ABSTRACT
This paper presents a specification-driven approach to test automation for GUI-based JAVA programs as an alternative to the use of capture/replay. The NetBeans Jemmy library provides the basic technology. We introduce a GUI-event test specification language from which an automated test engine is generated. The test engine uses the library and incorporates the generation of GUI events, the capture of event responses, and an oracle to verify successful completion of events. The engine, once generated, can be used to test multiple versions of the application. The approach defined in this paper provides a language front-end to the Jemmy library to eliminate the programming usually needed to use this Java API. Results from applying the specification-driven approach to automate the grading of student programs indicate the feasibility of this approach. The specification-driven approach is equally useful for testing during development and regression testing. The primary benefit is that testers can focus on test case design rather than building test harnesses. This approach supports N-version testing, where each version of the application is intended to satisfy the same specification, and where each version is tested in an identical manner.

1. INTRODUCTION
This paper addresses the problem of test automation for GUI-based JAVA programs. GUI-based programs are event driven, where an event is triggered when the user interacts with the program through the GUI. Common user interactions include moving or clicking the mouse, selecting a graphic object, typing into a text field, or closing a window. The fundamental differences between event-driven programs and data-driven programs complicate test automation. Simple test automation involving input or output redirection that is adequate for data-driven programs will not suffice for GUI-based testing which requires a combination of data and event stimuli. Special tools are needed to simulate inputs and user actions that occur through the graphical user interface.
Traditional test automation for GUI-based software has relied on the capture/replay techniques. These techniques are marketed as labor saving tools for regression testing where the focus is ensuring that a later version exhibits the same behavior as an earlier version under the same set of stimuli. The utility of capture/replay techniques for testing during software development testing is questionable since no correct execution has yet been recorded. As seen in Figure 1, capture/replay tools do provide rudimentary capture scripts that can be edited to account for incremental changes to expected behavior or test actions. As a rule, these scripting languages are difficult to use, especially when the software or the test scripts are evolving. The maintainability of captured test scripts has come under question: it is possible that changes to screen layouts may invalidate the stored script. Finally, capture/replay tools often require a separate verification process.

The enabling technology for this research is the Jemmy library [Skrivanek02], an open source Java™ library. The Jemmy library contains methods to reproduce all user actions that can be performed on Swing/AWT GUI components, such as pushing buttons or typing text. The reflection API is used to identify and access GUI components defined inside the software under test. The Jemmy library permits the development of a test harness capable of generating application-specific GUI events, capturing the responses to the events, and verifying correct behavior.

This paper makes the following contributions. We introduce a test specification language in which tester actions can be described in terms of user interactions with the GUI components. Next we provide a generator that converts the test specification into an event-driven test engine that uses Jemmy to generate GUI events that implement the user actions given in the test specification. The test engine also includes a test oracle to verify that correct responses occur [Author03].

The organization of this paper follows. In section 2 we overview features of the Jemmy library that are relevant to this paper. Section 3 introduces the architecture of our solution, the automated Java GUI testing environment (AJGUTE). In section 4 we introduce the test specification language. Section 5 presents a simple GUI-based application assigned in a programming course for which AGUITE was used to grade student programs. Section 6 reviews related work. Conclusions and suggestions for future work appear in section 7.

2. THE JEMMY LIBRARY

   An alternative to commercial capture/replay tools is the Java GUI testing framework provided by the Jemmy [Netbeans, Skrivanek02] open source library released by NetBeans in 2002. Jemmy contains methods to produce most user actions that can be performed on Swing or AWT user interface components. The Jemmy test engine JTE is a Java program that generates the sequence of events required to test a given program according to a test plan. Executing the driver has the effect of replay; user sequences are coded (recorded) inside the test engine rather than stored as a separate file.

   The Jemmy approach is an attractive alternative to commercial and public domain capture/replay tools for the following reasons: A major shortcoming of capture/replay testing tools is that they rely on program execution to establish the standard of correctness. A better alternative is to define the standard test solely in terms of the program specification. We propose such a specification-driven approach where test cases and expected results are defined first. Next, a single test driver program is developed to generate the GUI interactions containing the test cases and to verify test results. Test execution and results verification are combined into a single program. Our approach eliminates the need for manual testing, external oracle support,
and capture/replay. Our approach is Java-specific, and is based on the Jemmy testing framework.

3. THE AUTOMATED JAVA GUI TESTING ENVIRONMENT

Figure 2 depicts the architecture of the automated JAVA GUI testing environment (AJGUITE). The boxes inside the dotted box show where automation is provided and the stages of processing. The initial step is the preparation of a specification for the software to be developed in terms of required GUI and functional behavior. The tester uses the software specification to develop a GUI based test specification consisting of manipulations of GUI input controls and the expected responses visible in the GUI. (The test specification language is illustrated in sections 4 and 5.) The Jemmy Test Engine Generator (JTEG) translates the test specification into a Java program, the Jemmy Test Engine (JTE), which uses the Jemmy API to access GUI components defined in the software under test. JTE also contains an oracle that accesses GUI output controls to verify that expected responses to input actions occur; verification outcomes are written to a test log for the software under test. Unless changes occur to the program specification or the test specification, JTE can test multiple versions of the software under test. The example given in section 5 takes advantage of this feature to use AGUITE to grade Java programs in a programming class.

4. THE EVENT-BASED TEST SPECIFICATION LANGUAGE

A goal of AJGUITE is to eliminate programming the test harness using the Jemmy API. Instead, the tester learns a high-level event-based test specification language having the following properties:

- **Easy to write.** The tester can readily learn to write a test specification based on the program specification and test plan.
- **Easy to understand.** The test specification file is easy to read, easy to review for correctness and completeness, and easy to modify for a different program assignment.
- **Amenable to automation.** The specification language has an underlying syntax that can support automated generation of the Jemmy test engine source code.

The test specification language is illustrated by the test specification given in Table 4. A small number of features is required.

- The name of Java software (class) to be tested.
- The GUI components, described in terms of type, significant identifying text (label, caption etc), and creation order within a parent container if there are multiple components of the same type.
- Test cases, specified in terms of simulated user actions and inputs, the corresponding expected data and, optionally, a numeric deduction to be applied if expected results are not found (this feature is used in grading student programs)

Tables 1 and 2 illustrate the keywords in the specification language that are used to specify simulated actions (Table 1) required by a test case. Keywords are also provided for specifying expected results in terms of values stored in text fields, selection status of input controls, and visible properties of
the GUI display. Verification keywords permit exact matching and matching within value ranges.

<table>
<thead>
<tr>
<th>KeyWord</th>
<th>Simulated Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>enter</td>
<td>Type the text in JTextField or JTextArea</td>
</tr>
<tr>
<td>push</td>
<td>Push the JButton</td>
</tr>
<tr>
<td>selectItem</td>
<td>Select the item from JComboBox or JList by index or by value.</td>
</tr>
<tr>
<td>changeSelection</td>
<td>Change the selection of JCheckBox or JRadioButton</td>
</tr>
<tr>
<td>clear</td>
<td>Clear the text in JTextField or JTextArea</td>
</tr>
</tbody>
</table>

Restrictions in naming conventions and creation order of GUI components apply equally to the developer and the writer of the test specification. Failure to adhere to restrictions can result in the implementation of unspecified behavior that can not be triggered by a test execution. In such cases the application may enter a state for which the automated test contains no user action that will cause a transition. Beyond this point, all automated test actions have no effect. This problem is common to automated GUI testing.

<table>
<thead>
<tr>
<th>KeyWord</th>
<th>Comparison Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>equals</td>
<td>Compare the expected text string to the text being displayed in a JTextField or JTextArea</td>
</tr>
<tr>
<td>contains</td>
<td>Check whether the value displayed in a JTextField or JTextArea contains a specific substring.</td>
</tr>
<tr>
<td>matches</td>
<td>Check if there is one checked text is displayed in JTextField or JTextArea</td>
</tr>
<tr>
<td>between</td>
<td>Check whether a numeric value displayed in a JTextField or JTextArea falls within a numeric range</td>
</tr>
<tr>
<td>itemSelected</td>
<td>Check whether a specific value from a JComboBox or JList is currently selected</td>
</tr>
<tr>
<td>isSelected</td>
<td>Check whether a JCheckBox or JRadioButton is currently selected</td>
</tr>
</tbody>
</table>

Table 3. Test Plan for the Currency Converter Program

<table>
<thead>
<tr>
<th>Test case</th>
<th>Expected Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select &quot;USD – U.S. Dollar&quot; in Currency Type 1; Enter &quot;200.0&quot; in Currency Amount 1; Select &quot;JPY – Japanese Yen&quot; in Currency Type 2; Push &quot;Convert&quot; button</td>
<td>&quot;21640.34&quot; or &quot;21620&quot; in Currency Amount 2</td>
</tr>
<tr>
<td>Select &quot;EUR – The Euro&quot; in Currency Type 2; Enter &quot;200.0&quot; in Currency Amount 2; Select &quot;CAD – Canadian Dollar&quot; in Currency Type 1; Push &quot;Convert&quot; button</td>
<td>&quot;269.82&quot; or &quot;270.16&quot; in Currency Amount 1</td>
</tr>
</tbody>
</table>

The complete test specification for the Currency Converter program is shown in Table 4.
Table 4. Complete test specification for the Currency Converter Program

```<testspec>
<class> CurrencyConverter </class>
<component>
  type = JComboBox; order = 1;  type = JComboBox; order = 2; type = JTextField; order = 1;
  type = JTextField; order = 2;     type = JButton; name = “Convert”
</component>
<testcase>
  number = 1
  <action>
    type = JComboBox; order = 1; name = “Currency Type 1”; action = selectItem 2;
    type = JTextField; order = 1; name = “Currency Amount 1”; action = enter “200.0”
    type = JComboBox; order = 2; name = “Currency Type 2”; action = selectItem 4;
    type = JButton; order = 1; name = “Convert”; action = push
  </action>
  <expect>
    type = JTextField; order = 2; name = “Currency Amount 2”;
    content = between 21620 and 21621; deduction = 1
  </expect>
</testcase>
<testcase>
  number = 2
  <action>
    type = JComboBox; order = 2; name = “Currency Type 2”;
    action = selectItem “EUR-The Euro”
    type = JComboBox; order = 1; name = “Currency Type 1”; action = selectItem 4
    type = JTextField; order = 2; name = “Currency Amount 2”; action = enter “200”
    type = JTextField; order = 1; name = “Currency Amount 1”; action = clear
    type = JButton; order = 1; name = “Convert”; action = push
  </action>
  <expect>
    type = JTextField; order = 2; name = “Currency Amount 2”;
    content = matches “269.82|270.16”; deduction = 1
  </expect>
</testcase>
<testspec>
```

6. RELATED WORK

Test automation can be applied at any stage in the testing lifecycle, but it represents an investment that may not always have the expected payback. Automation in GUI-based testing has focused on three stages, test case generation [Chen01, Memon01a, Ostrand98], test execution, and results verification [Takahashi01].

Several approaches have been taken to test case generation. Memon, Pollack and Soffa [Memon01b] exploit planning techniques developed and used extensively in artificial intelligence to automate the generation of test cases for GUIs.

Given a set of operators, an initial state, and a goal state, the planner produces a sequence of the operators that will transform the initial state to the goal state. They have demonstrated that this technique is both practical and useful by generating test cases for the GUI of the Microsoft WordPad software. But the test case generator is largely driven by the choice of tasks given to the planner. Currently in the test case generation system they have developed, these tasks are chosen manually by the test designer. A poorly chosen set of tasks will yield a test suite that does not provide adequate coverage.

Memon, Soffa and Pollack [Memon01b] present white-box like GUI-coverage criteria to help determine whether a GUI has
been adequately tested. They also provide an oracle for state verification. These coverage criteria use events and event sequences to specify a measure of test adequacy. Since the total number of permutations of event sequences in a GUI is extremely large, the automation of the generation of test sequences to ensure coverage of the GUI is essential.

Ostrand, Anodide, Foster and Goradia [Ostrand98] have implemented an experimental Test Development Environment (TDE) intended to raise the effectiveness of tests produced for GUI systems and to raise the productivity of the GUI system tester. TDE replaces the low-level scripting language of the capture/replay tool with a high-level scenario language and provides a visual model for modifying and creating variations of recorded sequences. The TDE interface is very simple to use, easy to learn, and leads to test designs that expand to a large number of test cases to test features of a typical GUI system. For example, they report that a tester using the TDE prototype to generate test scripts for the user interface to a medical diagnosis machine needed less than 30 minutes to create a single test scenario that produced more than 2500 test cases.

Most existing automation tools usually cannot efficiently compare graphic objects, including data controls. In these cases testers must visually inspect output screens to verify results. Sitting in front of a computer to verify test results manually negates the benefits of automated testing. Takahashi [Takahashi01] has developed a reliable scheme for automating the verification of graphic output. His approach involves intercepting calls to graphical API routines. Takahashi’s work extends content-based verification beyond the comparison of the content of text fields.

7. CONCLUSIONS AND FUTURE WORK

This work has established the feasibility of an event-based, specification-driven approach to testing GUI-based Java programs. An appealing feature of this approach is automation aimed at generating the test harness from the test specification. An additional benefit of a specification-driven approach is robustness over program versions, which is highly typical of development testing. The specification driven approach to testing also benefits development in a very simple way: the test specification is a specification of intended behavior.

There are several areas for future work. Currently AJGUITE handles only programs with a single frame. More complex Java applications will have multiple windows. Because AJGUITE is specification-driven, extensions to the test specification language will be required to distinguish GUI components and user actions associated with different frames. A second area to be investigated is interaction with multiple threads supporting distinct frames through the Jemmy API features.

The test specification language, which is in the style of HTML or XML, will be made fully XML compliant in order to take advantage of existing XML development tools such as XMLSpy 5, an industry standard XML Development environment for designing, editing and debugging XML [15].

As a final remark, the role of specification is critical to the successful use of AJGUITE. A sound, unambiguous software specification is essential for ensuring both the developer and tester have a common understanding of the software requirements and design constraints. In an ideal world, the developer and tester would co-develop the test specification. The final area of research is to provide an environment where such co-development can be facilitated.

8. ACKNOWLEDGMENTS

Our thanks go to conference referees for insightful feedback that improved the quality of the paper.

9. REFERENCES


