Barad: An Automated SWT GUI Testing Tool

by

Chip Killmar
Bachelor of Science in Computer Science
Harvey Mudd College

Report

Presented to the Faculty of the Graduate School of
The University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science in Software Engineering

The University of Texas at Austin
December 2007

Barad: An Automated SWT GUI Testing Tool

Approved by
Supervising Committee:

_____

_____

Dedication

To Katerina and Lola

To my mom, dad, and brother

“Testing can only prove the presence of bugs, not their absence.” – E. Dijkstra

Acknowledgements

First and foremost, I would like to thank Dr. Sarfraz Khurshid for his advice and support with respect to automated test case generation and to this report as a whole. I would also like to thank Svetoslov Ganov for his help in implementing Barad. Dr. Suzanne Barber provided valuable feedback on an earlier draft of this paper. Lastly, this work would not be possible without the hard work of the SWT development team.

November 30, 2007

Abstract

Barad: An Automated SWT GUI Testing Tool

Chip Killmar, M.S. Software Engineering

The University of Texas at Austin, 2007

Supervisor: Sarfraz Khurshid

Barad is an automated GUI testing tool capable of generating and executing test cases for SWT applications. Test cases are generated using a runtime model of the system under test (SUT) that can achieve full coverage of GUI states. Several partial order reduction techniques that improve the performance of the test case generation algorithm are described. The client-server software design is also discussed. A JVM Tool Interface (JVMTI) agent and Java Remote Method Invocation (RMI) server called Barad Agent helps promote ease of use and has advantages over other in-process tools. Defects found in the SUT during test case execution, derived test cases, and a widget hierarchy visualization aid are presented in a GUI called Barad Studio.

Table of Contents

List of Tables

List of Figures

Introduction

GUI testing tools have been around for more than a decade[1] but many continue to suffer from the same problems since their introduction. A major problem for manual GUI testing tools is that writing test cases with them can be time consuming for complex GUIs, and the process requires a test engineer with a relatively high amount of knowledge about the SUT[2] in order to perform the role of test oracle. After test cases have been written, it can be difficult even for an experienced engineer to evaluate how well they cover desired functionality and whether there is any overlap among them. Ideally, a test engineer strives for a set of test cases that exhibit full coverage because this produces the highest chance of finding defects and for no overlap because this translates to lower cost and effort involved in writing and maintenance.

Tools that automate test case generation and error discovery aim to reduce the burden by eliminating the need to manually author test cases. While making it easier for a test engineer, test case generation can be slow because of the vast number of states that need to be tested in a large GUI. Compounding this problem is the difficult task of configuring the SUT to be tested by a particular tool. For instance, the popular open source Java GUI testing tool Abbot requires information about the SUT such as the main class, the Java classpath, and system properties that make up the SUT's environment in order to launch it. This information is esoteric to someone not intimately familiar with the SUT. On top of this, tools that automate GUI tests in-process must keep the tool GUI responsive during testing or they will be difficult to use. For example, this includes a modal dialog window opened in the SUT while interacting with the tool GUI, and

[1] See [1].
[2] System Under Test, or the system being tested.

intercepting calls to `System.exit()` made by the SUT in order to prevent the process from exiting abruptly and causing data loss.

Barad is an automated GUI testing tool capable of generating and executing test cases for Standard Widget Toolkit (SWT) applications. SWT is the Java UI toolkit used by the Eclipse Platform and most other Eclipse projects that is implemented by creating thin JNI wrappers for the underlying operating system's GUI APIs[3]. SWT has been chosen because it is a stable platform for UI development and has been adopted by many open source and commercial applications such as the Eclipse IDE and Azureus. An SWT application typically uses many widgets to create the GUI, which may include Button, Text, Tree, Table and a wide assortment of other available widgets included with SWT.

Key requirements for the tool are speed, coverage, and ease of use. Test cases are generated using a runtime model of the SUT that can achieve full coverage of GUI states and output a minimal set of covering test cases. Several partial order reduction techniques and optimizations have been made to improve the performance of the test case generation algorithm and are described. A client-server design is used with the goal to make Barad easy to test with. A JVM Tool Interface (JVMTI) agent called Barad Agent has advantages over other in-process tools and is relatively easy to configure and capable of testing different versions SWT that exist in the field. Barad can detect defects via uncaught exceptions that are thrown by the SUT at runtime. These defects, detailed information about generated test cases, and a widget hierarchy visualization aid are presented in a GUI called Barad Studio. The end user interacts with Barad Studio in order to connect to a running instance of Barad Agent and start, pause, and stop test case generation. This report concludes with a discussion of future ideas that will enhance the tool, such as improving the execution performance of the test case algorithm.

[3] See [3].

Test Case Generation

A GUI is modeled as a set W of widgets. Each widget $w \in W$ has a set P of properties, each of which in turn has a set V of values. The set P consists of properties `{isEnabled(), getText(), isVisible(), getSelection()}` and V is the set of return values from invoking the method corresponding to each element in P. The state of the GUI at a point in time is the set of all triples (w, p, v), $p \in P$ and $v \in V$. For example, one triple for the current state of the GUI containing the Find button in the Find dialog in Figure 1 is $(\texttt{Find Button}, \texttt{getEnabled()}, \texttt{false})$.

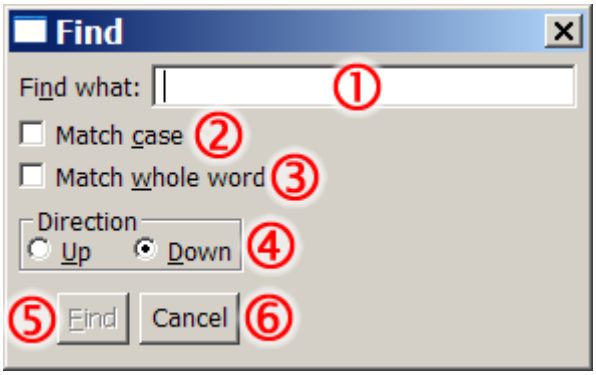

Figure 1: SWT Find dialog.

The test case generation algorithm in Table 1 is a combinatorial search algorithm that explores all combinations of widget-event pairs called test steps, (w, e), where $e \in E$. The set E consists of the events `{LEFT_CLICK (LC), ENTER_TEXT (ET)}` that can be executed on a widget in an automated fashion via a UI robot. Test cases themselves consist of an ordered list of one or more test steps.

Each iteration of the algorithm processes a collection of test cases known as the result set. To attempt to generate new test cases, an existing test case from the result set

is processed by executing all of its test steps and then determining the set of possible next test steps. From the set of possible next steps, a new collection of candidate test cases is formed by adding a possible next step such that the candidates differ from the parent test case only by last test step. Each candidate test case is then executed at which point the state of the GUI is measured and compared to a set of remembered states. If the state is unique, i.e. it does not exist in the set of remembered states, the candidate test case is added to the result set; otherwise it is pruned. The algorithm continues until it determines that no new unique test cases exist.

```
 1 Set<TestCase> uniqueTestCases = new LinkedHashSet<TestCase>();
 2 Set<TestCase> candidateTestCases;
 3 boolean newUniqueTestCasesFound;
 4 do {
 5     newUniqueTestCasesFound = false;
 6     candidateTestCases = new LinkedHashSet<TestCase>();
 7     // Generate candidate test cases from possible next steps.
 8     for (TestCase testCase : uniqueTestCases) {
 9         returnToInitialState();
10         executeTestCase(testCase);
11         List<TestStep> nextSteps = generateNextSteps();
12         for (TestStep nextStep : nextSteps) {
13             TestCase candidateTestCase = (TestCase) testCase.clone();
14             candidateTestCase.add(nextStep);
15             candidateTestCases.add(candidateTestCase);
16         }
17     }
18     // Determine uniqueness of candidate test cases (pruning).
19     for (TestCase candidateTestCase : candidateTestCases) {
20         if (!pruneTestCase(candidateTestCase)) {
21             if (uniqueTestCases.add(candidateTestCase)) {
22                 newUniqueTestCasesFound = true;
23             }
24         }
25     }
26 } while (newUniqueTestCasesFound);
```

Table 1: Test case generation algorithm.

For example, the Find dialog in Figure 1 illustrates a simple GUI dialog consisting of 9 widget instances: 1 - Label (Find what label), 1 - Text (Find what text field), 6 - Button (2 checkboxes, 2 radio buttons, Find and Cancel buttons), and 1 - Group (Direction). Given this dialog, test case generation begins by creating an initial set of candidate test cases from the set of possible next test steps.

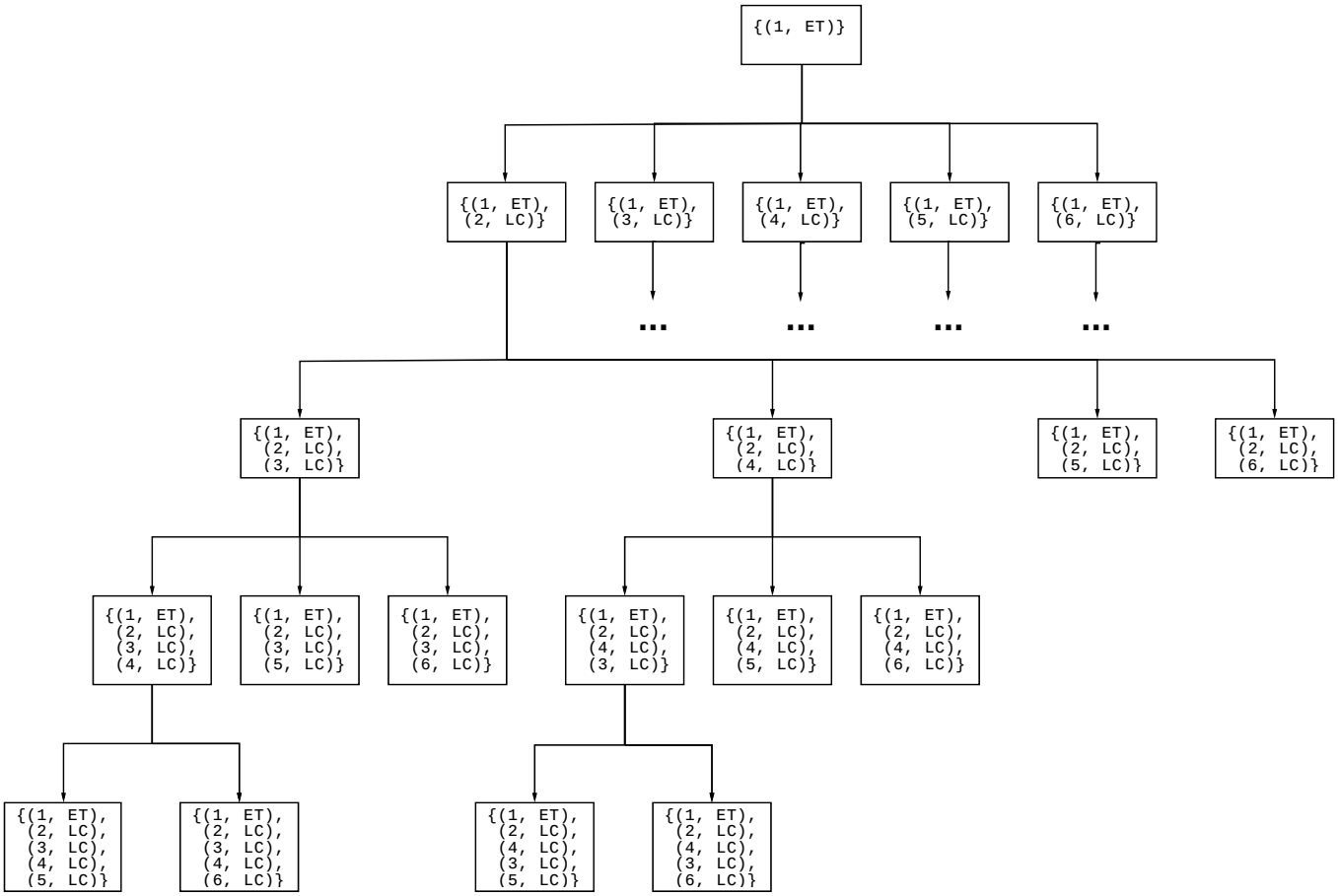

Figure 2: Test cases generated for the Find dialog example.

Widgets in Figure 1 that can be part of a test step are indicated with a red numbered circle next to the widget. In this case, possible next steps become the first test cases: {(①, ET)}, {(②, LC)}, {(③, LC)}, {(④, LC)}, and {(⑥, LC)}. Note that ⑤ is not a possible next step because it is disabled, but that it becomes enabled when any text is entered into ①. Also, the Label and Group widgets aren't considered by the algorithm because they don't respond to user input. The next iteration of the algorithm takes each existing test case and adds a next possible test step. For existing test case {(①, ET)}, this results in test cases {(①, ET), (②, LC)}, {(①, ET), (③, LC)}, {(①, ET), (④, LC)}, {(①, ET), (⑤, LC)}, and {(①, ET), (⑥, LC)}. Figure 2 shows the full list for the branch of the search space starting with {(①, ET)}.

Tool Design

A key requirement for the tool is ease of use in terms of configuring it to test an SWT application. Another important requirement is the ability to generate and execute test cases in the SUT without interfering with its GUI operations, which can happen when the tool and SUT run in the same process. Using a client-server architecture, both of these requirements can be achieved. Figure 3 illustrates the overall tool design. Barad Agent is a server that is launched by the SUT. Once the server is running, Barad Studio can connect as a client and send and receive information to and from Barad Agent. A client-server design also allows for remote GUI testing in addition to testing locally in the same process space.

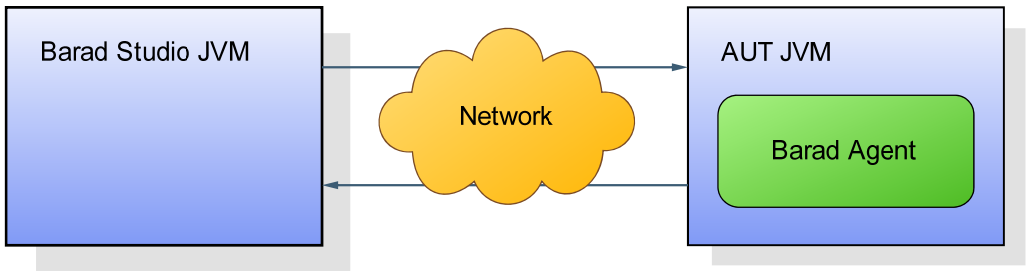

Figure 3: Barad design.

Barad Agent

The server component, Barad Agent, monitors the widget hierarchy in the SUT and provides the implementation for test case generation. Barad Agent consists of two components (Figure 4), a JVMTI agent (JAG) and a Java server (JAS). JAG is implemented as a JVM Tool Interface (JVMTI) agent. JVMTI is a programming interface used by development and monitoring tools that provides a way to inspect the

state and control applications running in a JVM. JVMTI agents are C or C++ native shared libraries that can be configured to load in-process automatically by the SUT JVM. It is not uncommon that modifying configuration files in the SUT to update parameters such as the Java classpath is required in order to hook into the SUT. Alternatively, JVMTI is a convenient approach to bootstrapping the server because very few changes are required to the SUT and computing environment. Setting two environment variables, `BARAD_HOME` and `JAVA_TOOL_OPTIONS`, are the only changes needed begin testing with the agent.

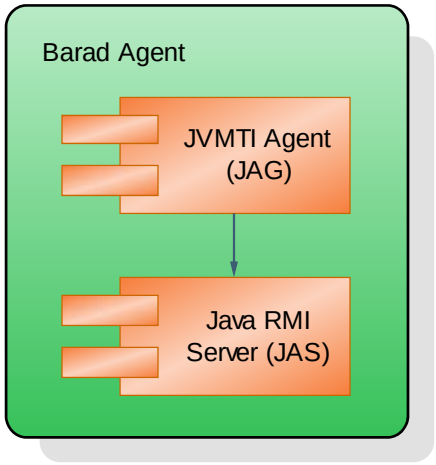

Figure 4: Barad Agent component diagram.

JAG uses callback events to bootstrap JAS, a Java Remote Method Invocation (RMI) server capable of connecting to Barad Studio. RMI enables distributed Java applications in which the methods of remote Java objects can be invoked from other JVMs, possibly on different hosts, using object serialization to marshal and unmarshal parameters[4]. The sequence of events for bootstrapping JAS presented in Figure 5 are: 1)

[4] See [4].

SUT executable is launched; 2) JVM loads JAG; 3) JAG launches JAS; 4) JAS waits for connections from Barad Studio; 4) main() is called on SUT main class; 5) SUT GUI is running and ready for testing.

Once connected to JAS, information about the SWT entire widget hierarchy in the SUT and the state of each widget can be sent to Barad Studio. Widgets in the SUT can be directly manipulated, e.g. clicking on a widget or typing a keystroke, using a GUI robot that can post event objects into the SWT event queue. Low-level robot commands can then be used to generate test cases which can be displayed in Barad Studio. Table 2 shows the list of RMI methods implemented by JAS.

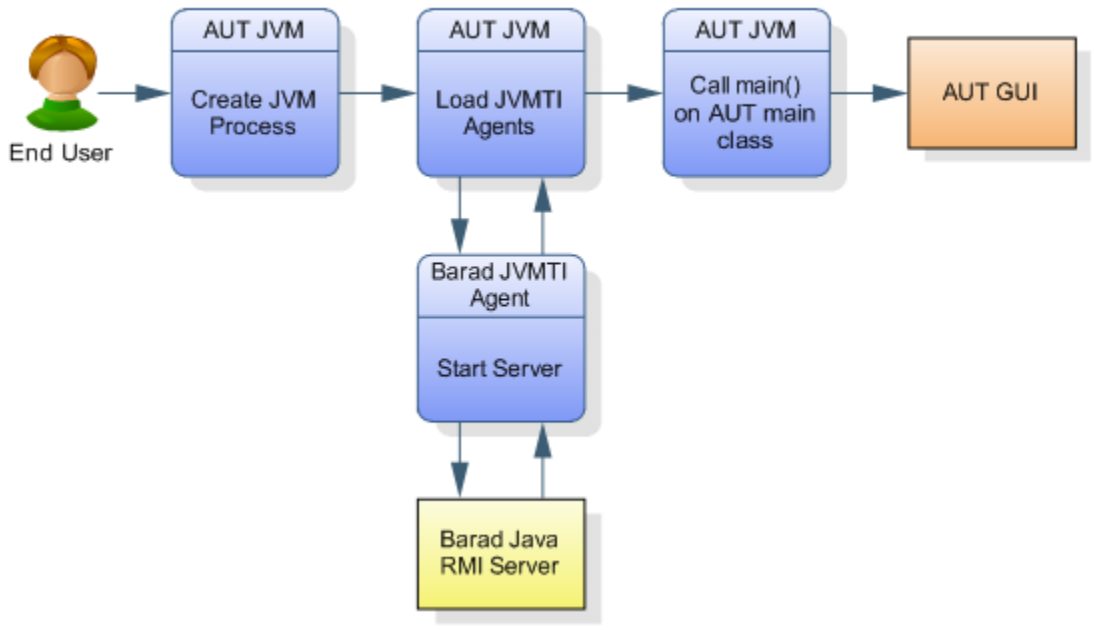

Figure 5: JAS bootstrap sequence diagram.

Java Reflection proxies for SWT widget instances are used between JAS and the SUT. A proxy is an object-oriented design pattern that employs a proxy wrapper class to delegate method calls made on it to a real contained instance. In this case, proxies have

been created to wrap instances of SWT widgets in the SUT. The level of indirection afforded using this technique translates to no compile-time dependencies on SWT. This has the benefit of making the agent compatible with any potential test application without changing the SWT library it uses, which if required could cause errors in an SUT developed using a specific version of SWT.

```
1     // Returns the root node of the widget hierarchy.
2     WidgetInfo getWidgetHierarchy(boolean rebuild);
3     // Returns the state of a widget as a value object.
4     WidgetValues getWidgetValues(WidgetInfo widgetInfo);
5     // Test case generation methods.
6     void startGenerateTestCases();
7     void stopGenerateTestCases();
8     void pauseGenerateTestCases();
9     void continueGenerateTestCases();
10    ExecutionState getExecutionState();
11    TestCase[] getGeneratedTestCases();
12    TestCase[] getExecutingTestCases();
13    TestCase getCurrentTestCase();
14    String getThrowableStackTrace();
```

Table 2: RMI methods in JAS.

Original method calls made on the proxy are mapped to the corresponding method on the SWT widget such that the real method can be found for each proxy method. The method mapping, lookup, and invocation process of the real method from the proxy counterpart use Java reflection. If the return value is itself an SWT widget type, it is further wrapped in a proxy before finally be returned. Conveniently, the proxies maintain the appearance that APIs are invoked directly on the SWT widget.

Barad Studio

End users interact with the tool through a client application called Barad Studio. The application is an SWT JFace thick client application that allows connections to Barad

Agent. Once a connection is made, a test case generation screen controls the generation of test cases in the SUT. Another screen is used to report the current widget hierarchy and the value state of widgets in the SUT.

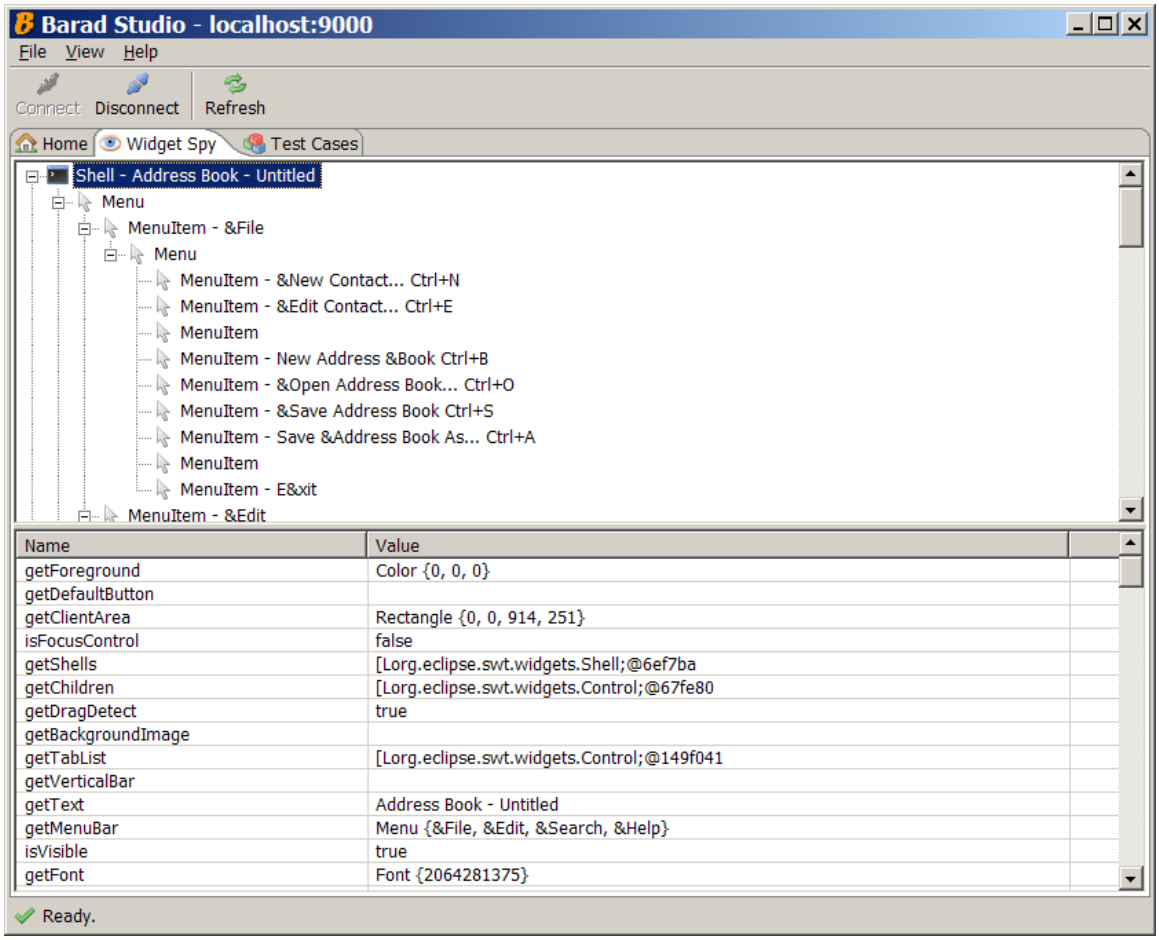

Figure 6: Widget spy in Barad Studio.

To facilitate the writing of the widget filter and initial state strategies, a widget spy is used to show a hierarchy of widgets in the SUT along with the current runtime values for a selected widget. The widget hierarchy and runtime values are retrieved from JAS once and cached for performance reasons, and the process of retrieval does not interfere with the functioning of the SUT. Cached values can be refreshed by the end

user after changes occur in the SUT in order to make the widget spy information consistent with current values the SUT.

Serializable data structures have been created for RMI communication between Barad Studio and Barad Agent. For example, `WidgetInfo` and `WidgetValues` (Table 2) data transfer objects contain hierarchy and runtime value information respectively without needing to serialize actual widget instances. Instead, the notion of a widget ID exists to convey information in a data structure to uniquely identify a widget in the SUT from a client based on characteristics such as widget type, parent widget ID, child index, and text. Likewise, runtime values are captured in a way that converts object references in JAS into scalar values that can be serialized to Barad Studio.

Figure 6 shows a screen capture of the widget spy for an example SUT that is an address book application. The name column is populated with all of the JavaBean property getter method names and the value column is populated with the `toString()` return value of the method invocation result occurring in JAS for each of the getter methods using Java Reflection.

As an example of how the widget spy can be a useful utility, Figure 6 demonstrates from the `getText()` method that the address book application's main shell widget has its title set to "Address Book – Untitled", and from the `getMenuBar()` method that it contains a menu bar with four top-level menus: File, Edit, Search and Help. In addition to this information, the child menu items of the File menu have been expanded to show the ordered children New Contact, Edit Contact, Separator, New Address Book, Open Address Book, etc.

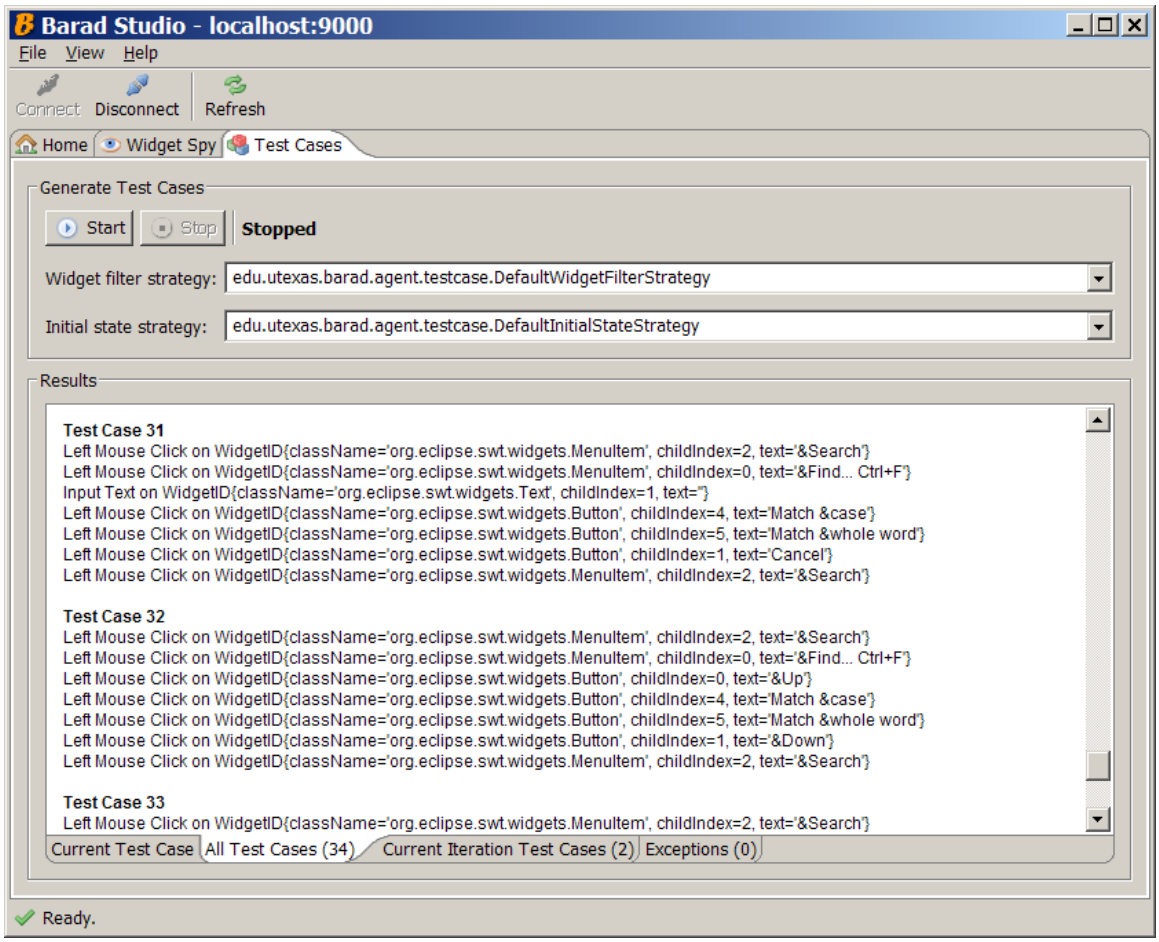

Figure 7: Test case generation in Barad Studio.

Test case generation has its own screen in Barad Studio. A widget filter strategy is used to specify what widgets in the SUT are available as next steps in a test case. This can be employed to limit the available widgets and thus focus testing on a part of the GUI, and to prevent unwanted test steps from being executed, such as a Close button that exits the SUT. An initial state strategy is custom logic for returning the SUT GUI to an initialized state. The advantage to having this strategy is speed: if the GUI process can be re-initialized without tearing down the process and restarting it, this can save considerable time during test case generation.

Custom classes for both widget filter and initial state strategies can be inputted in text fields as shown in Figure 7. Test case generation can be started and then paused during execution to monitor the progress of the generation algorithm. Feedback is presented in four tabs: Current Test Case, All Test Cases, Current Iteration Test Cases, and Exceptions. The Current Test Case tab displays the generated test case presently executing; All Test Cases shows a list of all uniquely generated test cases; Current Iteration Test Cases shows the list of candidate test cases that have yet to undergo comparison for uniqueness and pruning; and Exceptions will provide a stacktrace of any uncaught exceptions that have been thrown in the SUT during test case generation.

Future Work

There are several improvements that can be made in Barad. Major areas of focus include improving the performance of the test case generation algorithm, enhancing the defect detection logic, and adding more features to Barad Studio.

The current version of the test case generation algorithm uses brute force and will likely take a long time to finish for complex GUIs. If more states in the GUI could be pruned then it's believed that a faster algorithm would result. One method for achieving better pruning of states is symbolic execution of code paths in the SUT in order to determine a minimum set of input values required for complete coverage. Barad currently uses an end user supplied dictionary of strings and tries each string during test case generation. If the end user guesses what these values are, this can lead to slower performance because more test cases are generated than necessary. Symbolic execution of string values can instead determine this set of values and eliminate this inefficiency.

An important aspect of the tool is its ability to find bugs. The current implementation uses a default uncaught exception handler and assumes that any uncaught exception is a "crash" in the SUT. While in most cases this is a safe assumption, this approach fails if an underlying framework is used that itself catches exceptions in dependent code and prevents these exceptions from propagating. For example, Rich Client Platform (RCP) is an SWT plugin framework catches exceptions in a global exception handler in the method `WorkbenchAdvisor#eventLoopException(Throwable exception)`. This means using the current approach of detecting uncaught exceptions is not appropriate for RCP applications. However, using JVMTI there is a native callback method `ExceptionCatch()`which would allow for analysis to be performed when an

exception is caught, for example by `eventLoopException()`, and if so then fail the test case.

Appendix

Source code for Barad can be found online at http://code.google.com/p/barad.

Glossary

GUI: Graphical User Interface

SUT: System Under Test

SWT: Standard Widget Toolkit

RMI: Remote Method Invocation

JVM: Java Virtual Machine

JVMTI: JVM Tool Interface

JNI: Java Native Interface

IDE: Integrated Development Environment

API: Application Programming Interface

References

1. Abbot Java GUI Testing Framework. Available at http://abbot.sourceforge.net.

2. Azureus Java BitTorrent Client. Available at http://azureus.sourceforge.net.

3. Eclipse Foundation. SWT: The Standard Widget Toolkit. Available at http://www.eclipse.org/swt.

4. Gamma, E., Helm, R., Johnson, R., Vlissides, J. 1995. Design Patterns: Elements of Reusable Object-Oriented Software. First edition: Addison-Wesley.

5. Kepple, L. February 1994. The Black Art of GUI Testing: Automated testing in an event-driven environment. Dr. Dobbs Journal.

6. McCaffrey, J. September 2005. Low-Level UI Test Automation. MSDN Magazine.

7. Memon, A., Yuan, X. 2007. Using GUI Run-Time State as Feedback to Generate Test Cases. ICSE '07: Proceedings of the 29th International Conference on Software Engineering. Washington, D.C.: IEEE Computer Society.

8. Memon, A., Pollack, M., Soffa, M. 1999. Using a goal-driven approach to generate test cases for GUIs. ICSE '99: Proceedings of the 21st international conference on Software Engineering. Los Alamitos, CA: IEEE Computer Society.

9. Sun Microsystems, Inc. JVM Tool Interface Version 1.0. Available at http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html.

10. Sun Microsystems, Inc. Java Remote Method Invocation. Available at http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp.

Vita

Name: Henry “Chip” Merton Killmar III

Born: January 31, 1975 in Los Angeles, California

Parents: Henry M. Killmar Jr. and Genelle W. Killmar

Education: B.S., 1999, Harvey Mudd College

Major Fields of Interest:

Software Testing, Automated GUI Testing

Employment:

Software Architect, iTKO, Inc., 2005 –
Senior Software Developer, Cisco Systems, Inc., 2004 – 2005
Senior Software Developer, Forgent Networks, Inc., 2002 – 2004
Software Developer, Trilogy Software, Inc., 1999 – 2002
Software Developer, WolfeTech Corp., 1997 – 1999
Software Developer Intern, Compaq Computer Corp., 1997

Patent: Filed 2007, *Testing In-Container Software Objects*

Permanent Address:

2709 Grimes Ranch Rd., Austin, TX 78732

This report was typed by the author.