Rule-Based Exploratory Testing of Graphical User Interfaces

Theodore D. Hellmann and Frank Maurer
The University of Calgary
2500 University Drive NW
Calgary, Alberta, Canada T2N 1N4
{theodore.hellmann, frank.maurer}@agilesoftwareengineering.org

***Abstract*—This paper introduces rule-based exploratory testing, an approach to GUI testing that combines aspects of manual exploratory testing with rule-based test automation. This approach uses short, automated rules to increase the bug-detection capability of recorded exploratory test sessions. This paper presents the concept of rule-based exploratory testing, our implementation of a tool to support this approach, and a pilot evaluation conducted using this tool. The preliminary evaluation found that this approach can be used to detect both general and application-specific bugs, but that rules for general bugs are easier to transfer between applications. Also, despite the advantages of keyword-based testing, it interferes with the transfer of rules between applications.**

Keywords - GUI testing; rule-based testing; exploratory testing

I. MOTIVATION

Nearly every modern application uses a graphical user interface (GUI) as its main means of interacting with users. GUIs allow users to interact with applications via user interface elements – or *widgets* – that respond to text input, mouse movement, and mouse clicks, which makes interacting with computers more natural and intuitive. Of these GUI-based applications, 45-60% of the total code of the application can be expected to be dedicated to its GUI [1]. Because GUIs allow users so much freedom of interaction, they are very difficult to test. This paper discusses a new approach to GUI testing that enhances the process of manual exploratory testing with automated, rule-based verifications.

Perhaps the strongest case for GUI testing is that GUI-based bugs do have a significant impact on an application's users. 60% of defects can be traced to code in the GUI, and 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 [2].

This means that, regardless of the difficulties involved, GUI testing is important in reducing the number of bugs discovered by customers after release. However, despite the usefulness of automated testing in detecting and reproducing bugs, it is currently common for industrial testers to bypass the GUI during automated testing. This can be done using a test harness which interacts with the application below the level of its GUI [3] or by skipping automated GUI testing entirely in favor of manual approaches [4]. There are compelling reasons to use these approaches rather than to automate GUI tests: GUIs are very complicated in terms of the number of widgets the are composed of and the number of ways with which each widget can be interacted, leading to a huge state space to test; 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. These factors make the creation and maintenance of automated GUI tests quite difficult – but, at the same time, they also increase the need for it because, without an automated regression suite, it is easy to introduce and hard to detect regression errors in the GUI. In manual testing, on the other hand, these issues are somewhat mitigated due to a human tester's ability to use experience and intuition to focus testing effort on interesting parts of a GUI – in essence, to restrict the state space on which testing will focus to areas that are likely to contain bugs. However, manual approaches are unsuitable for use in repeated regression testing due to the effort and time required to perform this type of testing.

Two factors influence the ability of automated GUI tests to trigger bugs: the number of steps in a test; and the number of times each action in the GUI is taken within a test [5]. These factors can be visualized as the amount of the state space of the application under test that is explored during testing. Further, in order to notice that a bug has been triggered, a test must also contain verifications complex enough to notice that bug [6]. However, automated tests with complex verifications require a long time to execute [7] [8] [6]. Since a regression suite needs to be able to execute quickly, GUI tests that are included in the regression suite tend to be simple in order to allow them to execute quickly, resulting in tests that are unlikely to catch bugs [6] [9].

Also, GUIs tend to change over the course of development, which ends up breaking GUI tests - that is, the functionality of the GUI is still working, but the tests report that it is not [10] [11]. These false positives can lead to developers losing confidence in the regression suite, which undermines its original purpose [12].

Based on this, a method of GUI testing needs to be developed that makes it easy to verify the correctness of an application, that runs quickly in a regression suite, and will not break as the application is changed over time. We focused on *automated rule-based testing* and *manual exploratory testing* in our attempt to develop a better approach to GUI testing. This is based on previous suggestions that exploratory testing be enhanced with additional automated verifications [13]. Rule-based testing is a form of automated testing which can simplify the verification and validation logic of a test as well as reduce the chances that a test will break when a GUI is changed. Exploratory testing is an approach to testing GUIs manually

that leverages a human tester's ingenuity, but is expensive to perform repeatedly. In this paper, we propose a combination of these two methods into *rule-based exploratory testing* (R-BET), present a tool, LEET (LEET Enhances Exploratory Testing), that supports this approach, and perform a pilot evaluation to determine whether R-BET can be applied to a set of sample testing situations and what issues arise with regards to these applications.

The first step to combining these two methods is to record the interactions performed by a human tester during an exploratory test session as a replayable script. This can be accomplished using a capture/replay tool (CRT) – a tool that records interactions with an application as a script and can later replay these actions as an automated test. Next, a human tester defines a set of rules that can be used to define the expected (or forbidden) behavior of the application. The rules and script are then combined into an automated regression test which increases the state space of the system that is tested. This approach allows a human tester to use exploratory tests to identify regions of the state space of the system that need to be subjected to more rigorous rule-based testing, which, in effect, identifies an important subset of the system under test on which testing should focus. At the same time, this subset is tested thoroughly using automated rules in order to verify this subset more thoroughly than would be possible with exploratory testing alone.

The following section presents related work which lays the foundation for a discussion of our approach in Section III. In order to evaluate whether R-BET is actually practical, we investigated 4 research questions:

- Can rule-based exploratory testing be used to catch application-independent, high-level bugs?
- Can rule-based exploratory testing be used to catch application-specific, low-level bugs?
- How often is it possible to use keyword-based testing on GUIs?
- Is rule-based exploratory testing less effort than writing equivalent tests by using a CRT and inserting verifications manually?

Experiments were designed to investigate these topics, and the results of this pilot evaluation are presented in Section IV. These experiments tend to derive from the field of security testing in order to reinforce the applicability of our approach to an important area of GUI testing. Based on these results, we are able to suggest in Section V that R-BET may be a practical method of supporting exploratory testing with automated testing practices, and are able to make clear recommendations for future work based on our experiments on applying R-BET to real-world software systems.

II. Related Work

There have been many attempts to improve GUI testing through creating better CRTs and through the use of automatically-generated GUI test suites. However, as this paper presents an approach for combining rule-based GUI testing and exploratory GUI testing, this section will focus on related work directly related these two approaches.

Additionally, tools automated acceptance testing which interact with an application below the level of the GUI also exist. These test harnesses are able to bypass some of the issues with GUI testing identified in the previous section, but are unsuitable for making assertions about the GUI itself. Tools in this category include the acceptance testing tools FitNesse[1] and GreenPepper[2]. However, as our approach focuses on testing an application and its GUI through its GUI, these tools are not discussed in this section.

A. Exploratory Testing

Exploratory testing is a form of testing in which human testers interact with a system based on their knowledge, experience, and intuition in order to find bugs, and has been described as "simultaneous learning, test design, and test execution" [14]. By using a human's judgement to determine whether or not a feature is working correctly, it's possible to focus testing effort on areas of an application that are seen as more likely to contain bugs.

Despite its ad-hoc nature, exploratory testing has become accepted in industry and is felt to be an effective way of finding bugs [15]. Practitioner literature argues that exploratory testing also reduces overhead in creating and maintaining documentation, helps team members understand the features and behavior of the application under development, and allows testers to immediately focus on productive areas during testing [15] [16]. Further, a recent academic study has shown that exploratory testing is at least as effective at catching bugs as scripted manual testing – a similar technique in which tests are written down and executed later by human testers – and is less likely to report that the system is broken when it is actually functioning correctly [16].

However, there are several practical difficulties involved with exploratory testing. First, human testers can only test a subset of the functionality of an application within 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 and makes it impractical to use exploratory testing for regression testing. 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 [15]. This makes it difficult to direct exploratory testing towards areas of an application that have not been tested previously, and increases the risk of leaving sections of an application untested.

Because of this, practitioners of exploratory testing argue for a diversified testing strategy, including exploratory and automated testing [13]. Such a testing strategy would combine the benefits of exploratory testing with the measurability, thoroughness, and repeatability of automated testing. However, there is a lack of tool support that would augment exploratory testing with automated testing techniques.

B. Rule-Based Verification

A rule-based approach to GUI testing has been used in the past to validate each state of an AJAX web interface [17]. 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

[1] www.fitnesse.org
[2] http://www.greenpeppersoftware.com

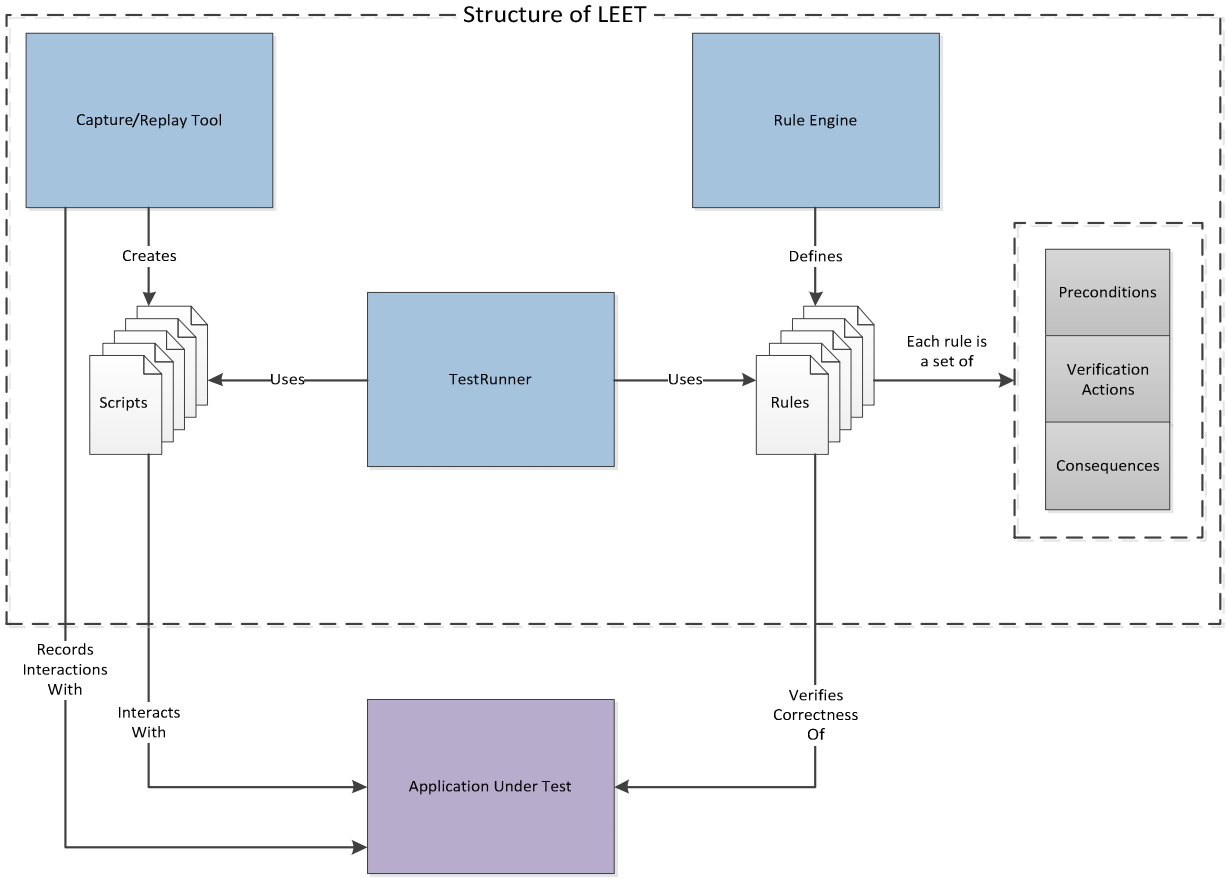

Figure 1. Structure of R-BET as implemented by LEET

procedure of an AJAX application can be easily automated using a web crawler, all that 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 only validation and are useful enough to aid in testing remains difficult.

A similar technique has been applied to GUI-based applications, in which events are defined as a set of preconditions and effects [18]. 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 correct for a given widget.

The value of both 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 verify that the application will not enter specific states, and these verifications can be performed during the execution of a large number of separate tests.

III. LEET Approach

As was stated in Section II.A, it has been suggested that manual exploratory testing could benefit from the addition of automated support. In light of this, we propose that manual exploratory test sessions be recorded in a replayable format, then enhanced with short, automated rules that increase the amount of verification performed when that test is replayed. In this way, only a subset of the state space of the application under test in which a human has expressed interest will receive additional automated scrutiny. The additional verifications provided by these rules will increase the parts of the state space that are tested, but only in that same subset identified by the human tester. In this way, we aim to create strong, relevant tests by relying on the repeatability and verification ability of rule-based testing as well as the intelligence of human testers. The fact that rules contain preconditions also makes it less likely that rule-based tests will falsely report failures when the GUI changes – instead, they will simply not fire when they are not applicable. It is important to note that this approach does not solve any of the difficulties of GUI testing identified in Section 1 – instead, R-BET represents a method of simplifying GUI testing such that these difficulties can be mitigated.

We developed a tool, LEET, to enable us to test out the concept of R-BET. The overall structure of LEET's implementation is shown in Figure 1. LEET can work as a CRT by recording events raised by the Windows Automation API [3] as users interact with applications developed for computers running Windows XP, Vista, or 7. This functionality can be used to record exploratory test sessions as test scripts and to replay them later as regression tests.

Next, LEET can be used to create rules – short verifications that will interact with a system to ensure that a specific property is always (or never) true. Each rule takes the form of

[3] http://msdn.microsoft.com/en-us/library/dd561932(v=vs.85).aspx

an "if… try… catch…" statement. The "if," or *precondition*, of a rule makes sure that it will only fire under certain circumstances. For instance, if a rule should only ensure that a widget is disabled when it is offscreen, a precondition for this rule might be "if the current widget is offscreen." If a rule has multiple preconditions, then all of these must be met before a rule will fire, because the preconditions are connected with a logical AND. The same precondition can be used by more than one rule. The "try," or *action*, represents the main body of the rule, and will be executed when all preconditions are met. In the previous example, the action might be "assert that the current widget is not enabled." An action can be as simple as a verification of a single property or as complex as a series of interactions with the application under test. The "catch," or *consequence*, determines what should happen if the action fails or throws an exception. This allows test authors to distinguish between *failures* that indicate bugs and *warnings* that represent that a coding standard has not been met. In the previous example, it might not be necessary to fail the entire test if an offscreen widget was enabled, but it might be helpful to log this warning so that developers will be made aware of its existence. Standard security tests could also be defined as rules through LEET. For example, a rule could be defined to attempt SQL or JavaScript injection attacks on an application under test. Additionally, preconditions could be used to ensure that rules only attempt these attacks through widgets matching certain criteria. The evaluation, in Section IV, includes several additional examples of rules that it is possible to create and use with LEET.

The rules are combined with recorded exploratory tests as a TestRunner – an executable program that combines a recorded exploratory test with a set of rules. A TestRunner runs a test script one step at a time and checks each rule against the system under test after each of step. In this way, testers are able to define rules that will help explore the state space of the application under test more thoroughly. In the example used in the previous paragraph, a rule can be defined that will check that each widget in the GUI is not both enabled and offscreen. This could be checked manually by a human tester, but it would be a tedious task. Creating a rule to test for this behavior will reduce the amount of work that must be done by human testers at the same time as increasing the number of different states from which this verification can be performed. Additionally, rules can be defined to test for typical errors that a human tester may overlook, may not have time to test for, may not be experienced enough to know about. Automated, rule-based verification not only allows a system to be tested more thoroughly than would otherwise be possible within a given timeframe, but also frees up human testers to perform more interesting testing.

IV. Pilot Evaluation

This section presents a preliminary evaluation of LEET's implementation of R-BET and is intended to show that this approach is practical. The questions in this section are drawn from the list of research questions in Section I and are investigated through four controlled experiments.

A. *Can rule-based exploratory testing be used to catch application-independent, high-level bugs?*

This section explores the ability of rule-based exploratory testing to catch application-independent, high-level bugs. In order to explore this topic, a specific security flaw 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 paired with these rules. The number of violations of these rules is given, and the implications of R-BET's ability to detect these violations are explored.

It is possible to initially create widgets outside of the visible area of a screen. This is sometimes done to increase the apparent speed of an application, since copying an existing widget from one position to another is a faster operation than creating it from scratch. This trick can make a GUI-based application appear to run faster after the initial setup is done. It is sometimes possible, however, to interact with widgets even if they not displayed within the visible area of the screen. This could be a problem in an application where users are given access to different features depending on whether they are logged in as a normal or administrative user. If the widgets relating to both user types are rendered offscreen when the application loads and these widgets are enabled, then it is possible for administrator-only functionality to be invoked without logging into an administrator account. Tools like UIA Verify[4] 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 respond to events until they are displayed onscreen.

Three (very different) applications in which it would be possible to record exploratory tests of the application's basic functionality using LEET were selected: Family.Show [5]; the Character Map application included with Windows 7; and the website for Resident Evil 5 [6]. These applications were selected because they are compatible with LEET – which is to say that they raise events appropriately through the Automation API – and they are significantly different from each other. As shown in this example, a single rule created for LEET can work with significantly different types of interfaces.

First, two rules were created to check for widgets are responsive to events even though they would not normally be visible to users: the first has a precondition that will cause it to fire only on dimensionless elements (those with dimensions 0 width by 0 height); the second has a precondition that will cause it to fire only on widgets that are rendered offscreen (outside of the visible area of the screen). The action on both of these rules will then check to see if widgets that meet the precondition are responding to events, and, if so, the consequence will cause a warning to be raised containing the number of widgets that violate the rule. Widgets that are not visible to users shouldn't be able to respond to events that are raised through the user interface, so the rule is considered violated if this is possible. These rules are defined through C# code, but a conceptual representation of them in a somewhat

[4] http://uiautomationverify.codeplex.com/

[5] http://familyshow.codeplex.com/

[6] www.residentevil.com

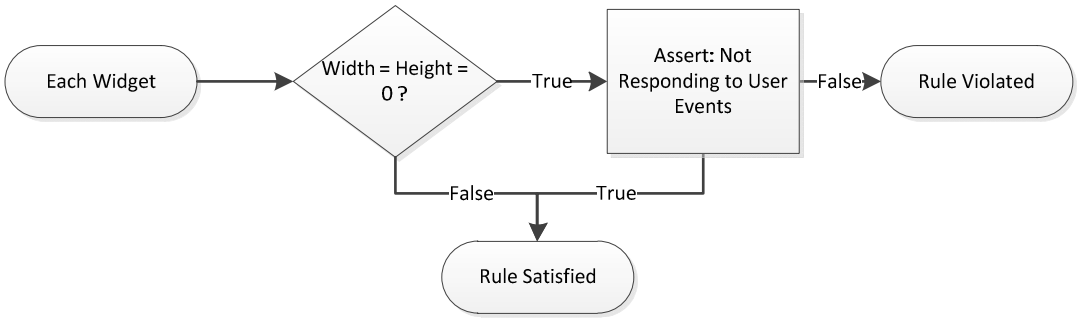

Figure 2. Structure of a rule designed to detect dimensionless widgets that are responding to simulated user input

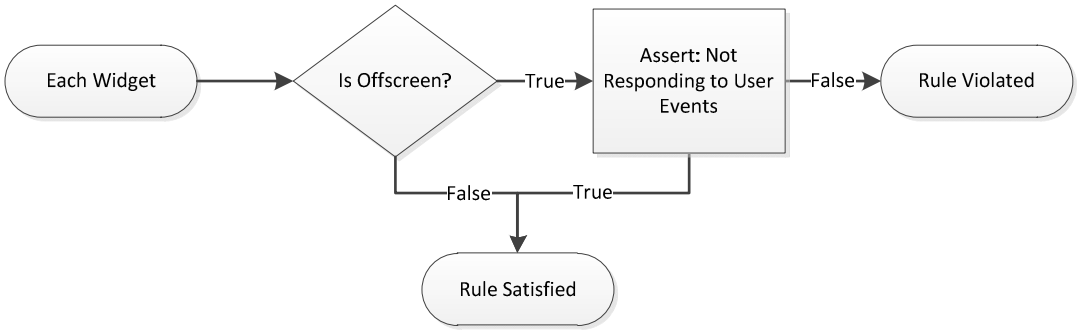

Figure 3. Structure of a rule designed to detect offscreen widgets that are responding to simulated user input

more readable format, similar to the domain-specific language in which LEET records exploratory test sessions, is shown in Figures 2 and 3. In interpreting this conceptual representation, 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.

Next, exploratory tests of some of the basic functionality of each of these three applications were recorded using LEET. Three TestRunners were created to combine each recorded exploratory test session with the two rules shown in Figures 2 and 3. Each TestRunner was run on the system for which its exploratory test session was recorded, and a substantial number of violations of both rules were discovered. The minimum number of violations of each rule discovered during the run of each TestRunner, as shown in Table I, was recorded. 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 IV.C.

TABLE I. MINIMUM NUMBER OF ERRONEOUSLY ENABLED WIDGETS IN EACH TEST APPLICATION

Application	Offscreen Widgets	Dimensionless Widgets
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 application-independent rules can detect when a GUI's widgets are responding to input even though they are not visible to users, which could lead to a security breach. Further, 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. Most importantly, by using three significantly different applications, the results imply that it is possible to catch high-level, application-independent bugs through R-BET.

B. *Can rule-based exploratory testing be used to catch application-specific, low-level bugs?*

In this part of the pilot study, the ability of rule-based exploratory testing to detect application-specific, low-level bugs was investigated. The widget focused on in this part of the evaluation is a validation interface used in many web-based applications. *Age gates* are interfaces used to verify that a user is old enough to view the content within a website. 7 websites that make use of age gates and are testable by LEET were selected for use in this experiment, and a single rule was created based on a manual inspection of 3 of these. This rule, explained below, was designed to determine whether each site's age gate represents a reliable security measure and 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 below. Finally, the implications of the results of this experiment are discussed.

The bug used to explore this topic is based on "Improperly Implemented Security Check for Standard" from the Common Weakness Enumeration [19]. 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. In order to determine whether R-BET can detect an improperly implemented security check in the same validation interface in different GUI-based applications using a single rule, it was necessary to determine what sort of publicly-available system could be vulnerable. The test systems 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. The age gate system used to prevent minors from accessing the content of websites of mature-rated video games is one such system. Age gates require a user 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.

The set of rules generated for this part of the evaluation first detects if an age gate is present at a given point during 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 as 1990 was not a leap year. 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:

- Is the website written in such a way that it can be accessed through the Automation API?
- Is the website sufficiently similar to previously-selected websites?

The first criterion is necessary because certain web languages are not testable using the Automation API, and it is consequently not possible to test them using LEET. For example, LEET will not work with applications coded in Flash. 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

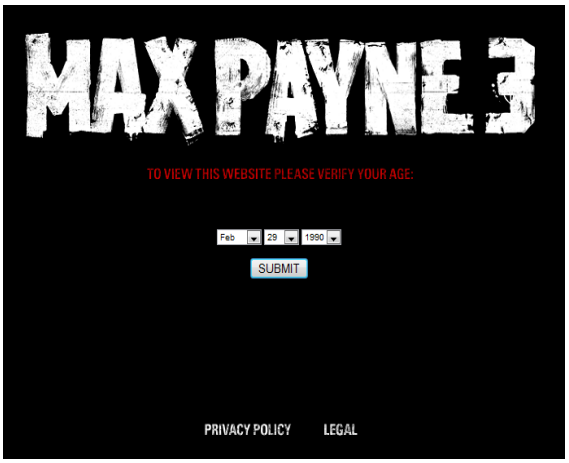

Figure 4. Age Gate for the Max Payne 3 website (www.rockstargames.com/maxpayne3)

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 not used in this preliminary evaluation.

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 [7] and PlayStation 3 [8] games listed on Wikipedia were used as a source of potential websites to test. Based on the criteria above, seven websites were chosen: *Max Payne 3* [9], *Deus Ex 3* [10], *Fallout 3* [11], *Resident Evil 5*, *Bulletstorm* [12], *Bioshock 2* [13], and *Dragon Age: Origins* [14].

In order to create a general rule base, three of the websites 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 specific bug in websites using similar – but not identical – age gates.

In order to test for the bug described above, the rule used a heuristic to allow it to identify widgets important for its functionality on different interfaces. The heuristic contained different names that an analogous widget might have in different interfaces. 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. Thus, 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. The rule was implemented using a set of three preconditions, four rule actions, and four consequences.

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 websites that

[7] http://en.wikipedia.org/wiki/List_of_Xbox_360_games (May 2010)
[8] http://en.wikipedia.org/wiki/List_of_PlayStation_3_games (May 2010)
[9] www.rockstargames.com/maxpayne3
[10] www.deusex3.com
[11] http://fallout.bethsoft.com/
[12] www.bulletstorm.com
[13] http://www.bioshock2game.com/
[14] dragonage.bioware.com

were selected. Each of these recorded exploratory tests was paired with the rule by creating a TestRunner object. These TestRunners were used to test the four remaining websites. Of these, the rule was unable to execute in three instances: *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, and 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 III.

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 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 might require some revisions. While the revisions encountered in this evaluation were minor, it is important to note that keyword-based testing – the system LEET uses to find widgets to interact with – makes it difficult to adapt R-BET to new situations. In keyword-based testing, only a single property of a widget is used to identify it. For example, widgets may be assigned an AutomationID that is expected to be unique, which makes it a good identifier to use in keyword-based testing. When a test is run, then, LEET will simply look for a widget with a given AutomationID rather than using a complicated heuristic to determine which widget to interact with. However, this means that rules are more likely to fail erroneously when running on a different application then the one they were coded against since it is unlikely that widgets for similar behavior will have exactly the same name in different applications. In the future, it will be important to investigate a form of similarity-based system of widget lookup for use instead of keyword-based testing in order to increase the reusability of rules between applications.

C. *How often is it possible to use keyword-based testing on GUIs?*

During the development of LEET, widgets without their AutomationID or Name fields set were frequently encountered. LEET uses these fields in its approach to keyword-based testing, and is not able test widgets without this information because it cannot locate them when a test is run. This difficulty led to the question: how often is it possible to use keyword-based testing to locate widgets for use with automated test procedures and oracles?

To investigate this question, rules were designed to explore how often it was 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 investigate the following testability issues:

- Is a widget's Name field empty?
- Is a widget's AutomationID field empty?
- Are 1 and 2 met on the same widget?
- Is a widget's Name field an integer?
- Is a widget's AutomationID field an integer?

For this experiment, the test scripts from several of the experiments run in Sections IV.A and IV.B were combined with these newly-created rules. The number of violations for each rule within each application is shown in Table IV.

None of the applications examined supported keyword-based testing completely. This could severely complicate the task of creating GUI test scripts using keyword-based testing, as is the case in LEET. Additionally, repairing broken test scripts in such cases has an added layer of difficulty: before it is possible to understand why a test has broken, testers first need to determine which widget the test was intended to interact with. Overall, the prevalence of empty AutomationID fields and anonymous elements within all tested applications poses a significant challenge to automated testing. While this is not an issue for manual exploratory testing in isolation, it is certainly an issue for R-BET in its current implementation.

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 properly naming widgets in their GUIs. 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 a similarity-based system of finding widgets 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.

TABLE II. 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
Deus Ex 3	Day Dropdown Box	Initial Value: Current Day
Deus Ex 3	Submit Button	Name: "Proceed"
Fallout 3	Submit Button	Name: "Submit"
Max Payne 3	(no changes)	(no changes)

TABLE III. VIOLATIONS OF TESTABILITY RULES

	Resident Evil 5	**Max Payne 3**	**BioShock 2**	**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

D. *Is rule-based exploratory testing less effort than writing equivalent tests by using a CRT and inserting verifications manually?*

The fourth evaluation was aimed at determining how much effort R-BET is compared to how much effort it would be to create a test by manually editing the script produced by a CRT. In this evaluation, equivalent tests were created using two methods: using LEET to record an exploratory test, then creating rules; and using LEET to record an exploratory test, then inserting a set of verification statements into that script. These tests were created for three different applications: Microsoft Calculator Plus [15]; Internet Explorer 8.0 (IE8); and LEET. In order to reduce learning bias, the order of test creation was alternated between systems. So, for Microsoft Calculator Plus, tests were created using R-BET first; in IE8, tests were created using the CRT-only method first; and in LEET, tests were created using R-BET first.

1) Microsoft Calculator Plus: The focus of the rule created Microsoft Calculator Plus is to ensure that division by zero will result in an appropriate error message being displayed in the result box of the calculator. 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. Creating the R-BET 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.

The times taken for each approach to produce a passing test are summarized in Table IV. Creating a script and adding verification points manually took around 23% less time than using the R-BET 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. Additionally, the R-BET approach uncovered an inconsistency in the application: when dividing 0 by 0, the message "Result of function is undefined." is displayed instead of the expected "Cannot divide by zero."

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 CRT-based test was created by recording visits to 9 pages, resulting in 8 states from which the back button could be invoked, and inserting a verification to ensure that the expected page transition had occurred. An equivalent script was also created using R-BET.

The results of this section of the preliminary evaluation are summarized in Table IV. 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.

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 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 R-BET version of this test recoded a test that had been part of LEET's regression suite since 2009, and included 50 interactions with the system.

Creating a test that performed all of these interactions strictly through a CRT 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. The results of this section of the preliminary evaluation are summarized in Table IV. In this example, using the R-BET approach presented a very significant decrease in the amount of time it took to create the test – it would take only 37% as long to create this test with R-BET compared with a CRT.

4) Intermediate Conclusions: Unfortunately, the results of this portion of the preliminary evaluation were inconclusive. In the first and second experiments, it would seem that R-BET is less efficient than coding an equivalent test by hand. In the third experiment, however, R-BET was more efficient even though only a subset of all required CRT tests were encoded.

E. *Weaknesses of Evaluations*

The primary weakness of these evaluations is that they are all self-evaluations. The tests were written by one of the authors, on systems with which he 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

TABLE IV. 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**
LEET	7.5	11	15.5	4	79*	25*	**38**	**104***

(* - Projected time)

[15] http://www.microsoft.com/downloads/en/details.aspx?familyid=32b0d059-b53a-4dc9-8265-da47f157c091 (February 2011)

these two groups could then be compared, and a more generally applicable assessment of the resulting data could be performed.

A second weakness is the narrow number of test applications used in each evaluation. Only 12 different applications were used throughout this evaluation, and the most used in any one experiment 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 this paper, only 11 rules are mentioned. Additional rules should be demonstrated in the future in order to better assess the applicability of R-BET as well as the reusability of rules across applications.

V. Conclusions

This paper 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. Our approach to R-BET was explained along with the structure of LEET, our implementation of a tool that supports R-BET. Pilot evaluations were conducted to point to potential answers for the research questions described in Section I, and to give insight into the strengths and weaknesses of rule-based exploratory testing.

Our approach to R-BET is interesting in that it leverages two very different approaches to GUI testing. R-BET is able to rely on the experience and intuition of a human tester through exploratory testing to record a test script, which can be visualized as a specific path through the state space of an application. With manual testing alone, it is not practical to explore much of this state space, so R-BET uses automated rule-based testing to test the state of the system close to the path identified through exploratory testing. The rules used in R-BET include preconditions, which can prevent a rule from firing when the context of the application doesn't make sense for it to do so. This can reduce the number of false failures caused by changes to a GUI. Rules are also intended to be small in scope, and to verify a small part of an application repeatedly over many states. Because of this, R-BET will not comprehensively test an application, but is expected to test the parts of an application that it does test more thoroughly than through manual testing alone.

While R-BET does not directly overcome the issues mentioned in Section I, it does provide a reasonable method of mitigating their impact on GUI testing. By relying on human intelligence and intuition, R-BET avoids issues associated with complexity. By leveraging automated rules, R-BET is able to provide regression support that is impractical with manual approaches alone and to avoid some of the problems associated with change. Through the combination of the two, R-BET is able to partially overcome issues with verification by strongly verifying that rules are upheld when their preconditions are met. However, this doesn't address the difficulty of creating verifications in the first place, so if it is difficult to create an automated verification for functionality, it will still by difficult with R-BET.

The usefulness and practicality of R-BET was explored through investigations into the four research questions posed in Section I. The purpose of this study is to determine whether R-BET, as implemented in LEET, is a methodology of which software testers would be able to make use. The first question, "Can rule-based exploratory testing be used to catch general bugs," was investigated in Section IV.A. From this pilot evaluation it would appear that R-BET can be used to catch a large number of high-level, general bugs by using a small number of short rules.

The second question, "Can rule-based exploratory testing be used to catch specific bugs," was investigated in Section IV.B. The pilot evaluation suggests that rule-based testing may be used to catch low-level, specific bugs that occur only when specific interfaces are used, but that keyword-based testing is problematic when used in these situations. 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 third research question, "How often is it possible to use keyword-based testing on GUIs," in Section IV.C. This section of the pilot evaluation suggests 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. Second, future implementations of systems that support R-BET could make use of a similarity-based system for widget lookup 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 fourth question, "Is rule-based exploratory testing less effort than writing equivalent tests using a capture/replay tool and inserting verifications manually," only got a preliminary answer from our 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.

In this study, it was shown that R-BET is usable in a variety of testing situations. For future studies, a tool that leverages testing with object maps should be developed. This tool should be used to compare R-BET to fully-manual and fully-automated approaches to GUI testing. These comparison studies should investigate deeper questions about R-BET, including:

- Is R-BET better at detecting bugs than manual or automated approaches?
- Is it faster to run tests using R-BET or an automated approach?
- Is it faster to create tests using R-BET than it is to create scripts for manual testing?
- What type of bugs are missed by R-BET, but found by manual or automated approaches?

Future work should also focus on the creation of a set of reusable rules for common situations. This would not only decrease the amount of effort required for testers who are looking to get started with R-BET, but it would also decrease the level of expertise required to perform thorough GUI testing.

VI. WORKS CITED

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

[2] 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.

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

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] A. Holmes and M. Kellogg, "Automating Functional Tests Using Selenium," in *AGILE 2006*, 2006, pp. 270-275.

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

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] (2010, April) CWE - Common Weakness Enumeration. [Online]. http://cwe.mitre.org/data/definitions/358.html