUI [1] [34] [35] [36]. This

is accomplished by modeling only interactions that cause transitions to take place, and states are represented implicitly. Because the number of such transitions is enormous even for relatively simple GUI-based applications, and because different windows in an application can represent independent arenas of interaction, EFGs can be simplified by breaking the main graph down into subgraphs representing individual windows [32] [35]. Paths through these EFGs can be used as test procedures, but the effectiveness of test oracles generated in this fashion is limited. Tests generated from EFGs can only be used to detect differences between versions of the same GUI, including unexpected crashes, and carry the risk of flooding developers with false positives, especially given the enormous number of tests that can be generated from EFGs.

2.1.3.3 Event-Interaction Graphs

Event-interaction graphs (EIGs) are a refinement of EFGs that focus on GUI events that trigger events in the underlying code of the application [34] [7]. EIGs model the ways in which the underlying application can be manipulated so that AI planning techniques can be used to automatically generate test cases from the model that will primarily test the underlying application code, rather than events provided by the framework used to write the GUI.

While it is possible to generate tests that are capable of detecting bugs in open-source systems using this technique [7], these automatically-generated tests tend to be much longer than necessary. In fact, it is both possible and advisable to pare these tests down to what's known as a *minimal effective event context* (MEEC) – the shortest sequence of events capable of triggering the fault – before adding them into a regression suite, as running the suite of generated tests can take a significant amount of time [7].

2.2 Dealing with Change

The second major challenge involved in GUI testing is the likelihood that the GUI will change repeatedly over the course of development, and the likelihood that tests developed against the previous version of the GUI will no longer work when run against the current GUI. This topic is discussed in more depth in Section 1.3.3. The basic problem is that an altered GUI can result in a false positive stemming from the procedure of a test. This means that it was not possible to run the test properly, and should usually be interpreted as a broken test. This is distinct from finding a bug, which would result in a failure stemming from the an inconsistency between the test oracle and the running system. While it is possible to repair broken test procedures automatically, determining whether its test oracle is still valid in its new context requires human intervention. The following subsections summarize attempts that have been made to make GUI test suites robust enough to deal with failures in test procedures caused by changes to a GUI, or to ease their repair when tests do break.

2.2.1 Testing Around the Interface

As mentioned in Section 1.3.3, one of the ways of avoiding the complexity of testing a GUI-based application is to create a test harness through which tests can be run. This harness will provide an interface through which widgets can be accessed. It can be used to test whether, if widgets are calling the correct events in the code-behind, the application will respond correctly. However, since testing harnesses skip the GUI entirely, they are of course unable to find GUI bugs.

Accessibility frameworks, like those mentioned in Section 1.3.3, carry the advantages of a test harness and are, at the same time, able to test GUI code as well as underlying application code.

Accessibility tools interact with the GUI in ways that make them usable as test harnesses. For example, the Windows Automation API and Microsoft Active Accessibility frameworks are built-in to GUI-based applications programmed using Windows Presentation Framework or Windows Forms [13] [12]. Since an accessibility framework used in this fashion is tightly integrated with the GUI itself and will call methods in and report events raised from these widgets, testing through an accessibility framework is analogous to testing the GUI itself, while avoiding the complexities involved in out-of-process testing – such as locating a widget to test and invoking its functionality.

Testing tools that are based on accessibility frameworks have the added advantage of not needing to know the specific class of a widget when a test is run. Rather, the basic functionality of a widget is used instead. In the Automation API, buttons, hyperlinks, drop-down menus, and many other widgets are categorized as "InvokePattern" objects. This means that a test that involved performing an invocation of a button would still work if that button were changed to a hyperlink.

Because of the potential of accessibility technology to assist in testing of GUI-based applications, it's no surprise that many new GUI testing approaches described in academic literature are starting to make use of this technology [37] [38] [39] [40] [41]. LEET, the implementation of rule-based exploratory testing presented in this thesis, makes use of the Automation API in order to take advantage of this added stability.

2.2.2 Prototyping

Approaches to test-driven development of GUI-based applications based around extensive user interface prototyping have sought to minimize change in the GUI once development has started in order to provide a more stable target for GUI tests [37] [38]. In these systems, a low-fidelity prototype of the GUI is created, and usability testing is iteratively performed on this prototype. This helps to discover and fix usability flaws in the GUI before actual GUI development begins. These usability flaws include issues which prevent users from actually using the system, and are likely to result in changes to the structure of the GUI. By dealing with usability flaws before coding begins on the actual GUI, it is less likely that the GUI will need to be changed during development. Tests can then be recorded from the final version of the prototype. As yet, this approach is the only one to focus on reducing the need for change, rather than on minimizing the impact of change. While this approach is promising, it has yet to be supported through any case studies.

2.2.3 Repairing Broken Test Procedures

When GUI test suites break, it is possible in some cases to repair the broken tests without human involvement, or to assist in manual repair. While the automated approaches to test repair are intriguing, human judgement is currently required in order to determine if a test oracle makes sense in the context of a revised GUI or repaired test procedure. Making this decision could prove more difficult than simply re-writing the entire test [16].

2.2.3.1 Approaches Based on Compiler Theory

One approach to the repair of broken test scripts is based around an error-recovery technique used in compilers. A broken test procedure can be seen as a sequence of events which is illegal for a given GUI, so by inserting events from the new GUI into the test procedure or by skipping events from the old procedure within the new one, it is sometimes possible to return the procedure to a usable state [15] [14] [42]. Due to the nature of this approach, it is entirely possible that there could be multiple ways to repair a single broken test procedure. This means that it can be used to automatically increase coverage of the application under test through these new versions of the same tests.

2.2.3.2 Assisting Manual Repair

Rather than attempting to automatically repair broken test scripts, it is also possible to automatically determine where changes in a GUI are likely to have broken test procedures, and to support manual test repair. The first step in this process is to determine which interactions in the test procedure refer to widgets that have been altered, and to determine the impact of their alteration on the procedure as a whole [40]. The impact of changing a widget between similar types, for example a RadioButton and a CheckBox, might be small in the context of a specific procedure, whereas removing a widget entirely would have a large impact within tests where the widget is referenced.

One of the difficulties with this approach is determining why, specifically, a change in the script is likely to break a test. It might not be obvious in the previous example why changing a widget from a RadioButton to a CheckBox should cause the test oracle to report a failure when both are accessed in the same way within the test. In fact, through the Automation API, selecting either would be accomplished through a

"SelectionPattern" object's "select" command. In order to better determine how a change in the GUI has broken a test script, it may help to automatically type all references to widgets used in the script [41]. This means that a tester would have access to the programmatic type of objects with respect to the language in which they are coded, rather than simply the language in which the test script is accessing them.

Another approach to simplifying the repair of broken GUI tests is to compose *macros*, or groupings of commands intended to make repeatedly performing the same set of interactions easier, out of sets of simpler interactions [43]. If a sequence of interactions is repeated across multiple tests, combining these interactions into a single macro can localize potential failures. The set of these macros can be separated from test scripts, and maintained independently. If a part of the macro breaks due to changes in the GUI, the fix simply needs to be applied to a single location in the script.

These approaches to dealing with problems involved in GUI testing are unique in that they attempt to keep test engineers involved in the testing process, rather than to automate all GUI testing activities. While these are promising new approaches, they lack evaluations that would be able to show if the approach is useful in practice.

2.2.4 Actionable Knowledge Models

Actionable knowledge models (AKM) are composed of actionable knowledge graphs (AKGs) – yet another form of FSA – and sub-goal lists (SGLs) [42]. Each SGL represents a node or set of nodes within an AKG, and can be envisioned as a test procedure. By using AI techniques to transition a GUI-based application between vertices its corresponding AKG with the goal of reaching states of the application corresponding to vertices stored in SGLs, it is possible to store test cases within the

abstract structure of an AKM. The advantage of doing so is that no test script is ever generated. This means that, when the GUI changes, these changes need merely be incorporated into the AKM, and in so doing will be propagated to every test for the system simultaneously. However, because this approach is based on the use of a FSA, it falls prey to the issues discussed in Section 2.1.3.

2.3 Verifying Correct GUI Behaviour

The final major challenge involved in GUI testing is the difficulty of creating strong test oracles. This subsection presents various academic approaches to this issue, from lightweight approaches that use weak oracles to approaches that attempt to systematically model the behaviour of an entire GUI-based application. It is interesting to note that verification is the least-well addressed of the three major challenges involved testing an application through its GUI in academic literature.

2.3.1 Smoke Testing

The majority of publications which address the problem of GUI testing focus on only half of the problem. Approaches to automatically creating GUI test procedures are discussed, but automated creation of effective GUI test oracles is not. Tests that either possess very simple oracles or lack them entirely are known as "smoke tests." Smoke tests aim to see whether the basic functionality of the system is present; in the case of GUI-based applications, smoke tests address questions such as "does the system crash during interactions," "do widgets react to interaction," and "does the system ever throw uncaught exceptions."

While this sort of testing is useful in that it can verify that it is possible to reach certain states in an application after it has been modified, it's necessary to add additional

verification information to these tests before it would be possible to assess whether the system is behaving as expected, instead of testing to ensure the system is not behaving as not expected. With weak oracles like these, GUI tests lose much of their ability to detect faults [44].

2.3.2 Verification Based on the Previous Build

When GUI testing is performed in an environment in which *continuous integration* is practiced, an interesting approach to oracle generation can be taken. Continuous integration is the practice of requiring tests for code to be accepted as part of a program, and of running all of these tests before a modification to the application is accepted. When continuous integration is used as part of a software development process, it is possible to assume that the previous version of the build is correct because it passed the previous run of tests in order to be accepted. Each of these test runs can then simply verify that the current build behaves in the same way as the previous one [34].

In order for such an approach to be useful, it would be necessary for the GUI-based application that's being tested to respond in a hard-coded correct fashion to each test until each bit of functionality is implemented. Additionally, such a system would treat any change to the GUI as an error, and would thus be likely to return a number of failing tests each build unless the GUI is exceptionally stable. Because of these limitations, this approach is only really practical for mature applications to which only maintenance work is being performed.

2.3.3 Model-Based Verification

In principle, it might be possible to use a detailed FSM, or some other model of an application, to describe all of the possible states of a GUI, and thereby to deduce the state

that should result from a given interaction. Though this has been attempted in the past [45], the overwhelming complexity of modern GUIs is making this approach increasingly less feasible. In fact, recent models, such as the EIG described in Section 2.1.3.3, abstract the state of the GUI entirely and focus only on possible events as a way of reducing complexity.

2.3.4 Rule-Based Verification

One approach to easing the creation of complex GUI test oracles is to define desired behaviour in terms of rules, like those discussed in Section 1.2. It is then possible, after each step of the test procedure, to verify each rule against the running system.

This approach has been used in the past to validate each state of an AJAX web interface [46]. In this system, defining specific warnings and errors in the HTML or DOM of the application in terms of rules presents a huge advantage, as they can simply be stored in a rule base that is queried repeatedly during test execution. Since the test procedure of an AJAX application can be easily automated using a web crawler, all that really needs to be done in order to perform automated testing is to define each rule that the system should ensure. Unfortunately, defining rules that perform validation only and are useful enough to aid in testing remains difficult.

This previous approach was limited to AJAX applications only, and as such represents a proof-of-concept on a simplified GUI system. However, a similar technique has been applied to GUI-based applications, in which events are defined as a set of preconditions and effects [23]. This technique is used primarily for automated creation of GUI test cases, but has the additional effect of verifying that the effects of each atomic interaction are as expected for a given widget.

The value of each of these approaches is that expected or unexpected states of the GUI are stored in terms of a reusable rule. This means that it is possible to ensure that a specific error does not occur during the execution of a large number of separate tests. Because of this ability to test for the same properties across a long number of states, rule-based testing plays a crucial role in LEET, as can be seen in Chapters 3 and 5.

2.4 Conclusion

In this section, academic publications on the topic of GUI testing were sorted into three categories based on their main contributions: those that helped deal with the complexity of modern GUIs; those that helped deal with changes in the GUI that would break tests; and those that dealt with the difficulty of creating automated test oracles for GUI-based applications. While work has been done to explore different methods of testing applications through their GUIs, no publications address the issue, originally introduced in the introduction to this thesis, of integrating exploratory testing with automated testing methods. Further, while rule-based testing has been briefly applied to GUI-based applications previously, LEET represents the first attempt to integrate manual exploratory testing with automated rule-based testing, or any other form of automated testing, and to apply this union to GUI testing.

Chapter Three: LEET

In Section 1, three major challenges were addressed: complexity, change, and verification. These three difficulties must be addressed by any GUI testing tool if it is to be successful. First, a tool must be able to deal with the overwhelming complexity of a GUI-based application: the vast number of states that it can enter and the number of events that can be triggered in order to move the application between states must be reduced to a manageable subset in order for tests to be able to run within a reasonable amount of time. Second, a tool must be able to deal with the rapid changes that GUIs undergo during development. Broken test procedures, resulting from changes to the GUI, should be minimized without obscuring the results of failing test procedures, which result from changes to the functionality of the application. Third, a tool must find a way to make use of strong, automated test oracles.

This chapter describes the design of LEET Enhances Exploratory Testing (LEET), the implementation of the combination of exploratory and rule-based testing created to determine whether this approach to testing GUI-based applications is practical.

3.1 Structure of LEET

LEET is designed to support manual exploratory testing of GUI-based applications through the addition of automated rule-based verifications. Through exploratory testing, a human test engineer explores a path through an application, and this path is recorded by LEET as a script of automatically replayable interactions – a test procedure. In this way, a subset of the entire application is identified for further testing. If this path proves interesting – if bugs are found, or if the system behaves suspiciously in the tester's opinion – then the next step is taken. Rules are used to define either the behaviour of

bugs in the application or a potential failure stemming from the suspicious behaviour – test oracles, in other words. LEET can then replay the test procedure and query the set of rules at each step of the procedure. The preconditions of these rules, the "if..." clauses, keep them from firing unless certain conditions are met, which reduces the number of false failures – bug reports resulting from an inability to correctly perform a verification rather than a failure of expectations to match the application's behaviour. This decreases the chances that changes to the GUI will end up breaking tests. These rules are then queried throughout replay of the exploratory tests recorded earlier in order to ensure they are met throughout all of those states of the application, which results in a stronger test oracle. This procedure can be seen in Figure 7. Because users of LEET must be able to define rules through code, LEET's target audience is test engineers.

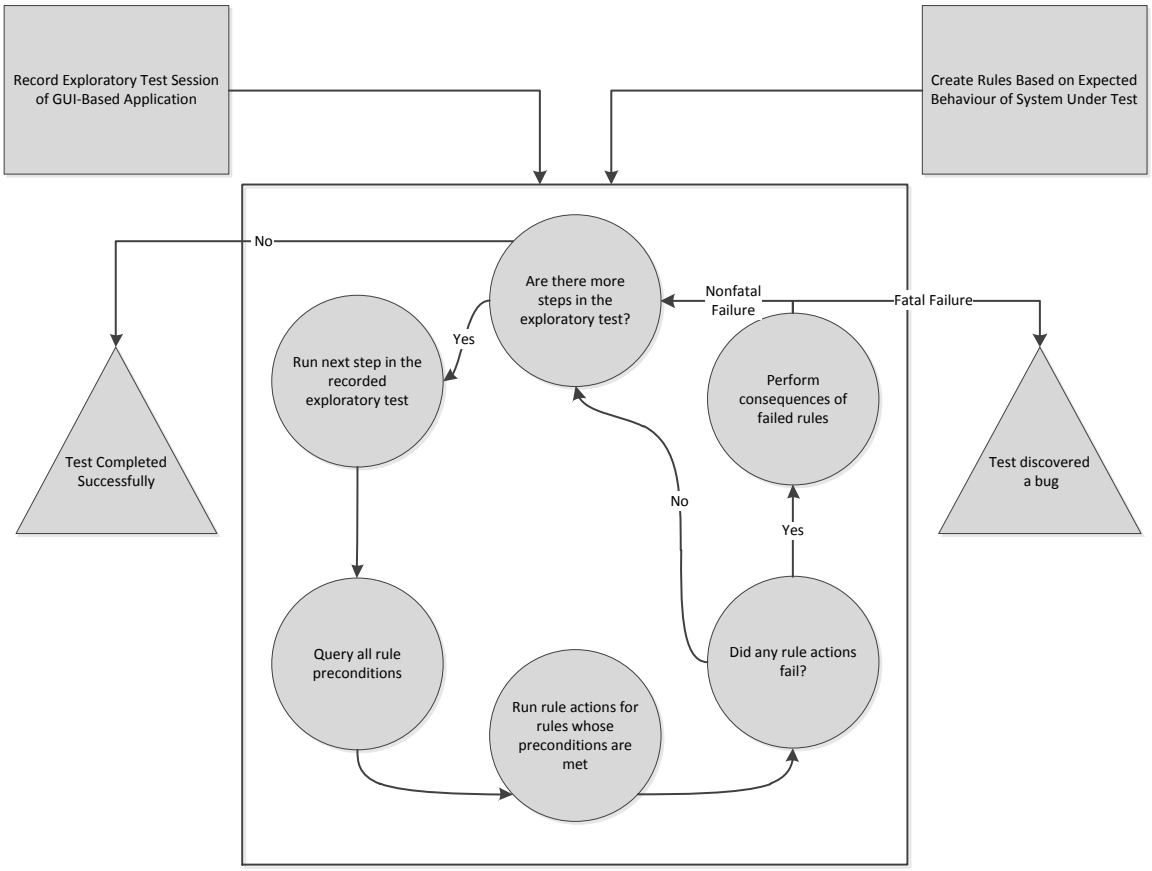

Figure 7: The process of rule-based exploratory testing.

Many different features are required in order for automated rule-based testing to complement manual exploratory testing. Figure 8 provides an overview of these different subsystems and how each interacts with a GUI-based application can be seen in.

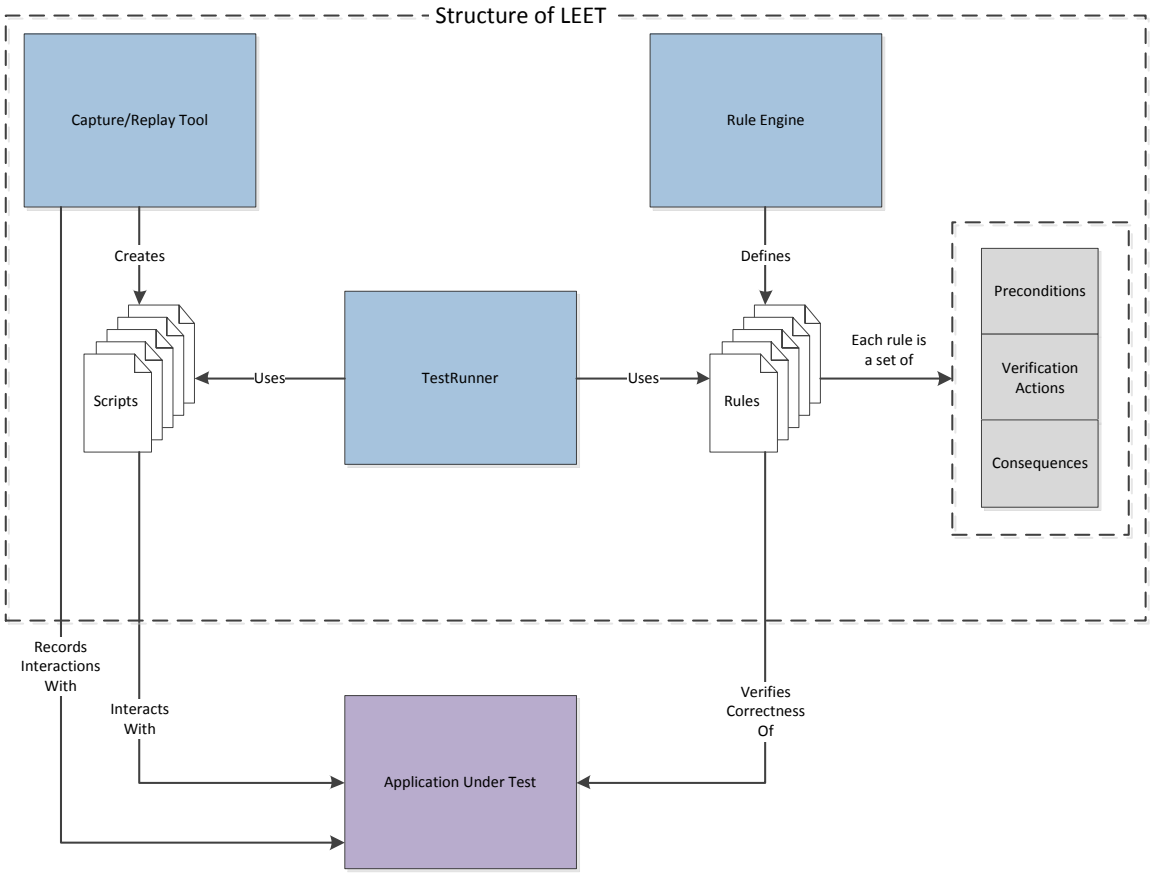

Figure 8: Diagram showing the structure of LEET

First, it must be possible to record interactions that a test engineer takes on an application. This is accomplished in LEET through the use of a *Capture/Replay Tool* (CRT). CRTs record interactions taken by a user through recording the positions of mouse clicks and keystrokes or, alternately, through recording the events raised by an accessibility framework. These interactions are recorded as a script, which can be translated to executable code to be replayed. Because CRTs are able to record exploratory test sessions, they are central to any implementation that seeks to enhance exploratory testing with automated verifications. Currently, events from only a handful

of AutomationPatterns are recorded by LEET. LEET can interact with widgets that expose InvokePattern, ValuePattern, ExpandCollapsePattern, TransformPattern, TogglePattern, TextPattern, SelectionPattern, SelectionItemPattern, and WindowPattern AutomationPatterns. However, this still leaves DockPattern, GridPattern, GridItemPattern, MultipleViewPattern, RangeValuePattern, ScrollPattern, ScrollItemPattern, TablePattern, and TableItemPattern, to be implemented in future work. In LEET, the CRT subsystem records interactions raised through the Automation API in a domain-specific language before compiling them into executable C# tests using the Compiler subsystem. This allows test engineers the ability to edit tests in whichever of these formats they find more easier to use. Compiled tests then use the Automation API to replay the previously recorded test on the application. Figure 9 shows LEET, top, in the process of recording a test session taking place on the sample application, bottom.

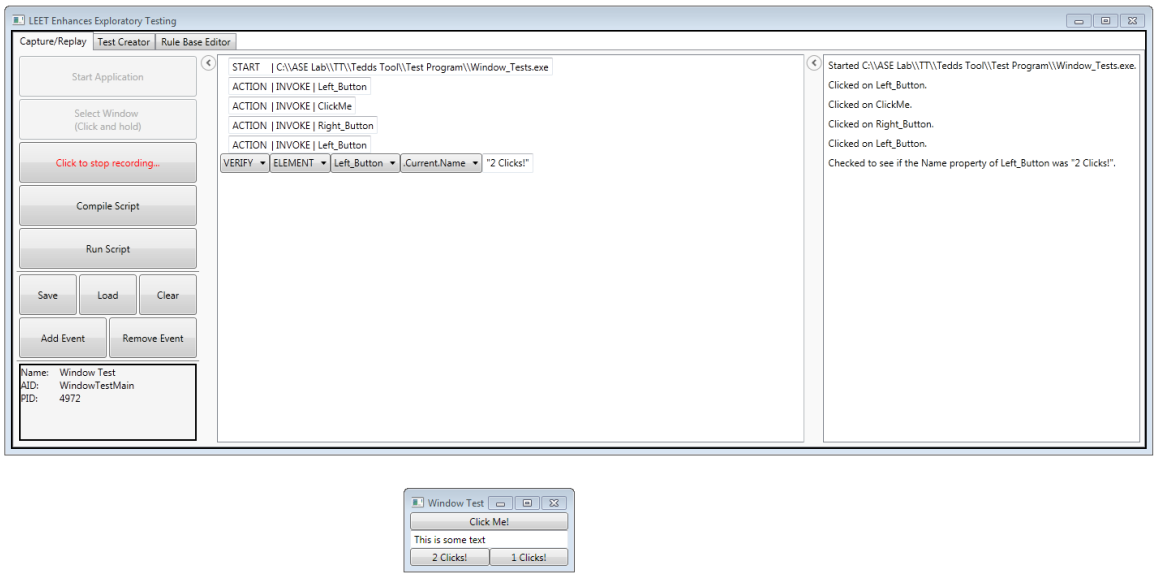

Figure 9: Recording an exploratory test session with LEET

Second, it must be possible to create rules (Section 1.2.2.1). This involves a system for checking preconditions against a running application, deciding if they are met, running the action of a rule, and determining the consequence of failure. In LEET, this is done through the Rule-Base subsystem. The Rule-Base contains a set of rules, each of which is a set of preconditions that are used to determine when a rule should take an action, actions that are used to verify parts of an application under test, and consequences failing the verification process. Preconditions and actions are checked against the running application via the Automation API. Consequences can be either fatal or nonfatal. Fatal consequences end a test run with an error, which signifies a bug in the application, while nonfatal consequences display an error dialog so that human test engineers can be made aware of a minor discrepancy and use their judgement as to whether to report it as an error.

Third, a system is needed in order to pair a recorded exploratory test session with a rule-base. This is necessary so that the automated test oracle can be applied to the running GUI-based application after every step of the test procedure, which expands the number of verifications that are performed in each state, thus increasing the strength of the test. In LEET, this is done through the TestRunner subsystem, which runs a single step from a test procedure at a time, then triggers the Rule-Base. In Figure 7, above, the rectangle enclosing the various circles is the set of actions initiated by a TestRunner to perform rule-based exploratory testing. The Rule-Base then checks all the preconditions of all of its rules against the running system, and triggers the actions of rules which have met all of their preconditions. This is a modified version of the Rete algorithm [6], an efficient method of pairing facts with productions in logic-based systems. While other methods of

pairing preconditions to actions to take exist, this system was chosen for use in LEET to increase the efficiency of querying the rule-base since there was concern during LEET's development about the efficiency of querying the rule-base. Finally, any consequences that are raised as the result of a failed rule action are dealt with, and, if any consequences are fatal, the test is terminated. Otherwise, the TestRunner repeats this process until there are no more steps in the test procedure.

3.2 Conclusions

This section described the design of LEET. The motivations for enhancing exploratory testing with rule-based testing were reiterated, and the benefits of the specific design implemented in LEET were explained. The process of recording an exploratory test was described, the process of creating rules was described, and the method of combining and running these together was explained. Specific interesting design decisions were explained. In this way, the necessity of each subsystem is made clear, and the reason that each specific subsystem was implemented in LEET is defended.

Chapter Four: Technical Challenges

Several technical challenges were encountered during the development of LEET. These challenges, along with the solutions that were decided upon, are described in this chapter in order to provide guidance for the development of alternate implementations.

4.1 Interacting with Widgets

From a test's point of view, three forms of interaction are required in order to test a GUI-based application:

1) Finding a specific widget
2) Invoking that widget's functionality
3) Verifying properties of that widget

The first two requirements were discussed in Section 1.3.3. The third requirement is the most challenging: how do automated test oracles determine whether the system is working correctly or not? The earliest approach to this problem was to capture screenshots of an application's GUI when the application was running correctly, and to then compare the application's GUI during the execution of a test to this screenshot. This approach, however, is very likely to result in false reports of test failures due to minor changes to the interface when the functionality of the application itself was working correctly. An accessibility framework, on the other hand, can provide information about the properties of widgets during the execution of a test, so determining the correctness of a GUI with an automated test oracle is much easier. On the one hand, the oracles will be more precise: it is possible to address only a small part of a GUI with a test oracle so that minor, unrelated changes to the interface can be ignored. On the other hand, specific information about a widget can be accessed, rather than just the way in which it is

rendered onscreen. Due to these advantages of testing a GUI-based application with the aid of an accessibility framework, this is the approach taken in LEET.

The accessibility framework that LEET makes use of is the Windows Automation API [13]. The Automation API works by providing a public interface for each widget based on which *AutomationPatterns* it implements. An AutomationPattern allows access to a widget based on the functionality of the widget. For example, buttons and hyperlinks have the same basic functionality: they receive mouse clicks and respond by performing a single, unambiguous function. An AutomationPattern encapsulating this functionality, then, would allow access to a widget through the invocation of a single method. This method would perform the equivalent of a mouse click, and thus trigger the basic functionality of the widget. By implementing a set of AutomationPatterns describing different parts of the functionality of a widget, its complete behaviour can be defined for use by other programs, including tests. A diagram of this interaction is shown in Figure 6, from Chapter 1.

“InvokePattern” and “ValuePattern” are two of the patterns that may be implemented by a widget, and each represents a part of the functionality that the widget provides.

Each widget will expose some default properties and methods through UIA, in addition to other properties specific to each AutomationPattern it implements. Some of these properties are implemented automatically. For example, the “AutomationID” property will be the same as the name defined for a widget created in WPF. AutomationPatterns for custom controls, however, must be implemented manually by developers. This information is used to locate specific widgets during the execution of a test. Unfortunately, it is possible that widgets can end up without AutomationIDs or other

identifying information. It is difficult or impossible to test properties of such anonymous widgets. For details on the frequency of such widgets, see Section 5.2.3.

4.2 Keyword-Based Identification

Two ways to identify widgets during the course of a test when using an accessibility framework were identified in Section 1.3.3. The first, testing with object maps, is robust in that it can still find widgets even when part of the information they should match has become obsolete, as can be the case when changes are made to the GUI. However, in order to gain this robustness, it's necessary to specify many different properties of the desired widget. In keyword-based testing, on the other hand, the value of a single property is used to identify a widget. This means that the values of this property cannot be the same between any two widgets in the application. While keyword-based testing is simpler for a human to write than testing with object maps, it can be difficult to ensure that all widgets in a complex GUI have different unique identifiers – something not required in testing with object maps. However, because of the simplicity of using keyword-based testing given that an application's widgets are uniquely identified, keyword-based testing was used in LEET. See the pilot evaluations in Chapter 5 for evaluation discussion of the weaknesses of this approach.

Widgets expose several default fields through the Automation API. The "AutomationID" field is particularly important in that it should be unique. Additionally, the Automation API can be used to locate widgets based on simple search criterion. For example, it is possible to search for children of a given window that have a given AutomationID.

In theory, it should be possible to use the AutomationIDs of widgets in GUIs for keyword-based identification. In practice, however, the AutomationID property is often

left blank by developers. For this reason, LEET will attempt to match the AutomationID of a widget first when searching for a specific widget, and then attempt find a matching "Name" field if that is not possible. While this sort of search is not guaranteed to find the correct widget, it is necessary in light of the prevalence of widgets without AutomationIDs.

4.3 Code Coverage Integration

One of the difficulties mentioned in Section 1.3.3 is determining what parts of a system have been covered during exploratory testing. This lack of coverage information makes it difficult to determine how adequately the system has been tested, or which parts of the system need further attention. The most commonly-used way of recording this information is to use a *code coverage tool*. Code coverage tools monitor the execution of an application in order to determine which lines of code were executed and which were not. This makes it possible to evaluate which parts of the system have been adequately tested and which require further scrutiny.

Since the approach to GUI testing used in this thesis relies on exploratory testing, it must also deal with this difficulty. LEET integrates with NCover 1.5.8 in order to make it possible to collect code coverage information [47]. This specific version of the tool was used because it was the last free, open-source version produced, and is distributed freely as part of TestDriven.NET [48].

4.4 Technical Limitations

Several of the issues encountered during the development of LEET could not be resolved, and remain outstanding issues. These technical limitations are described in this subsection so that the underlying issues can be understood, and future attempts to

implement a system of enhancing exploratory testing with rule-based verifications can take these issues into account in their design.

First, since tests are recorded through the Automation API, only widgets that implement at least one AutomationPattern are testable. While widgets provided by the Windows Presentation Framework and WinForms toolkit will implement appropriate patterns by default, it is necessary to manually define patterns for custom widgets. If this is not done, it will not be possible to record tests from or replay tests on the application. When using LEET to test web-based applications, this is compounded based on the browser used to display the application. AutomationPatterns will be somewhat available when a page is viewed in Internet Explorer, for example, when the same page displays no widgets when viewed in Google Chrome. Additionally, some functionality is not testable through the Automation API because the AutomationPatterns to describe this functionality do not exist. For example, there are currently no AutomationPatterns designed to support interactions that can take place on a digital tabletop, including interactions with ink canvases and gestures. This makes these interactions untestable through the Automation API at present and, consequently, untestable through LEET.

Second, because LEET uses keyword-based identification of widgets, it's necessary for each widget to be assigned an AutomationID or, at least, a Name. If the former is not done, it's possible that the wrong widget may be found and used by LEET. If neither is done, LEET will not be able to test that element. For an evaluation of how often this happens in certain systems, see Section 5.2.3. It is possible that the problems this may cause would be lessened if testing with object maps were used to locate widgets for

testing rather than keyword-based testing, so this may be an attractive approach for alternative implementations.

Third, the use version 1.5.8 of NCover engenders a distinct weakness: it only works with applications programmed in the .NET framework, version 3.5 and earlier. New applications programmed in .NET 4.0 will not be coverable and, worse, will cause NCover to crash. Further, due to the way NCover functions, the .PDB files created when an application is compiled must be present in the same directory as the executable for the application that's being tested. In order to fix this issue, all that would be necessary is to integrate with a different code coverage tool. Newer versions of NCover would be good candidates for this, though they require purchase.

Fourth, because LEET is programmed in .NET, it can only run on PCs. Further, because not all programming languages enable their widgets with AutomationPatterns, the amount of testing that can be performed on applications programmed in non-.NET languages is limited. For example, Java GUIs are partially enabled: it is possible to record certain interactions, like clicking a button, but replaying a test script on the GUI won't currently work. This is due to the fact that AutomationPatterns are only partially implemented in Java. Events are raised when widgets receive interactions, but the part of the AutomationPattern that receives events is not implemented, which makes it impossible to run a test on Java-based applications through the Automation API at present.

Fifth, it's not currently possible to record events taking place in more than one window at a time. This means that, when applications spawn additional windows, these applications must specifically list these windows as children of the application's main window. When

testing involves interactions between one or more application, it's necessary to switch LEET's focus between active windows. Future implementations of the approach to enhancing exploratory testing with rule-based verifications will need to be able to handle

4.5 Conclusions

In this chapter, technical challenges involved in the design and creation of LEET were introduced. Those challenges that it was possible to overcome were explained, and the solutions used in the implementation of LEET were stated. Those challenges that were not immediately addressable were also explained, and potential solutions were suggested.

Chapter Five: Preliminary Evaluation

This section presents a preliminary evaluation of LEET's implementation of a rule-based system of GUI testing. The questions addressed in this section are drawn from the list of research questions provided in Section 1.4. This preliminary evaluation is not intended to prove that the technique of enhancing exploratory manual testing with rule-based verifications is the solution to the challenge of GUI testing, or that this approach is better than other approaches. Rather, the preliminary evaluation is intended to show that this approach to GUI testing is novel and practical. In order to show that the approach is novel, LEET is compared to existing GUI testing tools in order to show that the functionality required to meet the first research goals of this thesis, described in Section 1.5, is not provided by currently-existing tools. In order to show that the approach is practical, four evaluations were conducted to determine whether the implementation of LEET has resulted in a tool that is practical for use in enhanced exploratory testing.

5.1 Comparison to Existing GUI Testing Tools

This section presents a set of existing GUI testing tools, and compares the major features of these tools to those found in LEET. The common features of these tools are enumerated, and a summary is provided.

In order to support exploratory testing with automated testing techniques, as argued for in [5], an exploratory GUI testing tool should provide capture/replay functionality. This allows the tool to record exploratory test sessions so that they can be enhanced with automated verifications for regression testing. Since the first research goal of this thesis, as outlined in Section 1.5, is to create a tool that can enhance exploratory testing with some form of automated testing, and capture/replay tools allow exploratory testing to be

recorded in a form that can be used as a test procedure, this feature is essential to successfully enhancing exploratory tests with rule-based verifications.

The second feature that a GUI testing tool that can be used to enhance exploratory testing with automated verifications should provide is keyword-based testing. Keyword-based testing is essential because it makes recorded exploratory test sessions maintainable. However, downsides of this approach when applied to rule-based verifications exist, and are explored in Section 5.2.

The third feature that a GUI testing tool should provide for the purposes of this thesis is the ability to use a rule-base as an automated test oracle. Without this, it would of course be impossible to enhance previously-recorded exploratory tests with rule-based verifications.

Three additional features that were encountered in existing GUI testing tools were also listed because it is possible that they could be used as a basis for future methods of enhancing exploratory testing, even though they are not essential to the approach described in this thesis. The first feature is *test abstraction*. Test abstraction is the ability to store test procedures or test oracles in an intermediate form. Tools that make use of test abstraction store tests in a high-level form, which is only mapped to discrete interactions with specific widgets when a test is run. The second additional feature that existing GUI tools may possess is the ability to provide assisted test maintenance. Assisted test maintenance is discussed in Section 2.2.3 and, while useful, is not essential to accomplishing the research goals outlines in Section 1.5. The third feature is automated test generation – the automatic creation of test procedures for a given GUI-based application, similar to the approaches discussed in Section 2.1.2. While this

feature could be useful in automated smoke testing, it does not currently leverage the advantages of exploratory testing. Even though these features are not central to the research goals described in this thesis, they are still powerful testing tools, and could be used as the basis for future attempts to enhance exploratory testing with rule-based verifications.

Table 1 shows each of the existing GUI testing tools that were evaluated for this thesis, as well as LEET, along with which of the features described above that they possess. Of the 24 tools compared in Table 1, all but 4 provide some form of CRT for automated recording of GUI test scripts. Of these 20 tools, only 5 provide support for keyword-based testing as well. Of these 5 tools that support recording of exploratory test sessions and support keyword-based testing, only one provides support for rule-based verifications – LEET. Only LEET provides all three features that are required in order to accomplish the research goals set out in Section 1.5.

Table 1 Major features of existing GUI testing applications.

	Capture/Replay Functionality	Keyword-Based Testing	Rule-Based Testing	Test Abstraction	Assisted Test Maintenance	Automated Test Generation
[49] Selenium	✓					
[50] WHITE	*	✓				
[51] Rational Functional Tester	✓	✓				
[52] Abbot	✓					
[53] AutoIt						
[54] Automation Anywhere	✓					
[55] Dogtail	✓					
[56] Eggplant	*				✓	
[57] GUIdancer		✓		✓		
[58] IcuTest						✓
[59] Linux Desktop Testing Project	✓					
[60] Phantom Test Driver	✓					
[61] QA Wizard pro	✓					
[62] Qaliber	*					
[63] QF-Test	✓					
[64] HP QuickTest Professional	✓	✓			✓	
[65] Ranorex	✓					
[66] RIATest	✓					
[67] SilkTest				✓		
[68] SWTBot	✓					
[69] Test Automation FX	✓					
[70] TestComplete	✓	✓				
[71] TestPartner	✓				✓	
[72] WindowTester	✓					
[73] LEET	✓	✓	✓			

✓	Feature met
*	Feature partially met

5.2 Preliminary Evaluations

Four evaluations were conducted in order to determine if the approach to enhancing exploratory manual testing with automated rule-based verifications as implemented in LEET is practical. The purpose of the first two evaluations is to show that rules are applicable to GUI-based testing, and can detect common weaknesses in GUI-based applications. The purpose of the third evaluation is to investigate how testable GUIs actually are when using keyword-based testing and automated rule-based verifications. The purpose of the fourth evaluation is to investigate how much effort it is to actually create tests for GUI-based applications by recording exploratory test sessions and adding rule-based verifications.

5.2.1 Can Rules Detect General Security Flaws in GUI-Based Applications?

In this part of the evaluation, the ability of rule-based exploratory testing to detect general, high-level bugs in GUI-based applications is explored. One such bug is described, and two automated rules that could be used to catch this bug are described. Three exploratory test sessions from three significantly different applications were recorded, and then paired with these rules. The number of violations of these rules are then described, and the implications of rule-based exploratory testing's ability to detect these violations are explored.

In some applications, widgets are initially created outside of the visible area of a screen. This means that, while the computer is actually going through the process of creating these offscreen widgets, users aren't able to see or interact with them. This is done because moving a GUI to a different position on the screen is a fast operation, once all of its widgets have been created. It can make a GUI-based application appear to run faster

than it actually does if its GUI is created offscreen initially, then copied to the visible area of the screen.

It is possible, however, to detect and interact with widgets even if they not displayed within the visible area of the screen. Tools like Microsoft's UIA Verify [74] can display different properties of widgets and invoke their functionality through the Automation API – even when they are offscreen. This means that care must be taken to ensure that an application's widgets do not perform their functionality in response to events until they are actually displayed onscreen.

As a hypothetical example of how this weakness could be exploited, imagine a situation in which options are added to an interface depending on whether a user logs in as a regular user or as an administrator. If these options are rendered offscreen initially and not copied to the visible area of the screen unless a user logs in as an administrator, they could still be detected and interacted with using a program like UIA Verify. Doing so would effectively bypass the authentication system entirely and allow non-administrators to access functionality that only administrators should be able to access.

Three applications that were compatible with LEET were selected for use in this section of the evaluation. In this case, a compatible application was defined as one for which it would be possible to record exploratory tests of the application's basic functionality using LEET. This turned out to be somewhat difficult, as many applications made heavy use of custom widgets. Since custom widgets are not based upon existing widgets in the Windows Presentation Framework or in WinForms, the AutomationPatterns accessed through the Automation API are not automatically implemented. In many of these applications, AutomationPatterns had not been implemented for these custom widgets,

and so, in large part, exploratory tests of the functionality of the applications could not be recorded with LEET. However, one application, Family.Show [75], was compatible with LEET, and is the first application used in this part of the preliminary evaluation. The second application, the Character Map application included with Windows 7, is an older application – it was included with Windows operating systems since at least 1993 – and is included to show that LEET can work on older Windows applications. The third application is the website for *Resident Evil 5* [76] , and it is used in this part of the evaluation to show that LEET can work with websites as well as standard GUIs. Using these three very different applications in this part of the preliminary evaluation has the added advantage of showing that rules created for LEET can work with various significantly different types of interfaces using the same rule.

First, two rules were created. These rules require widgets that are not displayed on screen correctly to be disabled – unable to respond to interaction. The first rule reports that the application being tested is broken when a widget is offscreen, but still responding to interaction. The second rule reports a problem when a widget is dimensionless – or contained within a 0-by-0 square on the screen – and still responding to interaction. These rules are defined through C# code, but a conceptual representation of them in a somewhat more readable format, similar to the domain-specific language in which LEET records exploratory test sessions, is shown in Figures 10 and 11.

In interpreting this conceptual representation, it might be helpful to point out that the result returned from a precondition determines if a rule's action should be taken, and the result of this action determines if a consequence is necessary. If anything besides "Null" is returned from a precondition, the following rule action will be taken, and if this action

returns anything besides “Null,” the following consequence will take place. Additionally, the assignment of the widget variable that takes place before preconditions can be used to input all widgets in the application under test, and in this manner to check a large number of widgets with the same rule.

```
<RulesBase.Rule>
  WIDGET := All
  <RulesBase.PreconditionInfo>
    CASE .IsEnabled AND .BoundingRectangleProperty == Rect.Empty
    TRUE? RETURN WIDGET
    FALSE? RETURN Null
  </RulesBase.PreconditionInfo>
  <RulesBase.RuleActionInfo>
    CASE .IsInvokePatternAvailableProperty
    TRUE? RETURN WIDGET
    FALSE? RETURN Null
  </RulesBase.RuleActionInfo>
  <RulesBase.ConsequenceInfo>
    FATAL True
    MESSAGE A dimensionless widget is enabled!
  </RulesBase.ConsequenceInfo>
</RulesBase.Rule>
```

Figure 10: Rule for detecting 0-by-0, enabled widgets

```
<RulesBase.Rule>
  WIDGET := All
  <RulesBase.PreconditionInfo>
    CASE .IsEnabled AND .isOffscreenProperty
    TRUE? RETURN WIDGET
    FALSE? RETURN Null
  </RulesBase.PreconditionInfo>
  <RulesBase.RuleActionInfo>
    CASE .IsInvokePatternAvailableProperty
    TRUE? RETURN WIDGET
    FALSE? RETURN Null
  </RulesBase.RuleActionInfo>
  <RulesBase.ConsequenceInfo>
    FATAL True
    MESSAGE A offscreen widget is enabled!
  </RulesBase.ConsequenceInfo>
</RulesBase.Rule>
```

Figure 11: Rule for detecting offscreen, enabled widgets

Next, exploratory test sessions in which some of the basic functionality of each of these three applications was tested were recorded using LEET. For example, the Family.Show application is a program for creating and updating a family tree. In the test for this part

of the preliminary evaluation, the basic functionality of adding family members to the tree was explored.

Finally, three TestRunner objects were created to combine each recorded exploratory test session with the two rules shown in Figures 10 and 11. Each TestRunner was run on the system for which its exploratory test session was recorded, and many violations of both rules were discovered. For the execution of each TestRunner, the maximum number of rule violations discovered after a single step in the procedure was recorded in Table 2. In other words, the minimum number of violations of each rule discovered during the run of each TestRunner was recorded, because there must be at least as many violations of the rule as the highest number of violations after each step of the test procedure. While it would have been preferable to list the total number of elements in violation of these rules throughout the execution of each test, this number is difficult to determine due to the number of anonymous widgets in each application – widgets that do not have values assigned to their AutomationID or Name fields. This problem is revisited in Section 5.2.3.

Table 2: Minimum number of erroneously enabled widgets in each test application

Application	Offscreen Widgets (Rule: Figure 10)	0-by-0 Widgets (Rule: Figure 11)
Character Map	306	0
Family.Show	913	73
Resident Evil 5 Website	3	4

In this section of the preliminary evaluation, it was shown that is possible for exploratory tests that have been enhanced with rule-based verifications to detect when a GUI's

widgets are in a state that could be used to lead to a breach of security. Further, by using three significantly different applications, the results show that it is possible to write rules that test for high-level errors and reuse these rules to find violations across a range of applications. Finally, the sheer number of violations detected – a minimum of 986 violations in Family.Show – implies that rules that test for high-level errors show good potential to detect a large number of violations.

This means that it should be possible to detect a large number of bugs using a small set of rules. This result implies that LEET's approach to enhancing exploratory test sessions with rule-based verifications is able to create strong test oracles – in other words, that it is possible to verify an application's functionality using this approach. Further, since rules can be created to test for high-level errors and then detect bugs when paired with recorded exploratory test sessions conducted on significantly different applications, these sorts of verifications should be resistant to changes in the GUI.

5.2.2 Can Rules Detect Specific Security Flaws in GUI-Based Applications?

In this part of the evaluation, the ability of rule-based exploratory testing to detect specific, low-level bugs in GUI-based applications was investigated. Low-level bugs, as opposed to the general bugs that were the focus of the previous part of this preliminary evaluation, result from specific failures in specific interfaces. The interface used as a focus in this part of the evaluation is a specific type of validation interface used in many web-based applications, an "age gate." Age gates are used to verify that a user is old enough to view the graphic content contained on the website. One bug that age gates are vulnerable to is described. First, a single rule was created based on a manual inspection of three of the seven websites that were selected for use. This rule utilized heuristics in

order to determine which widgets to interact with and whether or not the system had responded correctly. Exploratory test sessions were recorded for each of these websites, and the rule was paired with these recordings and run on each website. The changes to the heuristic that were necessary in order to make the rule function properly when used to test each new website are described. Finally, the implications of the results of this study are discussed.

The Common Weakness Enumeration (CWE) is a database of security vulnerabilities that have been encountered in the past. The bug used to explore this topic is based on CWE-358, "Improperly Implemented Security Check for Standard" [77]. This weakness arises when a security measure is implemented in such a way that it is possible for verification to succeed even when part of the input data is incorrect. For example, Cisco encountered this error in its VoIP phone system in 2005 [78]. In their case, phones were sent NOTIFY signals – signals which are used to indicate that messages exist in the customer's voice mailbox. These notifications did not contain any authentication information – yet they were still processed by the phone. This caused the phone to indicate that the customers had new messages waiting in their voice mailboxes. Cisco speculated that this bug could be exploited in order to conduct a denial of service attack on its system by causing a large number of customers to simultaneously check their voice mailboxes.

In order to evaluate whether LEET could detect this type of bug in the same interface in different GUI-based applications with a single rule, it was necessary to determine what sort of publicly-available system could be vulnerable to CWE-358 –type errors. The test systems would have to be able to accept multiple pieces of verification data so that it

would be possible to send some correct segments along with at least one incorrect segment. It was determined that the age gate system that is used to prevent minors from accessing the content of websites of mature-rated video games could be vulnerable to this sort of weakness. In this system, a user is asked to enter his or her age when the website initially loads. If the date entered is old enough, the website will redirect to its main page. Otherwise, the user is presented with an error message and denied access to the site's content.

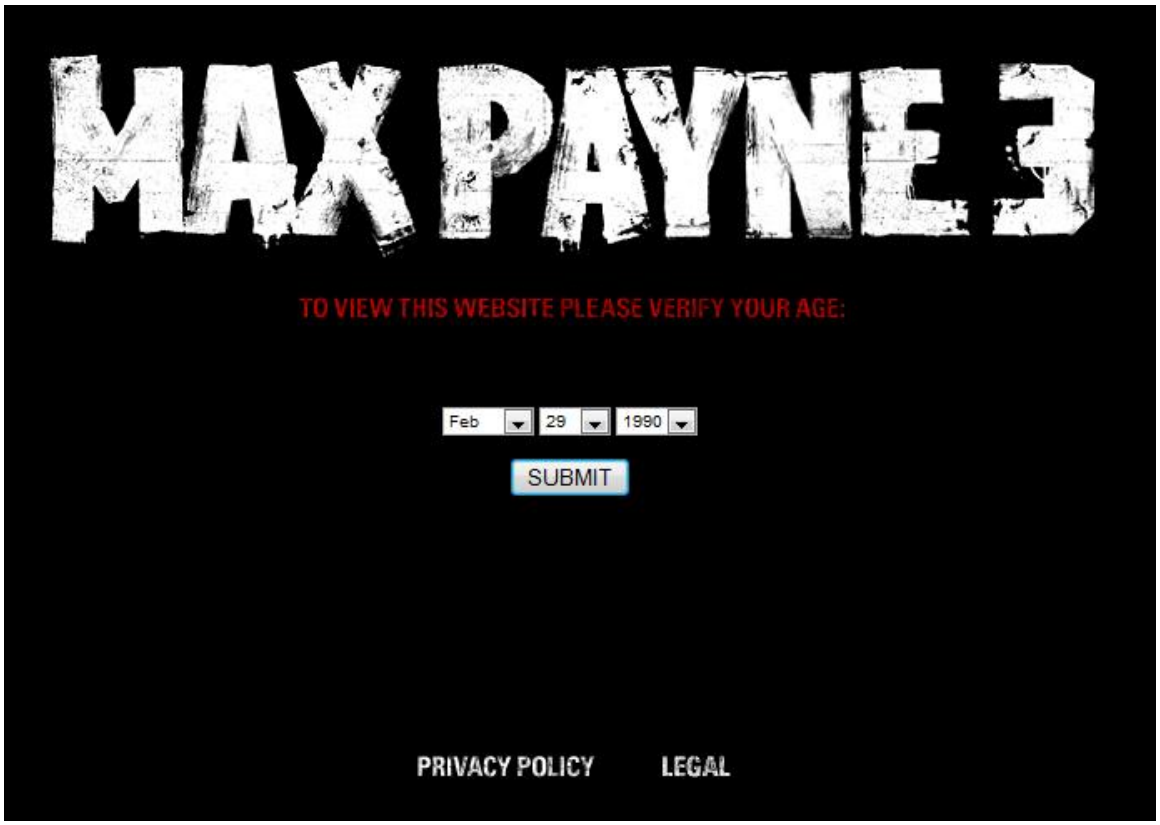

Figure 12 Age Gate for the *Max Payne 3* website (Image source: [79])

Age gates, like the one shown in Figure 12, take input arguments for the year, month, and day on which a user was born. This date is then used to determine whether to redirect to the main content of a site or to an error page. The set of rules generated for this part of

the evaluation, therefore, first detects if an age gate is present at a given state in test execution. If so, the rule then inserts a partially invalid date: 29 February, 1990. While each argument individually is valid, the date itself is imaginary since 1990 was not a leap year, and thus contained only 28 days. Since this date is invalid, the rule is considered to have been violated if the website redirects to its main page instead of its error page.

Websites on which to test this rule were chosen based on several criteria:

1) Is the website written in such a way that it can be accessed through UIA?
2) Is the website sufficiently similar to previously-selected websites?

The first criterion is necessary because certain web languages are not inherently testable using the Automation API, and it is consequently not possible to test them using LEET. For example, widgets coded in Flash do not expose any AutomationPatterns, so sections of pages that are coded in Flash do not exist from the point of view of the UIA Framework. Additionally, potential websites were manually inspected with UIAVerify to weed out websites whose age gates contained widgets that were missing information that was required for identifying them. For example, the Value property of the "ValuePattern" form of Automation Pattern is used by this rule to determine into which widget the year argument should be inserted, into which widget the month argument should be inserted, and so on. If the widget representing this field did not implement ValuePattern, or if it did implement ValuePattern but left its Value field blank, then the website was discarded from further consideration.

The second criterion simplified the coding of the rule itself. Age gates tend to fall into one of two categories. In the first, users select year, month, and day arguments from drop down lists of preset values. In the second, users type these values into text fields. Each

of these types requires a distinct set of interactions in order to select a date, so, for simplicity, only websites with age gates from the first category were selected.

The lists of Xbox 360 [80] and PlayStation 3 [81] games listed on Wikipedia were used as a source of potential websites to test. Based on the criteria above, seven websites were chosen from these lists: *Max Payne 3* [79], *Deus Ex 3* [82], *Fallout 3* [83], *Resident Evil 5* [76], *Bulletstorm* [84], *Bioshock 2* [85], and *Dragon Age: Origins* [86].

In order to code a general rule base, three of the websites that were selected were used as models when constructing the rule: *Bulletstorm*, *Bioshock 2*, and *Dragon Age: Origins*. A set of elements crucial to the functionality of the rule were identified: the dropdown boxes and their contained elements and the button that must be invoked to send this data to the server for validation.

Each site contained various quirks that were accounted for in the creation of the rule. These quirks made it difficult to create a single, general rule to detect this very specific type of bug in websites that made use of similar, but far from identical, age gates. This is different from the previous part of the preliminary evaluation, in which the error was general enough that very different applications could contain the exact same bug, which could be detected in the exact same way in every case.

In order to test for the bug described in this part of the preliminary evaluation, a set of acceptable values was created within the rule in order to allow it to function correctly on different interfaces. In this way, a heuristic of what sorts of widget names were acceptable for a given section of the rule was created. In addition to the names of widgets, the page to which each website redirects in the event of a valid or invalid date is different, so another heuristic was developed to determine whether the sites had

redirected to the error page or the main page when the invalid date was submitted. First, the value of the address bar is checked against the value it held at the beginning of the rule's invocation. If the two are not equal, it can be assumed that a page transition occurred. If the new address contains the text "sorry," as in the case of a too-recent date being entered into the *Bioshock 2* age gate, it is assumed that entry was denied. If this is not found to be the case, the page is searched for an image whose name, when converted to lower case, contains the text "esrb." This is to determine if the Entertainment Software Rating Board, or ESRB, logo is present. Consistently across sites, the rating of the game was displayed as an image containing "esrb" somewhere in the text of its name, but only after the age gate was passed. So, if this ESRB element was detected, it was assumed that the website redirected to its main page. The rule – which is actually accomplished using a set of three preconditions, four rule actions, and four consequences – as it looked after creating heuristics that enabled it to run correctly on the first three websites used in this part of the preliminary evaluation, is shown in Figures 13 through 16.

```
<RulesBase.Rule>
  WIDGET := (Value = "Jan") OR (Value = "January") OR (Value = "Month") OR (Value = "MM")
  <RulesBase.PreconditionInfo>
    CASE != Null
    TRUE? RETURN WIDGET
    FALSE? RETURN Null
  </RulesBase.PreconditionInfo>
  <RulesBase.RuleActionInfo>
    ACTION .Invoke
    WIDGET := (Name = "Feb") OR (Name = "February")
    CASE != Null
    FALSE? RETURN Null
    ACTION .Invoke
  </RulesBase.RuleActionInfo>
  <RulesBase.ConsequenceInfo>
    FATAL False
    MESSAGE (Error setting month to February).
  </RulesBase.ConsequenceInfo>
</RulesBase.Rule>
```

Figure 13: Conceptual representation of the rule that interacts with the month combo box of age gates

```
<RulesBase.Rule>
  WIDGET := (Value = "1") OR (Value = "01") OR (Value = "Day") OR (Value = "DD")
  <RulesBase.PreconditionInfo>
    CASE != Null
    TRUE? RETURN WIDGET
    FALSE? RETURN Null
  </RulesBase.PreconditionInfo>
  <RulesBase.RuleActionInfo>
    ACTION .Invoke
    WIDGET := (Name = "29")
    CASE != Null
    FALSE? RETURN Null
    ACTION .Invoke
  </RulesBase.RuleActionInfo>
  <RulesBase.ConsequenceInfo>
    FATAL False
    MESSAGE (Error setting day to 29).
  </RulesBase.ConsequenceInfo>
</RulesBase.Rule>
```

Figure 14: Conceptual representation of the rule that interacts with the day combo box of age gates

```
<RulesBase.Rule>
  WIDGET := (Value = "2009") OR (Value = "2010") OR (Value = "Year") OR (Value = "YYYY")
  <RulesBase.PreconditionInfo>
    CASE != Null
    TRUE? RETURN WIDGET
    FALSE? RETURN Null
  </RulesBase.PreconditionInfo>
  <RulesBase.RuleActionInfo>
    ACTION .Invoke
    WIDGET := (Name = "1990")
    CASE != Null
    FALSE? RETURN Null
    ACTION .Invoke
  </RulesBase.RuleActionInfo>
  <RulesBase.ConsequenceInfo>
    FATAL False
    MESSAGE (Error setting year to 1990).
  </RulesBase.ConsequenceInfo>
</RulesBase.Rule>
```

Figure 15: Conceptual representation of the rule that interacts with the year combo box of age gates

```
<RulesBase.Rule>
  WIDGET := Null
  <RulesBase.PreconditionInfo>
    ALL
  </RulesBase.PreconditionInfo>
  <RulesBase.RuleActionInfo>
    WIDGET := (Name = "Address")
    TEMP Address_Before := .Value
    WIDGET := (Name = "SUBMIT") OR (Value CONTAINS "submit") OR (Name = "Enter")
              OR (Value CONTAINS "enter") OR (Value CONTAINS "proceed")
              AND (.IsInvokePatternAvailableProperty)
    ACTION .Invoke
    WIDGET := (Name = "Address")
    NOT CASE .Value.Equals(Address_Before)
    FALSE? RETURN WIDGET
    NOT CASE .Value.Contains("sorry")
    FALSE? RETURN WIDGET
    WIDGET := (Name CONTAINS "esrb")
    True? RETURN WIDGET
    False? RETURN Null
  </RulesBase.RuleActionInfo>
  <RulesBase.ConsequenceInfo>
    FATAL True
    MESSAGE (Age gate passed an imaginary date).
  </RulesBase.ConsequenceInfo>
</RulesBase.Rule>
```

Figure 16: Conceptual representation of the rule that submits the age and decides whether the site allowed entry or not

Creating rules that can be used to detect general bugs in a variety of circumstances does not appear to require additional effort, as demonstrated by the previous section. However, creating rules that can be used to detect specific bugs in a variety of circumstances necessitates the use of heuristics to identify which elements to interact with and to determine what sort of response the system should show. It is possible in the future that these heuristics could be collected into a centralized database in order to help with the creation of rule-based tests, but this is left as future work.

After creation of the rule base was completed, exploratory test sessions were recorded for each of the seven selected websites that were selected earlier. Each of these test procedures loads the base URL for its target website and checks to see if the appropriate redirection occurs. For example, when www.bulletstorm.com is loaded, the website should automatically redirect to www.bulletstorm.com/home.

Each of these recorded exploratory tests was paired with the single rule by creating a TestRunner object. The TestRunners that tested the three websites that were used to develop the rule and for *Max Payne 3* ran correctly, while the other four failed. Even though these four tests ran correctly, it should be noted that these websites redirected to their main pages when given the imaginary date. In these four cases, the rule detected that this had happened, triggered a notification dialog, and paused execution of the test script in this state – the current behaviour for a rule violation with a nonfatal consequence.

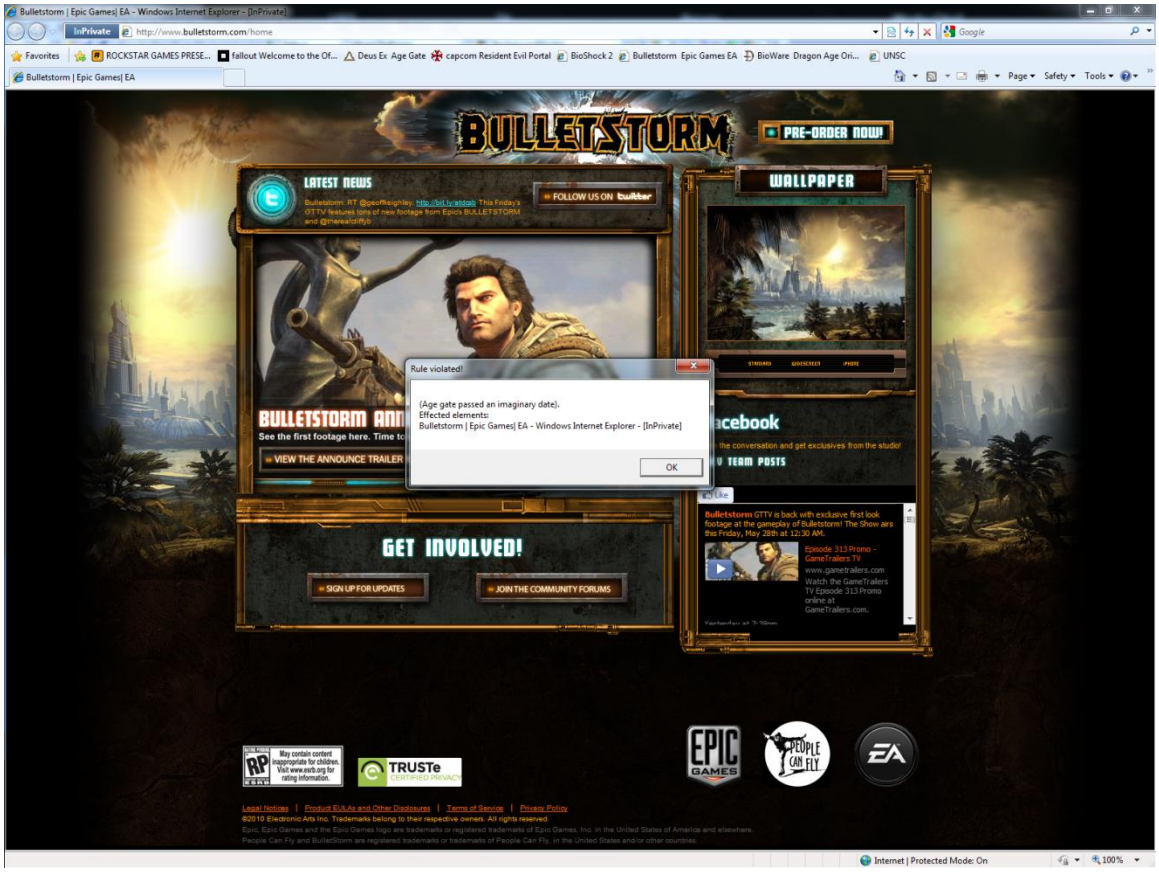

Image 1 *Bulletstorm* website, with rule failure notification at center

After the rule had failed to execute correctly for *Deus Ex 3*, *Fallout 3*, and *Resident Evil 5*, changes were made to the rule's heuristic based on a manual inspection of the failing test's website. After changes were made to the heuristic, all seven tests were run again in order to ensure that breaking changes to the rule had not been made. The results of the changes required in order for all tests to execute successfully are described in Table 3.

Table 3 Required changes for additional test websites

Game Website	*Changed Element*	*Required Change*
Resident Evil 5	Submit Button	Name: "ENTER SITE"
Deus Ex 3	Month Dropdown Box	Initial Value: Current Month
	Day Dropdown Box	Initial Value: Current Day
	Submit Button	Name: "Proceed"
Fallout 3	Submit Button	Name: "Submit"
Max Payne 3	(no changes)	

Additionally, the rule that determines if the address bar has changed to an inappropriate URL was updated to include the postfix displayed when a too-recent date was entered for each website. This resulted in the addition of checks for "noentry," "error," and "agedecline." The final version of the rule can be seen in Figures 17 through 20, with sections surrounded by green boxes indicating newly-added parts.

```
<RulesBase.Rule>
  WIDGET := (Value = "Jan") OR (Value = "January") OR (Value = "Month") OR (Value = "MM") OR (Value = CURRENT MONTH)
  <RulesBase.PreconditionInfo>
    CASE != Null
    TRUE? RETURN WIDGET
    FALSE? RETURN Null
  </RulesBase.PreconditionInfo>
  <RulesBase.RuleActionInfo>
    ACTION .Invoke
    WIDGET := (Name = "Feb") OR (Name = "February")
    CASE != Null
    FALSE? RETURN Null
    ACTION .Invoke
  </RulesBase.RuleActionInfo>
  <RulesBase.ConsequenceInfo>
    FATAL False
    MESSAGE (Error setting month to February).
  </RulesBase.ConsequenceInfo>
</RulesBase.Rule>
```

Figure 17: The current month needed to be added as a possible value for rule that interacted with the month combo box

```
<RulesBase.Rule>
  WIDGET := (Value = "1") OR (Value = "01") OR (Value = "Day") OR (Value = "DD") OR (Value = CURRENT_DAY)
  <RulesBase.PreconditionInfo>
    CASE != Null
    TRUE? RETURN WIDGET
    FALSE? RETURN Null
  </RulesBase.PreconditionInfo>
  <RulesBase.RuleActionInfo>
    ACTION .Invoke
    WIDGET := (Name = "29")
    CASE != Null
    FALSE? RETURN Null
    ACTION .Invoke
  </RulesBase.RuleActionInfo>
  <RulesBase.ConsequenceInfo>
    FATAL False
    MESSAGE (Error setting day to 29).
  </RulesBase.ConsequenceInfo>
</RulesBase.Rule>
```

Figure 18: The current day needed to be added as a possible value for rule that interacted with the day combo box

```
<RulesBase.Rule>
  WIDGET := (Value = "2009") OR (Value = "2010") OR (Value = "Year") OR (Value = "YYYY")
  <RulesBase.PreconditionInfo>
    CASE != Null
    TRUE? RETURN WIDGET
    FALSE? RETURN Null
  </RulesBase.PreconditionInfo>
  <RulesBase.RuleActionInfo>
    ACTION .Invoke
    WIDGET := (Name = "1990")
    CASE != Null
    FALSE? RETURN Null
    ACTION .Invoke
  </RulesBase.RuleActionInfo>
  <RulesBase.ConsequenceInfo>
    FATAL False
    MESSAGE (Error setting year to 1990).
  </RulesBase.ConsequenceInfo>
</RulesBase.Rule>
```

Figure 19: No changes needed to be made to the third rule

```
<RulesBase.Rule>
  WIDGET := Null
  <RulesBase.PreconditionInfo>
    ALL
  </RulesBase.PreconditionInfo>
  <RulesBase.RuleActionInfo>
    WIDGET := (Name = "Address")
    TEMP Address_Before := .Value
    WIDGET := (Name = "SUBMIT") OR (Value CONTAINS "submit") OR (Name = "Submit") OR (Name = "Enter")
              OR (Name = "Proceed") OR (Name = "ENTER SITE") OR (Value CONTAINS "enter")
              OR (Value CONTAINS "proceed") AND (.IsInvokePatternAvailableProperty)
    ACTION .Invoke
    WIDGET := (Name = "Address")
    NOT CASE .Value.Equals(Address_Before)
    FALSE? RETURN WIDGET
    NOT CASE .Value.Contains("sorry") OR .Value.Contains("decline") OR .Value.Contains("error") OR .Value.Contains("noentry")
    FALSE? RETURN WIDGET
    WIDGET := (Name CONTAINS "esrb")
    True? RETURN WIDGET
    False? RETURN Null
  </RulesBase.RuleActionInfo>
  <RulesBase.ConsequenceInfo>
    FATAL True
    MESSAGE (Age gate passed an imaginary date).
  </RulesBase.ConsequenceInfo>
</RulesBase.Rule>
```

Figure 20: Three changes were required to make the final rule compatible with the three additional websites

The results of this evaluation show that, while it is possible to create rules to test for specific weaknesses in an interface, applying this rule to similar interfaces will require revisions to the rule. While the necessary revisions encountered in this evaluation were

minor, the fact that the creation of a heuristic was necessary shows that keyword-based testing is perhaps as much of a liability to rule-based exploratory testing than it is an asset. While keyword-based testing makes it easier to manually edit tests, it makes it difficult to adapt rules to new situations. In other words, while it supports manual testing techniques, it encumbers automated testing techniques. In the future, it would be useful to add the ability to create tests that utilize a form of similarity-based widget lookup – like testing with object maps – or a form of component abstraction – in which complicated widgets are defined before rules are written – instead of keyword-based testing. This conclusion stems from the fact that it was necessary to create heuristics to complete the rule created in this part of the preliminary evaluation, and testing with object maps is, after all, a form of finding widgets based on heuristics.

5.2.3 How Often Is Keyword-Based Testing Possible?

During the previous two evaluations, several complications were encountered that prevented tests from running on certain applications or that complicated the calculation of results (Section 1.3.3). These complications were caused by the necessity of interacting with widgets that were not uniquely identifiable. The difficulties caused by these complications led to the question: how often is it possible to use keyword-based testing as a primary means of locating widgets for use with automated test procedures and oracles? This section of the preliminary evaluation presents an exploration of this issue, as well as a discussion of the findings.

A recurring difficulty encountered in the previous two sections of the preliminary evaluation is that elements were encountered that could not be tested by LEET. Widgets were encountered that had not been assigned values for either their AutomationID or

Name properties. This meant that LEET would not be able to locate them when a test was run. An additional problem was that some widgets were assigned integers for their AutomationID or Name fields. These integers appeared to be random, and would change every time applications started. This effect can be easily seen in Microsoft Visual Studio 2010. The "Properties" pane of this application contains a toolbar that is assigned a different AutomationID every time Visual Studio 2010 loads, making it unlocatable. While assigning a random integer does ensure that this widget's AutomationID field has a unique value, it also makes that value useless for testing purposes.

Based on these observations, rules were designed to explore how often it would be possible to use keyword-based testing as a primary means of locating widgets for use with automated test procedures and oracles. Five rules were created to explore the following testability issues, and these rules can be seen in Figures 21 through 25:

1) Is a widget's AutomationElement.Current.Name field empty?
2) Is a widget's AutomationElement.Current.AutomationID field empty?
3) Are 1 and 2 met on the same widget?
4) Is a widget's AutomationElement.Current.Name field an integer?
5) Is a widget's AutomationElement.Current.AutomationID field an integer?

```
<RulesBase.Rule>
  WIDGET := All
  <RulesBase.PreconditionInfo>
    CASE .Name.Trim().Equals("")
    TRUE? RETURN WIDGET
    FALSE? RETURN NULL
  </RulesBase.PreconditionInfo>
  <RulesBase.RuleActionInfo>
    RETURN WIDGET
  </RulesBase.RuleActionInfo>
  <RulesBase.ConsequenceInfo>
    FATAL False
    MESSAGE Namless widget found.
  </RulesBase.ConsequenceInfo>
</RulesBase.Rule>
```

Figure 21: Detecting nameless widgets

```
<RulesBase.Rule>
  WIDGET := All
  <RulesBase.PreconditionInfo>
    CASE .AutomationId.Trim().Equals("")
    TRUE? RETURN WIDGET
    FALSE? RETURN NULL
  </RulesBase.PreconditionInfo>
  <RulesBase.RuleActionInfo>
    RETURN WIDGET
  </RulesBase.RuleActionInfo>
  <RulesBase.ConsequenceInfo>
    FATAL False
    MESSAGE AutomationId-less widget found.
  </RulesBase.ConsequenceInfo>
</RulesBase.Rule>
```

Figure 22: Detecting id-less widgets

```
<RulesBase.Rule>
  WIDGET := All
  <RulesBase.PreconditionInfo>
    CASE .Name.Trim().Equals("")
    TRUE? RETURN WIDGET
    FALSE? RETURN NULL
  </RulesBase.PreconditionInfo>
  <RulesBase.RuleActionInfo>
    CASE .AutomationId.Trim().Equals("")
    TRUE? RETURN WIDGET
    FALSE? RETURN NULL
  </RulesBase.RuleActionInfo>
  <RulesBase.ConsequenceInfo>
    FATAL False
    MESSAGE Anonymous widget found.
  </RulesBase.ConsequenceInfo>
</RulesBase.Rule>
```

Figure 23: Detecting anonymous widgets

```
<RulesBase.Rule>
  WIDGET := All
  <RulesBase.PreconditionInfo>
    TEMP Name := .Name
    CASE TryParse(Name)
    TRUE? RETURN WIDGET
    FALSE? RETURN NULL
  </RulesBase.PreconditionInfo>
  <RulesBase.RuleActionInfo>
    RETURN WIDGET
  </RulesBase.RuleActionInfo>
  <RulesBase.ConsequenceInfo>
    FATAL False
    MESSAGE Widget name parses to an integer.
  </RulesBase.ConsequenceInfo>
</RulesBase.Rule>
```

Figure 24: Detecting integer names

```
<RulesBase.Rule>
  WIDGET := All
  <RulesBase.PreconditionInfo>
    TEMP id := .AutomationId
    CASE TryParse(id)
    TRUE? RETURN WIDGET
    FALSE? RETURN NULL
  </RulesBase.PreconditionInfo>
  <RulesBase.RuleActionInfo>
    RETURN WIDGET
  </RulesBase.RuleActionInfo>
  <RulesBase.ConsequenceInfo>
    FATAL False
    MESSAGE Widget id parses to an integer.
  </RulesBase.ConsequenceInfo>
</RulesBase.Rule>
```

Figure 25: Detecting integer ids

For this experiment, the test scripts from several of the experiments run in sections Section 5.1 and Section 5.2 were combined with these newly-created rules. The number of violations for each rule within each application are shown in Table 4.

Table 4 Violations of testability rules

	Resident Evil 5 Age Gate	Max Payne 3 Age Gate	BioShock 2 Age Gate	CharMap	Family.Show
Missing Name	17	19	32	2	416
Missing AutomationId	23	27	38	270	795
Missing Both of the Above	17	19	32	0	103
Name is an Int	0	0	0	10	44
AutomationId is an Int	0	0	0	0	0

Several observations can be drawn from these results. First, none of the applications examined supported keyword-based testing through the Automation API completely. This could severely complicate the task of creating test scripts using the current implementation of LEET. This means that the process of testing GUI-based applications using LEET would often be encumbered by the added difficulty of figuring out how to identify an element in a way that is robust against changes to the application's GUI in addition to the normal task of testing an application's functionality. Additionally, repairing broken test scripts in such cases has an added layer of difficulty: it is necessary to determine which element was initially required for a broken test script, and what has changed with it that has broken the script. Only after this is done can the basic question, "Does this failure stem from an error in the application," even be addressed.

It is interesting to note that no application tested was assigning integers to the AutomationID fields of widgets. While this was sometimes the case with Name fields, these widgets may still be robustly identifiable given that their AutomationID fields are not empty. It is also interesting to note that, in the three web applications tested, no widgets have Names that parse as integers. Another interesting result is that, whenever a widget in one of the tested web pages is missing its AutomationID, it is also missing its Name. Overall, the prevalence of empty AutomationID fields and anonymous elements within all tested applications still poses a significant challenge to automated testing. While this is not an issue for exploratory testing in isolation, it is certainly an issue for the exploratory tests enhanced with rule-based verifications that are presented in this thesis, as it makes creating and maintaining these automated verifications more difficult.

The results of this part of the preliminary evaluation can be split into two recommendations. First, effort should be placed on educating software developers who hope to make use of systems like LEET on the importance of assigning values to the AutomationID and Name fields of widgets. If all widgets in a GUI-based application were required to have a unique value assigned to their AutomationID field, for example by including a rule ensuring that this was the case as part of the suite of tests that are required to pass before new code can be accepted into an application's current build, then good coding habits could be enforced. While this option would solve the basic issue of not being able to identify a specific widget, it would not address the problem uncovered in the previous section – that applying specific rules to different interfaces required the use of heuristics. The second option, therefore, would be to use testing with object maps in future versions of LEET instead of keyword-based testing. While this option would make it harder for human testers to edit test procedures and test oracles used by LEET, it would overcome some of the issues encountered when attempting to test widgets that do not have unique AutomationIDs or when applying rules to different applications. The best way of increasing the chances of success when using rule-based exploratory testing would, of course, be to follow both of these recommendations.

5.2.4 How Much Effort Is Rule-Based Exploratory Testing?

The fourth evaluation is designed to determine how much effort recording exploratory tests and creating rules requires compared how much effort is required to simply record an exploratory test and adding static verification points to it. This second option is the approach currently used for creating tests for GUI-based applications with capture/replay tools. This comparison is done by writing simple but equivalent tests for three

applications using both approaches to GUI testing. This part of the preliminary evaluation is broken down into three sections, one for each application under test. At the end, the implications of the results are discussed.

5.2.4.1 Microsoft Calculator Plus

Microsoft Calculator Plus [87] was used as the first test application. The focus of the rule created for this calculator application is to ensure that division by zero will result in an appropriate error message being displayed in the result box of the calculator. The procedure and oracle created for the rule-based version of this test are displayed in Figure 26. Creating a test that did not use rules was accomplished by using LEET to record interactions with Microsoft Calculator Plus and adding statements to verify that the result of a series of rule actions was as expected where appropriate. This script can be seen in Figure 27. Creating the rule-based version of this test was done by creating a rule that would divide the current number by zero after each step of the test, checking to see that "Cannot divide by zero" is displayed, and clicking the clear button to ready the calculator for the next step of the test. The rule was paired with a recorded exploratory test script that simply invokes the 0 through 9 keys and closes the application. The amount of time taken to create each version of the test was recorded so that this could be used as the basis of comparison.

```
OPTIONS          | coverage=false
START    | C:\\Target Applications\\Microsoft Calculator Plus\\CalcPlus.exe
ACTION | INVOKE | 124
ACTION | INVOKE | 125
ACTION | INVOKE | 126
ACTION | INVOKE | 127
ACTION | INVOKE | 128
ACTION | INVOKE | 129
ACTION | INVOKE | 130
ACTION | INVOKE | 131
ACTION | INVOKE | 132
ACTION | INVOKE | 133
ACTION | CLOSE |
```

```
<RulesBase.Rule>
  WIDGET := (AutomationId = "90")
  <RulesBase.PreconditionInfo>
    CASE != Null
    TRUE? RETURN WIDGET
    FALSE? RETURN Null
  </RulesBase.PreconditionInfo>
  <RulesBase.RuleActionInfo>
    ACTION .Invoke
    WIDGET := (AutomationId = "124")
    ACTION .Invoke
    WIDGET := (AutomationId = "112")
    ACTION .Invoke
    WIDGET := (AutomationId = "403")
    TEMP Result := True
    CASE .Value.Equals("Cannot divide by zero.")
    FALSE? Result := False
    WIDGET := (AutomationId = "81")
    ACTION .Invoke
    CASE Result = False
    TRUE? RETURN WIDGET
    FALSE? RETURN Null
  </RulesBase.RuleActionInfo>
  <RulesBase.ConsequenceInfo>
    FATAL True
    MESSAGE Unexpected divide by zero result.
  </RulesBase.ConsequenceInfo>
</RulesBase.Rule>
```

Figure 26: Procedure (left) and oracle (right) for the first rule-based test.

For this section, the rule-based approach was taken first, followed by the creation of the CRT-only version of the test. This was alternated for each of the following two sections – Section 5.2.4.2 was CRT first, then rule-based, and Section 5.4.2.3 was rule-based first, then CRT based. This was done in order to minimize any potential learning effect.

Figure 27: The CRT-only version of the first test.

Creating the simple script to invoke the 0 through 9 buttons and close Microsoft Calculator Plus took 1 minute, and ran correctly on the first try. Creating the rule took just under 9 minutes. The rule did not run correctly on the first attempt, and a further 3 minutes were required in order to get the rule into working order. Therefore, the total time required to create the rule-based version of the test was a bit under 13 minutes. This rule-based approach did uncover a bug (by the definition of the rule): when dividing 0 by 0, the message "Result of function is undefined." is displayed instead of the expected "Cannot divide by zero."

Creating a script that performed all of the interactions in the simple script as well as all of the rule actions performed by the rule base required just less than 8 minutes. This script did not work on the first attempt. Irregularities involving the hierarchy of elements in Microsoft Calculator Plus required 2 minutes of debugging to fix. In sum, creating this equivalent script took just under 10 minutes.

The results of this section of the evaluation are summarized in Table 5. Creating a script and adding verification points manually took around 23% less time than using the rule-based approach. However, where the equivalent script has no further uses, the rule base created in the first half of the test – which took the majority of the time to create – could be paired with other tests of that application.

Table 5: Breakdown of time taken to create each test, in minutes

	Creation of procedure for rule-based version	Debugging of procedure for rule-based version	Creation of rule-based verifications	Debugging of rule-based verifications	Creation of CRT-only version	Debugging of CRT-only version	Total time for rule-based version	Total time for CRT-only version
Microsoft Calculator Plus	1	0	9	3	8	2	13	10

5.2.4.2 Internet Explorer 8.0

Internet Explorer 8.0 (IE8) was used as the second test application. The rule created for this test focused on the functionality of the back and forward buttons in IE8's interface. It was expected that invoking the back and forward buttons in that order should result in a return to the current page. The test procedure was created by recording visits to 9 pages, resulting in 8 states in which this rule could be applied as the back button is not enabled until at least one page has been visited. The procedure and oracle created for the rule-based version of the test can be seen in Figure 28. The time required to create and debug both the test script and the rule for this part of the evaluation were recorded. An equivalent script was also created using a CRT-only approach instead of rules. The CRT-only version of this test can be seen in Figure 29.

```
OPTIONS          | coverage=false
OPTIONS          | delay=3000
START    | C:\\Program Files\\Internet Explorer\\iexplore.exe | -private
ACTION | INVOKE | ROCKSTAR GAMES PRESENTS MAX PAYNE 3 (2)
ACTION | INVOKE | fallout Welcome to the Official Site
ACTION | INVOKE | Deus Ex  Age Gate
ACTION | INVOKE | capcom Resident Evil Portal
ACTION | INVOKE | BioShock 2
ACTION | INVOKE | Bulletstorm  Epic Games EA
ACTION | INVOKE | BioWare  Dragon Age Origins
ACTION | INVOKE | UNSC
ACTION | INVOKE | Home
ACTION | CLOSE |
```

```
<RulesBase.Rule>
  WIDGET := Null
  <RulesBase.PreconditionInfo>
    WIDGET := (Name = "Back")
    CASE .IsEnabled
    TRUE? RETURN WIDGET
    FALSE? RETURN Null
  </RulesBase.PreconditionInfo>
  <RulesBase.RuleActionInfo>
    WIDGET := (Name = "Address")
    TEMP Address_Before := .Value
    WIDGET := (Name = "Back")
    ACTION .Invoke
    WIDGET := (Name = "Forward")
    ACTION .Invoke
    WIDGET := (Name = "Address")
    CASE .Value.Equals(Address_Before)
    TRUE? RETURN Null
    FALSE? RETURN WIDGET
    </RulesBase.RuleActionInfo>
  <RulesBase.ConsequenceInfo>
    FATAL True
    MESSAGE Back/Forward button behavior is inconsistent.
  </RulesBase.ConsequenceInfo>
</RulesBase.Rule>
```

Figure 28: Procedure (left) and oracle (right) for the second rule-based test.

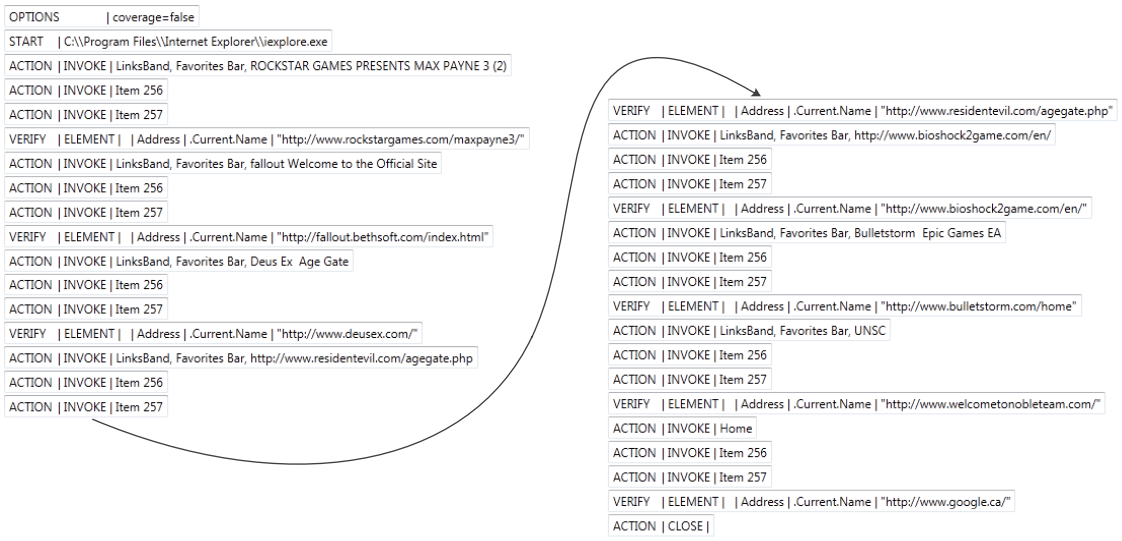

Figure 29: CRT-only version of the second test.

Creating a script to load IE8, visit each of the 9 pages, and close IE8 at the end took almost 4 minutes, with an additional minute and a half required for debugging, bringing the total time required to get a script into working order with LEET up to around 5 and a

half minutes. Creating the rule for this test required 5 and a half minutes, with an additional 10 minutes for debugging, leading to a total rule creation time of 15 and a half minutes. In total, then, writing and debugging the script and rule for this test took 21 minutes. No irregularities were uncovered in the functionality of IE8 through this test scenario.

Creation of the equivalent test without rules took 10 minutes, and several errors prevented it from running initially. Debugging these errors took 2 minutes, bringing the time required to code this equivalent script to 12 minutes.

The results of this section of the preliminary evaluation are summarized in Table 6. In this case, creating a script that performed all of the interactions performed by the simple script and rule base combination took 41% less time to do.

Table 6: Breakdown of time taken to create each test, in minutes

	Creation of procedure for rule-based version	Debugging of procedure for rule-based version	Creation of rule-based verifications	Debugging of rule-based verifications	Creation of CRT-only version	Debugging of CRT-only version	Total time for rule-based version	Total time for CRT-only version
Microsoft Calculator Plus	1	0	9	3	8	2	13	10
Internet Explorer 8.0	4	1.5	5.5	10	10	2	21	12

5.2.4.3 LEET

LEET itself was used as the third test application. The rule for this test focused on the functionality of the "Add Event" and "Remove Event" buttons in the in the capture/replay functionality provided by LEET. It is expected that selecting the "Add Event" button should add a new event to the script LEET is currently creating, and that selecting this event and invoking the "Remove Event" button should remove that event from the CRV.

The script used for this test was based on a test that is part of the suite of tests that were originally developed for LEET, and has been in use since August of 2009. Recoding this test took 7 and a half minutes, and fixing the errors made while coding it took 11 minutes. Creating the rule took 15 and a half minutes, with an additional 4 minutes for debugging. Overall, coding and debugging the script and the rule took 38 minutes. Both this test and the equivalent approach created without rules can be seen in Figures 30 through 32.

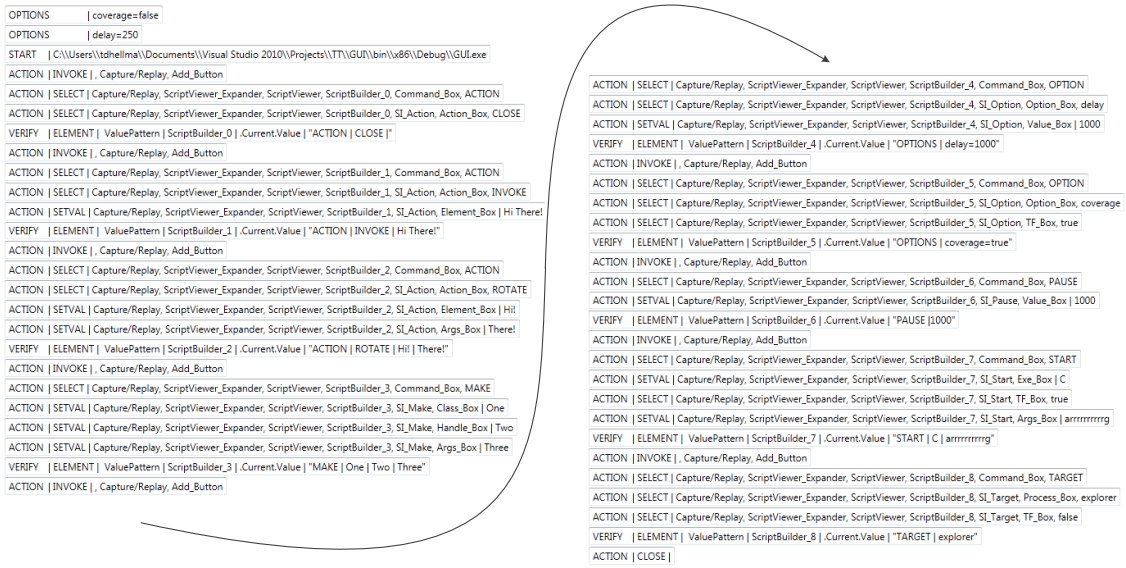

Figure 30: The procedure used for the rule-based version of the test.

```
<RulesBase.Rule>
  WIDGET := (AutomationId = "ScriptViewer")
  <RulesBase.PreconditionInfo>
    CASE .IsEnabled
    TRUE? RETURN WIDGET
    FALSE? RETURN Null
  </RulesBase.PreconditionInfo>
  <RulesBase.RuleActionInfo>
    TEMP OriginalChildren := .Value
    WIDGET := (AutomationId = "Add_Button")
    ACTION .Invoke
    WIDGET := (AutomationId = "ScriptViewer")
    CASE .Value > OriginalChildren
    FALSE? RETURN WIDGET
    WIDGET := (AutomationId = "ScriptBuilder_" + .Value)
    ACTION .Select
    WIDGET := (AutomationId = "Remove_Button")
    ACTION .Invoke
    WIDGET := (AutomationId = "ScriptViewer")
    CASE .Value = OriginalChildren
    TRUE? RETURN Null
    FALSE? RETURN WIDGET
  </RulesBase.RuleActionInfo>
  <RulesBase.ConsequenceInfo>
    FATAL True
    MESSAGE Error using Add/Remove Action functionality.
  </RulesBase.ConsequenceInfo>
</RulesBase.Rule>
```

Figure 31: The oracle used for the rule-based version of the test.

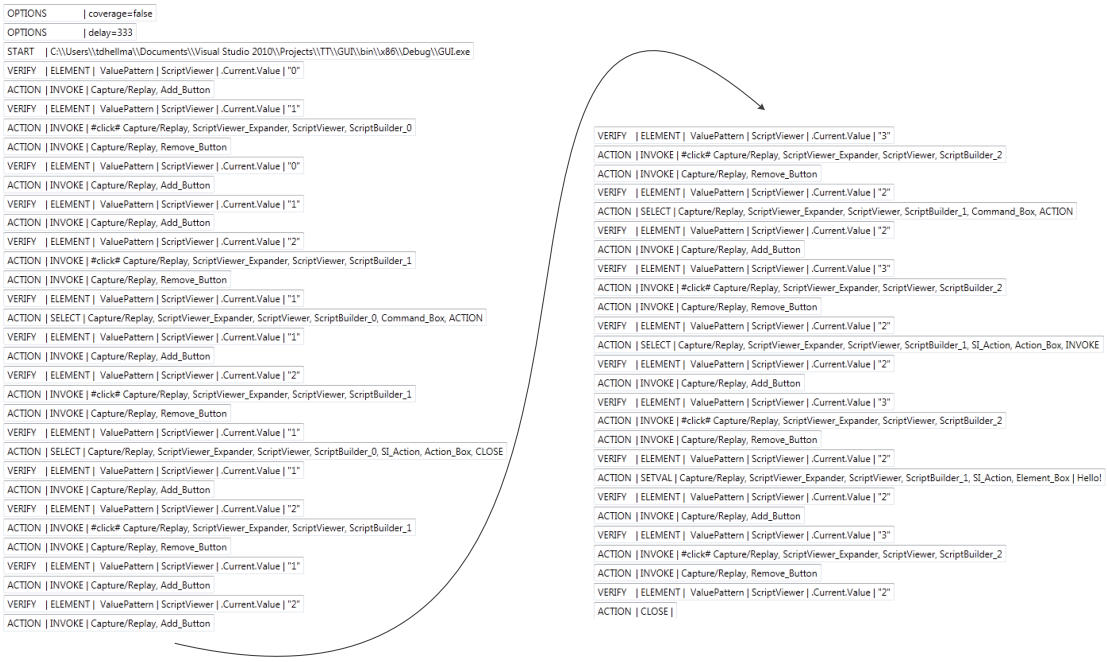

Figure 32: The CRT-only version of 12 out of 50 interactions in the test.

Coding a test that performed all of the interactions performed by the above script and rule is very difficult, so a subset was coded. The first 12 of the 50 interactions performed in the original script were rerecorded as well as each action performed by the rule-based verifications in the previous approach. Performing the necessary verifications accounted for most of the effort involved in this process and was tedious and error-prone. Doing so took 19 minutes, with an additional 6 minutes for debugging. The amount of time for each of these was averaged over the number of interactions recorded, and these averages were used to project the time that would be required to record every interaction in the original test/rule base combination by hand: 1 hour and 44 minutes. In this case, rule-based testing presented a projected savings of over one hour. The results of this section of the preliminary evaluation are summarized in Table 7.

Table 7: Breakdown of time taken to create each test, in minutes (* - projected)

	Creation of procedure for rule-based version	Debugging of procedure for rule-based version	Creation of rule-based verifications	Debugging of rule-based verifications	Creation of CRT-only version	Debugging of CRT-only version	Total time for rule-based version	Total time for CRT-only version
Microsoft Calculator Plus	1	0	9	3	8	2	13	10
Internet Explorer 8.0	4	1.5	5.5	10	10	2	21	12
LEET	7.5	11	15.5	4	79*	25*	38	104*

5.2.4.4 How Much Effort Is Rule-Based Exploratory Testing? – Conclusions

In this subsection, the effort required to write rule-based exploratory tests was compared to the effort required to write equivalent tests using static verification points instead of rules. Tests were created for three different applications using both of these methods, and the time required for each was compared. The results of this portion of the preliminary evaluation were inconclusive.

In Sections 5.2.4.1 and 5.2.4.2, it would seem that rule-based was less efficient than coding an equivalent test by hand. In Section 5.2.4.3, however, rule-based testing was projected to be more efficient than inserting verification points by hand. Further evaluations are necessary before any sort of statement can be made about the efficiency of rule-based testing.

5.3 Weaknesses of Evaluations

The primary weakness of these evaluations is that they are all self-evaluations. The tests were written by the author, on systems with which the author familiarized himself. In order to increase their credibility, it would be best to conduct user studies, in which test subjects would be asked to write rule-based tests and non-rule-based tests. Different aspects of these two groups could then be compared, and a more generally applicable assessment of the resulting data could be performed. However, this is left for future work.

A second weakness is the narrow number of test applications used in each evaluation. Only 12 different applications were used throughout Chapter 5, and the most used in any one evaluation was 7. In order to strengthen these evaluations, additional test applications should be included.

A third weakness is the low number of rules overall that are demonstrated. Throughout Chapter 5, only 11 rules are demonstrated. Additional rules should be demonstrated in the future.

5.4 Conclusions

In this section, various preliminary evaluations of rule-based testing with LEET were undertaken, and the implications of the results were discussed. A summary is provided here.

In Section 5.2.1, the ability of general rules to detect violations in various GUI-based applications was explored. It was found that rules for the detection of high-level, general bugs could be created and used without alteration to detect bugs in three very different applications.

In Section 5.2.2, the practicality of creating a specific, low-level rule and trying to apply it to different applications was explored. It was found that, due to the level of detail required in the rule that was created, it was necessary to use heuristics in order to locate widgets. This implies that testing with object maps may be a better strategy for locating widgets to use with exploratory rule-based testing than keyword-based testing.

It has been noted that keyword-based testing might cause problems when used with rule-based exploratory testing, so Section 5.2.3 explored this issue further. Rules were constructed to explore the testability of applications. It was found that it the applications tested all included some widgets that were not properly enabled for testing. This implies that effort should be spent on ensuring that developers are creating testable applications and that it would be useful use testing with object maps in future implementations of rule-based exploratory testing.

In Section 5.2.4, the amount of effort required to perform rule-based testing in specific systems was explored. In two of the three tests, it was found that rule-based testing could be expected to take longer to perform than simply coding a test that performs all of the verifications that the rule-based approach would perform. However, in the third test, rule-based testing proved less time-consuming than coding this equivalent test. The results of this part of the preliminary evaluation were inconclusive, and further effort should be dedicated to this issue.

Chapter Six: Conclusions

This thesis presents an approach to the testing of GUI-based applications by combining manual exploratory testing with automated rule-based testing. First, an overview of the challenges involved in GUI testing was presented to provide the background necessary to understand the challenges of this field. Next, a discussion of previous attempts to provide automated support for GUI testing was presented, and the strengths and weaknesses of these approaches were discussed. A tool, LEET, was created to make rule-based exploratory testing possible. This was done so that this approach to GUI testing could be subjected to pilot evaluations. The structure of LEET was explained, and the strengths and weaknesses of both the design of the system and its concrete implementation were discussed. Pilot evaluations were then conducted to point to potential answers for the research questions described in Section 1.4, and to give insight into the strengths and weaknesses of rule-based exploratory testing.

6.1 Thesis Contributions

The first contribution of this thesis was the literature review covering the current state of GUI testing from an academic perspective presented in Chapter 2. This review not only provided a background to understanding the challenges involved in attempting to automate GUI testing, but it also categorized past attempts into the major difficulty of GUI testing, as described in Section 1.3, that each attempt addressed. This categorization should make it easier to focus GUI testing efforts in the future because it will be easier to determine which difficulties the advantages of a new approach address, and to understand what's already been done in this direction. At present, this is difficult to do in that

research on GUI testing tends to be self-identified based on the kind of testing involved – self-categorized as research in regression testing of GUIs or goal-driven automated test generation rather than as approaches that lessen the complexity of GUIs or approaches that lessen the impact of changes to the GUI on GUI tests.

The second contribution of this thesis is LEET. LEET fulfills the first research goal listed in Chapter 1. Not only is LEET the only tool currently able to perform automated rule-based verifications on GUI-based applications, it's also the only tool that provides any sort of automated oracles in support of manual exploratory testing. Most important, however, is that LEET makes it possible to conduct pilot evaluations to evaluate under which circumstances rule-based exploratory testing is useful, and what pitfalls to avoid in future implementations.

Third, this thesis explored the usefulness and practicality of rule-based exploratory testing through investigations into the 5 research questions posed in Section 1.5. In this way, it accomplished the second research goal listed in Chapter 1. The first question, "Can rule-based exploratory testing be used to catch general bugs," was investigated in Section 5.2.1. From this pilot evaluation it would appear that not only can rule-based exploratory testing be used to catch high-level, general bugs, but these sorts of rules can detect a large number of general bugs by using short rules.

The second question, "Can rule-based exploratory testing be used to catch specific bugs," was investigated in Section 5.2.2. The pilot evaluation suggests, again, that rule-based testing is a practical way of testing for low-level, specific bugs that occur only when specific interfaces are used. However, one problem encountered in this section was that LEET's use of keyword-based testing detracted from the variety of automated rules that

LEET could use. It was found necessary, in fact, for heuristics to be built into the rules used in this section in order to enable them to correctly identify widgets in a variety of specific interfaces.

This issue was further investigated in the fourth research question, "how often is it possible to use keyword-based testing on GUIs," in Section 5.2.3. This section of the pilot evaluation seems to suggest that issues confounding keyword-based testing may be widespread. There are two ways of dealing with this issue. First, effort could be spent educating developers on the importance of making sure the GUIs they create are compatible with keyword-based testing in the same way the importance of Model-View-Controller pattern has been stressed in the past. Second, future implementations of systems that support manual exploratory testing with automated rule-based verifications could make use of testing with object maps rather than keyword-based testing. This also seems to be indicated by the fact that it was found necessary to start building heuristics within rules to identify the widgets required for testing.

The third research question, "can rules be reused in tests of different applications," is very much related to this discussion of keyword-based testing. It was possible to use general rules on a variety of different interfaces in Section 5.2.1 and Section 5.2.2 of the pilot evaluation. However, effort was required to adapt the more specific rules used in Section 5.2.2 to work on other interfaces. The pilot evaluations that were conducted were insufficient to answer this research question, and it would be prudent to revisit this issue with a tool that makes use of testing with object maps before passing judgement on the reusability of rules.

The fifth question, “is rule-based exploratory testing less effort than writing equivalent tests using a capture/replay tool and inserting verifications manually,” is likewise unanswerable from these pilot evaluations. In order to answer a question of this magnitude, more detailed case studies should be conducted using a second-generation tool for enhancing exploratory testing with rule-based verifications.

The last contribution of this thesis is a clear plan for future research on rule-based exploratory testing. For the next study, a second-generation tool, one that leverages testing with object maps, should be developed. First, evaluations should be conducted to determine whether this new implementation is superior to LEET in its ability to create low-level, specific rules that can be run on multiple applications. Next, this tool should be used to compare rule-based exploratory testing to testing performed by adding verifications manually to a recorded exploratory test. Several axes should be used in this evaluation: how much effort is it to create each kind of test; how many bugs can be caught by each method from a test system that has been seeded using mutation testing; and what kind of bugs are caught by each approach. However, these evaluations would are at present left as future work.

6.2 Future Work

There are many directions in which LEET and rule-based GUI testing can be taken. These directions could not be explored within the scope of this thesis, and so they remain work for the future.

The first step that needs to be taken is the completion of the implementation of LEET. Currently, events from only a handful of AutomationPatterns are being recorded. LEET can interact with widgets that expose InvokePattern, ValuePattern,

ExpandCollapsePattern, TransformPattern, TogglePattern, TextPattern, SelectionPattern, SelectionItemPattern, and WindowPattern AutomationPatterns. However, this still leaves DockPattern, GridPattern, GridItemPattern, MultipleViewPattern, RangeValuePattern, ScrollPattern, ScrollItemPattern, TablePattern, and TableItemPattern, to be implemented. Doing so will expand the range of applications that can be tested with LEET

The next enhancement to LEET to make is multi-window recording. When LEET records an event, this event is given a timestamp. This makes it is possible to simultaneously record interactions occurring in several windows at once and to replay the entire test procedure in order. However, LEET currently focuses on recording a single window at a time, and enabling multi-window recording is left for future work.

Third, much work remains to be done with test-driven development (TDD) of GUIs-based applications. While two publications [37] [38] have featured the use of LEET in conjunction with ActiveStory: Enhanced [88] to enable TDD, but these publications focus on the use of LEET as a CRT, and rule-based TDD of GUI-based applications remains an unexplored topic.

Fourth, there is a possibility that rule-based testing could be used to increase the code coverage of a base test suite. In this approach, pre-existing tests could be paired with rules that take actions in order to move the GUI-based application into new states to increase code coverage. While various attempts have been made in the past to automatically increase code coverage these approaches tend to drive state expansion using test procedures alone. Using rule-based testing to perform state expansion keeps a test oracle at the center of the testing process, and could help keep the focus of testing on catching bugs rather than on running code. Fourth, while the original application that

LEET was constructed to test, APDT, featured a tangible user interface (TUI), the main focus of LEET's development has been on testing general GUI-based applications. It would be possible at this point to modify LEET to be useful for applications with TUIs, and very few contributions have been made to this area of research.

As for rule-based exploratory testing in general, experimental and industrial evaluations need to be performed based on the recommendations made above. Several pilot evaluations were conducted during the course of this thesis that indicate where rule-based exploratory testing could be useful and what steps can be taken to make it more useful. User studies should be performed based on these recommendations in order to determine how useful rule-based exploratory testing of GUI-based applications can be.

References

[1] A. M. Memon, "A Comprehensive Framework for Testing Graphical User Interfaces," University of Pittsburgh, PhD Thesis 2001.

[2] IEEE. (2004) SWEBOK Guide - Chapter 5. [Online]. http://www.computer.org/portal/web/swebok/html/ch5#Ref3.1.2

[3] Juha Itkonen and Kristian Rautiainen, "Exploratory Testing: A Multiple Case Study," in *International Symposium on Empirical Software Engineering*, Noosa Heads, Australia, 2005, pp. 84-92.

[4] Juha Itkonon, Mika V. Mäntylä, and Casper Lassenius, "Defect Detection Efficiency: Test Case Based vs. Exploratory Testing," in *First International Symposium on Empirical Software Engineering and Measurement*, Madrid, Spain, 2007, pp. 61-70.

[5] James Bach. (2000) James Bach - Satisfice, Inc. [Online]. http://www.satisfice.com/presentations/gtmooet.pdf

[6] The Wikimedia Foundation, Inc. (2010, June) Rete algorithm - Wikipedia, the free encyclopedia. [Online]. http://en.wikipedia.org/wiki/Rete_algorithm

[7] Qing Xie and Atif M. Memon, "Using a Pilot Study to Derive a GUI Model for Automated Testing," *ACM Transactions on Software Engineering and Methodology*, vol. 18, no. 2, pp. 1-35, October 2008.

[8] Q. Xie and A. M. Memon, "Studying the Characteristics of a "Good" GUI Test Suite," in *Proceedings of the 17th International Symposium on Software Reliability*

Engineering, Raleigh, NC, 2006, pp. 159-168.

[9] Scott McMaster and Atif Memon, "Call Stack Coverage for GUI Test-Suite Reduction," in *International Symposium on Software Reliability Engineering*, Raleigh, 2006, pp. 33-44.

[10] C. Kaner and J. Bach. (2005, Fall) Center for Software Testing Education and Research. [Online]. www.testingeducation.org/k04/documents/BBSTOverviewPartC.pdf

[11] A. Memon, I. Benerjee, and A. Nagarajan, "What Test Oracle Should I Use for Effective GUI Testing," in *18th IEEE International Conference on Automated Software Engineering*, Montreal, 2003, pp. 164-173.

[12] Microsoft Corporation. (2010, June) MSDN Canada. [Online]. http://msdn.microsoft.com/en-us/library/ms697707.aspx

[13] Microsoft Corporation. (2010, June) MSDN Canada. [Online]. http://msdn.microsoft.com/en-us/library/ms726294(VS.85).aspx

[14] A. M. Memon and M. L. Soffa, "Regression Testing of GUIs," in *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2003, pp. 118-127.

[15] Atif M. Memon, "Automatically Repairing Event Sequence-Based GUI Test Suites for Regression Testing," *ACM Transactions on Software Engineering and Methodology*, vol. 18, no. 2, pp. 1-36, October 2008.

[16] A. Holmes and M. Kellogg, "Automating Functional Tests Using Selenium," in

AGILE 2006, 2006, pp. 270-275.

[17] B. Robinson and P. Brooks, "An Initial Study of Customer-Reported GUI Defects," in *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*, 2009, pp. 267-274.

[18] Brian Marick, "Bypasing the GUI," *Software Testing and Quality Engineering Magazine*, pp. 41-47, September/October 2002.

[19] Brian Marick, "When Should a Test Be Automated?," in *Proceedings of the 11th International Software Quality Week*, vol. 11, San Francisco, May 1998.

[20] Wirold Wysota, "Testing User Interfaces in Applications," in *1st International Conference on Information Technology*, Gdansk, Poland, 2008.

[21] Steve Burbeck. (1987, 1992) How to Use Model-View-Controller (MVC). [Online]. http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html

[22] Svetoslav Ganov, Chip Killmar, Sarfraz Khurshid, and Dewayne E. Perry, "Test Generation for Graphical User Interfaces Based on Symbolic Execution," in *International Conference on Software Engineering*, Leipzig, 2008, pp. 33-40.

[23] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa, "Hierarchical GUI Test Case Generation Using Automated Planning," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 144-155, February 2001.

[24] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa, "Using a Goal-Driven Approach to Generate Test Cases for GUIs," in *International Conference on Software Engineering*, Los Angeles, 1999, pp. 257-266.

[25] Xun Yuan, Myra B. Cohen, and Atif M. Memon, "Towards Dynamic Adaptive Automated Test Generation for Graphical User Interfaces," in *International Conference on Software Testing, Verification, and Validation Workshops*, Washington D.C., 2009, pp. 263-266.

[26] Xun Yuan and Atif M. Memon, "Alternating GUI Test Generation and Execution," in *Testing: Academic and Industrial Conference - Practice and Research Techniques*, 2008, pp. 23-32.

[27] Kai-Yuan Cai, Lei Zhao, and Feng Wang, "A Dynamic Partitioning Approach for GUI Testing," in *30th Annual International Computer Software and Applications Conference*, Chicago, 2006, pp. 223-228.

[28] Antti Jääskeläinen et al., "Automatic GUI Testing for Smartphone Applications - An Evaluation," in *International Conference on Software Engineering*, Vancouver, 2009, pp. 112-122.

[29] Tamás Dabóczi, István Kollár, Gyula Simon, and Tamás Megyeri, "How to Test Graphical User Interfaces," *IEEE Instrumentation and Measurement Magazine*, vol. 6, no. 3, pp. 27-33, September 2003.

[30] Hassan Reza, Sandeep Endapally, and Emanuel Grant, "A Model-Based Approach for Testing GUI Using Hierarchical Predicate Transition Nets," in *International Conference on Information Technology*, Las Vegas, 2007, pp. 1-5.

[31] Richard K. Shehady and Daniel P. Siewiorek, "A Method to Automate User Interface Testing Using Variable Finite State Machines," in *27th International*

Symposium on Fault-Tolerant Computing, Seattle, 1997, pp. 80-88.

[32] Christof J. Budnik, Fevzi Belli, and Axel Hollmann, "Structural Feature Extraction for GUI Test Enhancement," in *IEEE International Conference on Software Testing, Verification, and Validation Workshops*, Denver, 2009, pp. 255-262.

[33] Lee Whilte, Husain Almezen, and Shivakumar Sastry, "Firewall Regression Testing of GUI Sequences and their Interactions," in *International Conference on Software Maintenance*, Amsterdam, 2003, pp. 398-410.

[34] Atif M. Memon and Qing Xie, "Studying the Fault-Detection Effectiveness of GUI Test Cases for Rapidly Evolving Software," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 884-896, October 2005.

[35] Atif Memon, Ishan Benerjee, and Adithya Nagarajan, "GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing," in *Proceedings of the 10th Working Conference on Reverse Engineering*, Victoria, B.C., Canada, 2003, pp. 260-270.

[36] Yongzhong Lu, Danping Yan, Songlin Nie, and Chun Wang, "Development of an Improved GUI Automation Test System Based on Event-Flow Graph," in *International Conference on Computer Science and Software Engineering*, Washington D.C., 2008, pp. 712-715.

[37] Theodore D. Hellmann, Ali Hosseini-Khayat, and Frank Maurer, "Agile Interaction Design and Test-Driven Development of User Interfaces - A Literature Review," in *Agile Software Development: Current Research and Future Directions*, Torgeir

Dingsøyr, Tore Dybå, and Nils Brede Moe, Eds. Trondheim, Norway: Springer, 2010, ch. 9.

[38] Theodore D. Hellmann, Ali Hosseini-Khayat, and Frank Maurer, "Supporting Test-Driven Development of Graphical User Interfaces Using Agile Interaction Design," in *Third International Conference on Software Testing, Verification, and Validation Workshops*, Paris, 2010, pp. 444-447.

[39] Mark Grechanik, Qing Xie, and Chen Fu, "Creating GUI Testing Tools Using Accessibility Technologies," in *International Conference on Software Testing, Verification, and Validation Workshops*, Denver, 2009, pp. 243-250.

[40] M. Grechanik, Q. Xie, and F. Chen, "Maintaining and Evolving GUI-Directed Test Scripts," in *IEEE 31st International Conference on Software Engineering*, 2009, pp. 408-418.

[41] C. Fu, M. Grechanik, and Q. Xie, "Inferring Types of References to GUI Objects in Test Scripts," in *International Conference on Software Testing, Verification, and Validation*, 2009, pp. 1-10.

[42] Z. Yin, C. Miao, Z. Shen, and Y. Miao, "Actionable Knowledge Model for GUI Regression Testing," in *IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, 2005, pp. 165-168.

[43] Woei-Kae Chen, Sheng-Wen Shen, and Che-Ming Chang, "GUI Test Script Organization with Component Abstraction," in *Second International Conference on Secure System Integration and Reliability Improvement*, Yokohama, 2008, pp. 128-

134.

[44] Qing Xie and Atif M. Memon, "Designing and Comparing Automated Test Oracles for GUI-Based Software Applications," *ACM Transactions on Software Engineering and Methodology*, vol. 16, no. 1, pp. 1-36, February 2007.

[45] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa, "Automated Test Oracles for GUIs," in *ACM Special Interest Group on Software Engineering*, San Diego, 2000, pp. 30-39.

[46] Ali Mesbah and Arie van Deursen, "Invariant-Based Automatic Testing of AJAX User Interfaces," in *International Conference on Software Engineering*, Vancouver, 2009, pp. 210-220.

[47] GNOSO. (2010, June) NCover - Code Coverage for.NET Developers. [Online]. http://www.ncover.com/

[48] Mutant Design Limited. (2010, June) TestDriven.Net > Home. [Online]. http://www.testdriven.net/

[49] (2010, June) Selenium Web Application Testing System. [Online]. http://seleniumhq.org/

[50] ThoughtWorks. (2010, June) white. [Online]. http://white.codeplex.com/

[51] International Business Machines Corporation. (2010, June) IBM - IBM Rational Functional Tester - Functional Testing - Rational Functional Tester - Software. [Online]. http://www-01.ibm.com/software/awdtools/tester/functional/

[52] Timothy Wall. (2005, May) Abbot Framework for Automated Testing of Java GUI

Components and Programs. [Online]. http://abbot.sourceforge.net

[53] Jonathan Bennett. (2004, February) AutoIt Script Home Page. [Online]. http://www.autoitscript.com/

[54] (2010, June) Automation Anywhere - Leader in Automation Software, Automated Testing. Automate with Ease. [Online]. http://www.automationanywhere.com/

[55] Edgewell Software. (2010, June) dogtail - Trac. [Online]. https://fedorahosted.org/dogtail/

[56] TestPlant Limited. (2009, December) Automated Software Testing for teh User Interface | TestPlant. [Online]. http://www.testplant.com/

[57] Bredex GmbH. (2010, May) Bredex GUIdancer. [Online]. http://www.bredex.de/en/guidancer

[58] NXS-7 Software Incorporated. (2009, December) IcuTest | Automated GUI Unit Testing for WPF. [Online]. http://www.icutest.com

[59] (2010, February) Home - Linux Desktop Testing Project. [Online]. http://ldtp.freedesktop.org

[60] Phantom Automated Solutions, Incorporated. (2009, December) Phantom Automated Solutions - Phantom Test Driver GUI Test Automation. [Online]. http://phantomtest.com/PTDInfo.htm

[61] Seapine Software, Incorporated. (2010, June) Automated testing, Software Testing, Application Testing - QA Wizard Pro - Seapine Software. [Online]. http://www.seapine.com/qawizard.html

[62] Qalibers. (2010, March) Qalibers. [Online]. http://www.qaliber.net/

[63] Quality First Software GmbH. (2010, June) Quality First Software GmbH & GF-Test - The Java GUI Test Tool. [Online]. http://www.qfs.de/en/

[64] Hewlett-Packard Development Company, L.P. (2010, June) HP QuickTest Professional Software - HP - BTO Software. [Online]. https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-127-24^1352_4000_100__

[65] Ranorex GmbH. (2010, May) Test Automation Tools - Ranorex Automation Framework. [Online]. http://www.ranorex.com/

[66] Cogitek Incorporated. (2010, June) Flex Test Automation Tool - RIA Test. [Online]. http://www.riatest.com/

[67] MicroFocus Limited. (2009, December) DATA SHEET | SilkTest. [Online]. http://www.microfocus.com/assets/silktest-data-sheet_tcm6-6802.pdf

[68] The Eclipse Foundation. (2010, June) SWTBot - UI Testing for SWT and Eclipse. [Online]. http://www.eclipse.org/swtbot/

[69] Test Automation FX. (2008, December) Test Automation Fx - UI Testing with Visual Studio. [Online]. http://www.testautomationfx.com/

[70] AutomatedQA Corporation. (2010, June) Automated Testing Tools - TestComplete. [Online]. http://www.automatedqa.com/products/testcomplete/

[71] Micro Focus Limited. (2009, December) DATA SHEET | TestPartner. [Online]. http://www.automatedqa.com/products/testcomplete/

[72] Instantiations, Incorporated. (2010, June) WindowTester Pro. [Online]. http://www.windowtester.com/

[73] Microsoft Corporation. (2010, March) LEET (LEET Enhances Exploratory Testing). [Online]. http://leet.codeplex.com/

[74] Microsoft Corporation. (2008, March) UI Automation Verify (UIA Verify) Test Automation Framework. [Online]. http://uiautomationverify.codeplex.com/

[75] Vertigo Software. (2009, Feb) Family.Show. [Online]. http://familyshow.codeplex.com/

[76] Capcom, Inc. (2010, May) Capcom: Resident Evil Portal. [Online]. www.residentevil.com

[77] (2010, April) CWE - Common Weakness Enumeration. [Online]. http://cwe.mitre.org/data/definitions/358.html

[78] Tobias Glemser and Reto Lorenz. (2005, July) Tele-Consulting security | networking | training GmbH. [Online]. http://pentest.tele-consulting.com/advisories/05_07_06_voip-phones.txt

[79] Rockstar Games. (2010, May) Rockstar Games Presents Max Payne 3. [Online]. www.rockstargames.com/maxpayne3

[80] (2010, May) List of Xbox 360 games - Wikipedia, the free encyclopedia. [Online]. http://en.wikipedia.org/wiki/List_of_Xbox_360_games

[81] (2010, May) List of Xbox 360 games - Wikipedia, the free encyclopedia. [Online]. http://en.wikipedia.org/wiki/List_of_PlayStation_3_games

[82] Square Enix Ltd. (2010, May) Deus Ex : Human Revolution. [Online]. www.deusex3.com

[83] Bethesda Softworks LLC. (2010, May) Fallout: Welcome to the Official Site. [Online]. www.fallout.bethsoft.com

[84] Electronic Arts Inc. (2010, May) Bulletstorm | Epic Games | EA. [Online]. www.bulletstorm.com

[85] Take-Two Interactive Software. (2009) BioShock 2. [Online]. www.bioshock2game.com

[86] BioWare Corp. (2010) BioWare | Dragon Age: Origins. [Online]. dragonage.bioware.com

[87] Microsoft Corporation. (2005, June) Download Details: Microsoft Calculator Plus. [Online]. http://www.microsoft.com/downloads/details.aspx?familyid=32b0d059-b53a-4dc9-8265-da47f157c091&displaylang=en

[88] Ali Hosseini-Khayat, Yaser Ghanam, Shelly Park, and Frank Maurer, "ActiveStory Enhanced: Low-Fidelity Prototyping and Wizard of Oz Usability Tool," in *Agile Processes in Software Engineering and Extreme Programming*, 2009, pp. 257-258.

[89] A. Ruiz and Price Y. W., "Test-Driven GUI Development with TestNG and Abbot," in *IEEE Software*, 2007, pp. 51-57.

[90] A. Ruiz. (2007, July) Test-Driven GUI Development with FEST. From Test-driven GUI development with FEST- JavaWorld. [Online]. Http://www.javaworld.com/javaworld/jw-07-2007/jw-07-fest.html

[91] A. Ruiz and Y. W. Price, "GUI Testing Made Easy," in *Testing: Academic and Industrial Conference - Practice and Research Techniques*, 2008, pp. 99-103.

[92] N. Nagappan, E. M. Maximilien, T. Bhat, and L. Williams, "Realizing Quality Improvement through Test Driven Development: Results and Experiences of Four Industrial Teams," in *Empirical Software Engineering*, 2008, pp. 289-302.

[93] R Jeffries and G. Melnik, "Guest Editors' Introduction: TDD - The Art of Fearless Programming," *IEEE Software*, pp. 24-30, 2007.

[94] Theodore D. Hellmann. (2010) LEET (LEET Enhances Exploratory Testing) - CodePlex. [Online]. http://leet.codeplex.com/

[95] W. Chen, T. Tsai, and H. Chao, "Integration of Specification-Based and CR-Based Approaches for GUI Testing," in *19th International Conference on Advanced Information Networking and Applications*, 2005, pp. 967-972.

[96] MITRE Corporation. (2010, May) CWE - Common Weakness Enumeration. [Online]. http://cwe.mitre.org/

[97] (2010, May) CodePlex - Open Source Project Hosting. [Online]. http://www.codeplex.com/

[98] (2010, June) Wikipedia. [Online]. http://en.wikipedia.org/wiki/List_of_GUI_testing_tools

[99] (2009, May) Agile Software Engineering - University of Calgary | AgilePlanning / Agile Planner For Digital Tabletop. [Online]. ase.cpsc.ucalgary.ca/index.php/AgilePlanning/AgilePlannerForDigitalTabletop