An Automated Testing Perspective of Graphical User Interfaces

Robert M. Patton
Oak Ridge National Laboratory
Oak Ridge, TN
pattonrm@ornl.gov

Gwendolyn H. Walton
Florida Southern College
Lakeland, FL
gwalton@flsouthern.edu

ABSTRACT

Software test automation provides hope for testers who face the daunting task of testing large, complex software systems. However, this hope often fades like a sunset as testing progresses from unit testing to system testing. One reason for this is that the interface between the system and its operating environment or the user becomes increasingly more complex as testing progresses from unit testing to system testing. In particular, the graphical user interface (GUI) has proven to be one of the most difficult challenges for test automation. This paper describes the efforts and difficulties of creating and using an automated test harness for the GUI of the U.S. Army's OneSAF Testbed Baseline (OTB) simulation system.

There is a significant amount of effort required for a user to create scenarios with the OTB user interface, even for simple scenarios. From a testing perspective, a test harness is needed to reduce this effort, thereby allowing more testing to be done within a given amount of time. Unfortunately, this generally requires that the test harness be custom made for the system being tested. This paper describes the methodology of creating a custom test harness for the OTB system. The goal in developing this test harness was to automate completely the process of creating scenarios in OTB using the GUI. This allows the tester to focus more on the execution of the scenario and determining its correctness. The test harness that was developed is capable of reading test scenarios provided by the tester, and then automates creating the scenario via the OTB user interface. Only partial test automation was achieved due to irregularities and complexities in the OTB user interface. This paper describes these problems and provides suggestions for improved user interface design strategies that can enhance possibilities for test automation of simulation system GUIs.

ABOUT THE AUTHORS

Robert M. Patton is a Postdoc Research Associate at Oak Ridge National Laboratory. His research interests include validation and verification of simulation systems, automated testing and analysis of software intensive systems, and automated systems. He received a Ph.D. in Computer Engineering from the University of Central Florida in 2002. His worked there focused on tool development to support statistical usage testing and military simulation testing. In addition, he has experience working as a Research Assistant with Oak Ridge National Laboratory. His worked there focused on the software development and simulation of cooperative mobile robotics.

Gwendolyn H. Walton is an Associate Professor of Computer Science at Florida Southern College. Her research interests include improved methods for specification and testing of large-scale systems. In addition to her academic experience, Walton has over 20 years of experience in all aspects of software engineering. Her previous positions have included President of Software Engineering Technology, Inc; Assistant Vice President, Division Manager, and Senior Systems Analyst for Science Applications International Inc; Senior Data Systems Programmer for Lockheed Missiles and Space Company, Inc; and Research Associate for Oak Ridge National Laboratory. She received a Ph.D. in Computer Science from the University of Tennessee in 1995.

An Automated Testing Perspective of Graphical User Interfaces

Robert M. Patton
Oak Ridge National Laboratory
Oak Ridge, TN
pattonrm@ornl.gov

Gwendolyn H. Walton
Florida Southern College
Lakeland, FL
gwalton@flsouthern.edu

INTRODUCTION

Software test automation provides hope for testers who face the daunting task of testing large, complex software systems. However, this hope often fades like a sunset as testing progresses from unit testing to system testing. One reason for this is that the interface between the system and its operating environment or the user becomes increasingly more complex as testing progresses from unit testing to system testing. In particular, the graphical user interface (GUI) has proven to be one of the most difficult challenges for test automation [1][2][3].

At the system level, the GUI provides a means for the user (in this case, a person, rather than another system) to interface with the system. In most systems, this involves the use of a computer monitor, keyboard, and mouse as shown in Figure 1. The system uses the monitor to communicate with the user using visual cues. The user uses the keyboard and mouse to communicate with the system.

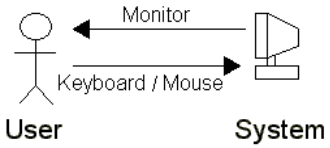

Figure 1. Human - Computer Interface

When the system communicates with the user, some level of error is tolerable. This is because the user is capable of adapting to variations. For example, a particular web site may look different using different web browsers, but people are still able to navigate the web site. If text was once displayed on the right hand side, but is now on the left, then the user recognizes the change and adapts. The system, on the other hand, is not so adaptable. As a result, the level of error is much less tolerable when the user communicates with the system (unless the system was built to handle and interpret some input error; however, this is usually not the case).

For automated testing, the human user (or human tester) is replaced with test automation software, called a test harness, which is capable of interfacing with the system to be tested. This allows control of the system to be automated. Unfortunately, this means that the level of error that was once tolerable when the system communicated to the user is no longer tolerable. Consequently, the GUI has proven to be a significant and difficult challenge for test automation.

This paper describes the efforts and difficulties of creating and using an automated test harness for the GUI of the U.S. Army's OneSAF Testbed Baseline (OTB) simulation system (version 1.0) [4]. The purpose is to introduce and provide a different perspective (a testing perspective) of GUIs, so that GUIs can be created that are not only easy and intuitive for users to use, but also facilitate automated testing.

BACKGROUND

There is a significant amount of effort required for a user to create scenarios with the OTB user interface, even for simple scenarios. A test harness is needed to reduce this effort, thereby allowing more testing to be done within a given amount of time or the same amount of testing in less time. Unfortunately, this usually requires that the test harness be custom made for the system being tested.

The goal in developing this test harness was to automate completely the process of creating scenarios in OTB using its GUI. This allows the tester to focus more on the execution of the scenario and determining its correctness. The test harness that was developed is capable of reading test scenarios provided by the tester, and then automates creating the test scenario via the OTB user interface. As will be discussed, only

partial test automation was achieved due to irregularities and complexities in the OTB user interface.

For this application, there were 200 test cases to be executed. Each test case was automatically generated using another testing tool [6]. The test cases represented one of two possible scenario concepts as shown in Figure 2 [3].

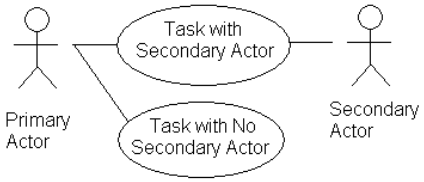

Figure 2. Two possible scenario concepts

Each scenario concept consists of some primary actor that performs some task. In order for the task to be issued to the Primary Actor, some of the tasks require a second entity (i.e., a secondary actor). For example, a "move" task does not require specifying some other entity. So, it can be issued to the Primary Actor. Other tasks, however, do require another entity. For example, "assault" tasks usually require specifying a second entity that should be assaulted. This entity would then be referred to as a Secondary Actor. These two basic scenario concepts provide the foundation for larger, more complex scenarios.

The test harness that will be described was created to read these test scenarios and then create them in OTB using its GUI.

TEST HARNESS DEVELOPMENT

To provide automation support for test case execution, the test harness mimics the input of a human tester (keyboard and mouse inputs), such that the OTB interface does not know the difference between the harness and a human tester.

For this application, the test harness was written in the Java programming language. The foundation of the harness is the Robot class that is part of the core Java programming language [5]. The Robot class provides a means to automate keyboard and mouse inputs (i.e., communication from the user to the system as shown in Figure 1). Figure 3 shows an example program that uses the Robot class to send mouse and keyboard events to the operating system.

```
1  import java.awt.Robot;
2  import java.awt.event.InputEvent;
3  import java.awt.event.KeyEvent;
4
5  public class example {
6
7      public static void main (String args[]) {
8          try {
9              // create instance of Robot
10             Robot TestRobot = new Robot();
11
12             // move the mouse to this location
13             TestRobot.mouseMove(30, 40);
14
15             // click the left mouse button
16             TestRobot.mousePress(InputEvent.BUTTON1_MASK);
17             TestRobot.mouseRelease(InputEvent.BUTTON1_MASK);
18
19             // type the text "aBc"
20             TestRobot.keyPress(KeyEvent.VK_A);
21
22             TestRobot.keyPress(KeyEvent.VK_SHIFT);
23             TestRobot.keyPress(KeyEvent.VK_B);
24             TestRobot.keyRelease(KeyEvent.VK_SHIFT);
25
26             TestRobot.keyPress(KeyEvent.VK_C);
27         }
28         catch (Exception e) {
29             e.printStackTrace();
30         }
31     }
32 }
```

Figure 3. Using Java's Robot class to simulate user input

The example code shown in Figure 3 does not do anything useful, except to show how to use the Robot class. For testing purposes, the harness was developed in three phases: mapping, parsing, and timing.

In the mapping phase, the user interface of the system to be tested was mapped so that screen coordinates for menus, toolbars, and other interface items are known. With this information, the test harness can mimic user inputs by representing a mouse click at a specific location on the screen. For example, suppose the test harness needs to click the "Save Scenario" button on the menu bar. Line 13 in the sample code of Figure 3 would need to have the screen coordinates of the "Save Scenario" button on the menu bar. With the correct coordinates, this would cause the mouse cursor to move to the button, and the following two lines of code (lines 16 and 17) in Figure 3 would then cause the button to be "clicked." Obviously, the drawback to this is that if the button is moved in a future version of the system, then the test harness will also need to be changed. However, there is currently no way to avoid this (other than having the test harness built into the system). The mapping phase is the process of finding the screen coordinates of the necessary parts of the GUI, which are specific to the screen resolution on the machine that will be used for testing.

The mapping of the OTB interface was a trial and error process of identifying the correct screen coordinates. Only the parts of the interface that were needed for testing were mapped. For this particular application, these items consisted of:

- File menu button
- Save Scenario menu button on the File menu
- Create Units button on the toolbar
- Done button on the Unit Editor screen
- Coordinate, Direction, and Competence input boxes on the Unit Editor screen
- Unit Type text box on the Unit Editor screen

Once the mapping phase was complete, the test harness would now be able to use the OTB GUI. However, it would not have the necessary data to input into the GUI.

To get the necessary input data, the next development phase was the parsing phase. In this phase, code was developed that parses the test cases looking for key words or phrases that identify important data (e.g., primary actor, task, secondary actor) to be used for executing the test case. Once the key words or phrases have been identified, then they are used in the harness to trigger certain events such as when to stop parsing, when to create a unit, etc. The following is a list of phrases or words that were used to parse the test cases generated for this particular application:

- "Attach Vehicle: "
- "Competence: "
- "Direction: "
- "Infantry Task Secondary Actor: "
- "Initialize: "
- "Load Items: "
- "Location: "
- "Notes:"
- "Pass"
- "Secondary Actor: "
- "Secondary Actor for Aircraft: "
- "Select Primary Actor: "
- "Select Secondary Actor: "
- "Select Terrain: "
- "Terrain: "
- "Test Script: "
- "Tester ID"

These key words or phrases are specific to the test cases generated for this application.

Once the mapping and parsing phases were complete, the two were "connected" together such that the test harness would parse the test cases to get the necessary input data and then send the appropriate events to the OTB user interface. However, it is possible that the test harness will send events to the GUI faster than the GUI can respond to the events. This results in events being "dropped" by the GUI. To compensate, the third phase of the harness development is the timing phase. In this phase, appropriate delays must be inserted into the test harness so that the harness will run as fast as possible as but not faster than the GUI. Like the mapping phase, this involves an initial trial and error process to identify where to place delays in the test harness code and for how long. As a rule of thumb, the places to insert delays are usually after input events that cause the GUI to update large sections of the screen (or some other lengthy background computation is performed). The length of the delay is directly related to the size of the screen section that needs updated (i.e., large screen section / large delay, small screen section / small delay or no delay).

DIFFICULTIES IN TEST AUTOMATION

During the development of the test harness, several issues were encountered because of inconsistencies in the OTB user interface. Specifically, the issues involved the "Unit Type" selection area of the "Unit Editor" screen, and the selection of tasks for entities. These inconsistencies are discussed in this section. Note that the discussion is from the perspective of testing, not from the perspective of the user. These are testability concerns, not necessarily human / computer interface concerns (although they may be related).

Figure 4 shows the default settings for the "Unit Type" selection area of the "Unit Editor" screen. Notice that the selection box under the heading "Experimental:" is set to "Operational," while all the others are set to "Any." In order for the test harness to select any entity provided by OTB, the selection box for "Experimental" must be set to "Any." This minor inconsistency causes an extra step to be required during test execution, which ultimately slows down testing.

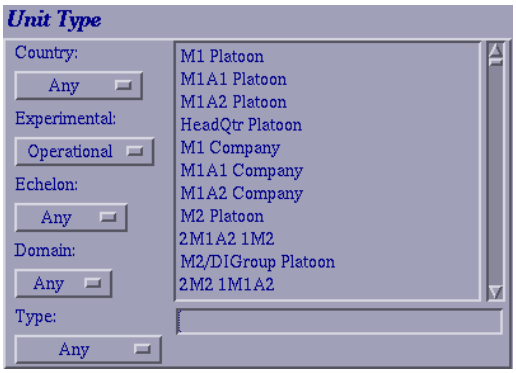

Figure 4. Default settings for Unit Type

Figure 5 shows the result of changing the selection box to "Any." Notice that an additional scroll bar (horizontal) has been added. As a result, the text box located underneath the listing has been moved downward (compare Figure 4 and Figure 5). Figure 8 shows an instance where the text box has been moved down even further.

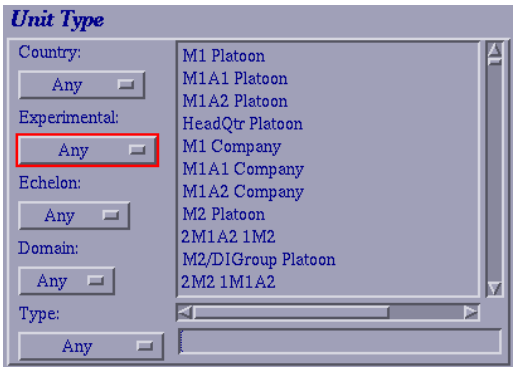

Figure 5. Unit Type display after the user changes Experimental to "Any"

From the user's perspective, this minor variation is not really a problem. The user is capable of adapting quickly. However, from a testing perspective, this is a very difficult issue to handle because the location of the text box varies at run-time and is not predictable. This means that the test harness must be developed to handle this inconsistency; otherwise, the test harness may not correctly execute the test cases. A better location for the text box would have been above the listing so that its screen position is not affected by changes in the listing. This probably would not have caused a significant impact to the user, and would have made automated testing easier.

The next issue with the "Unit Type" selection area concerns the automatic resizing of the selection area. This is shown in Figure 6. This figure has not been altered in any way. After the user changes one of the selection boxes on the left hand side, the listing of entities on the right hand side will automatically resize to fit the entity with the longest name exactly. Figure 4 shows the initial view of the selection area. Figure 5 shows the first view of the selection area after the user has set "Experimental" to "Any." Figure 6 shows the final view of the selection area after the user has set "Experimental" to "Any." The transition time between Figure 5 and Figure 6 does not occur consistently.

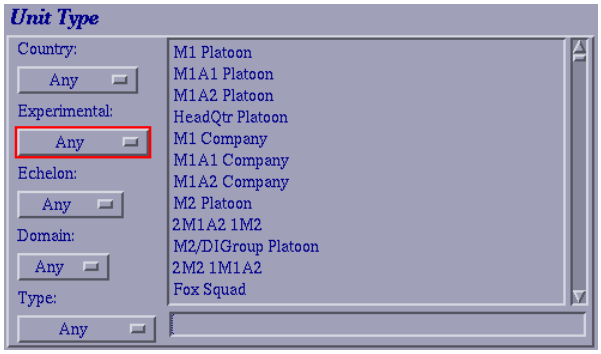

Figure 6. Unit Type display after auto-sizing for current selection

From a testing perspective, the automatic resizing causes a problem for automated testing. If the test harness needed to use the scroll bar to select an entity, it would be difficult to know when and where the scroll bar is located on the screen. To complicate further automated testing, it creates a "domino" effect such that other features of the GUI are affected that otherwise would not be. Finally, the transition time for automatic resizing is not consistent, which complicates automated testing even more.

In Figure 7, notice that the listing has again automatically resized to fit the names of the entities in the current listing. In this case, the selection area is smaller than the original size and both scroll bars have been removed.

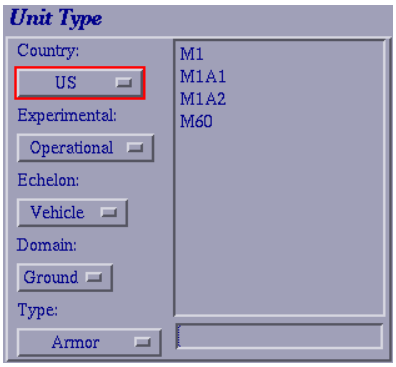

Figure 7. Unit Type display after auto-sizing for the current listing

Despite these issues, the "Unit Type" selection area does have a positive aspect, with respect to automated testing, in that it provides a text box that allows the user to type in the name of the entity to be selected. This is shown towards the bottom of Figure 8. Notice in Figure 8 that the user has typed "M1A1" in the text box (shown at the bottom), and the OTB user interface has automatically matched the name to an entity in the listing. The name of any entity can be typed into this text box, and, if it is in the listing, will be automatically selected. This particular behavior was crucial in developing the test harness. If this text box were not available, creating a test harness to select the correct entity automatically would have been much more difficult, if not impossible.

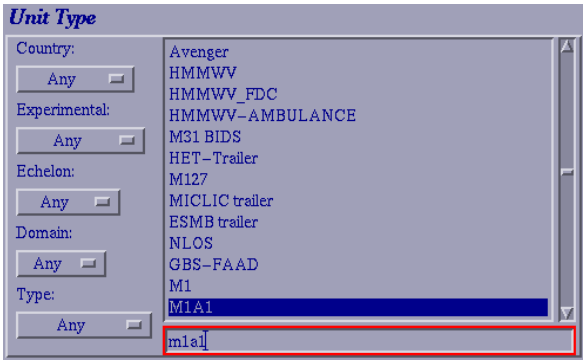

Figure 8. Unit Type display allows for typing in the name of desired entity

The next issue with the OTB user interface deals with the selection of tasks. Figure 9 shows the default task listing for the entities M3A3 and MARDER1A3.

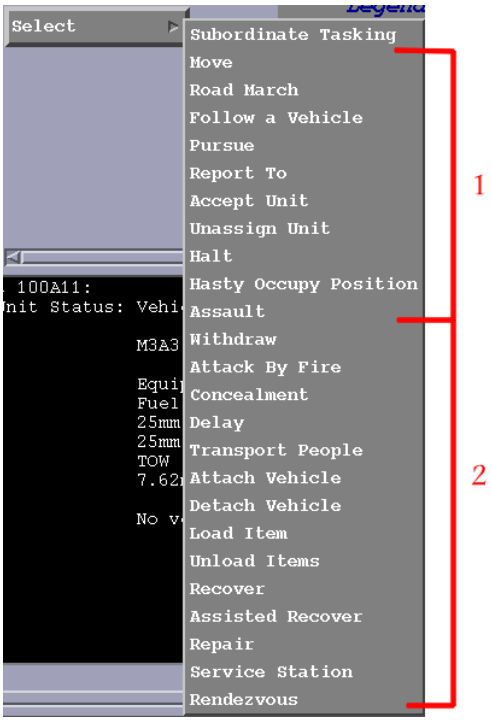

Figure 9. M3A3 (shown) and MARDER1A3 default task listing

The tasks in Figure 9 have been separated into two groups. Group 1 ranges from "Move" to "Assault." Group 2 ranges from "Withdraw" to "Rendezvous." The M3A3 is an American armored personnel carrier (APC), while the MARDER1A3 is a German APC. Both entities share exactly the same list of tasks. The listing of tasks for both entities is also displayed in exactly the same order, as shown in Figure 9.

The BTR-80 is a Soviet APC entity and is available for use in OTB. Its default task listing is shown in Figure 10.

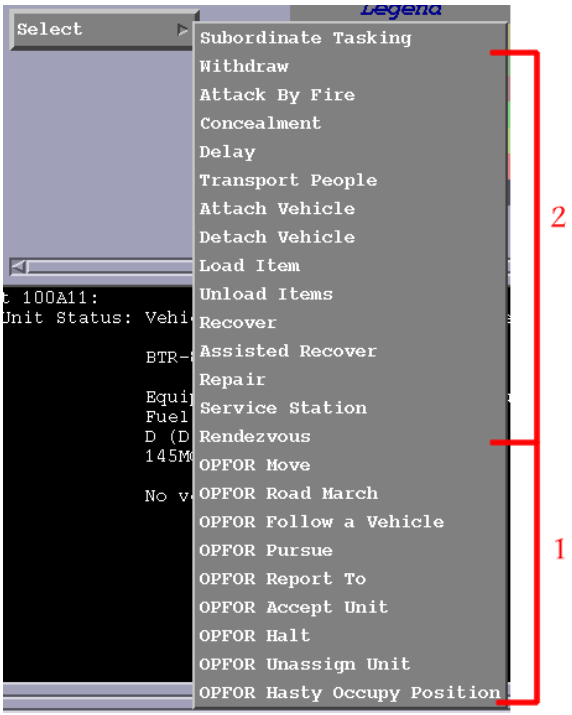

Figure 10. BTR-80 default task listing

The tasks in Figure 10 have been separated into two groups. Group 1 goes from "OPFOR Move" to "OPFOR Hasty Occupy Position." Group 2 goes from "Withdraw" to "Rendezvous." In comparing Figure 9 and Figure 10, notice that group 2 for the BTR-80 is identical to group 2 for the M3A3 and the MARDER1A3. Also, notice that group 1 for the BTR-80 is almost identical to group 1 for the M3A3 and MARDER1A3, except that there is no "Assault" for the BTR-80, and the acronym "OPFOR" has been added to the beginning of each task name.

There are several issues with the task listing of Figure 10. First, the ordering has changed. Most of the tasks are identical to both the M3A3 and MARDER1A3, yet the order is changed. Group 2 appears first, then Group 1. Second, some of the task names have changed. Group 1 tasks have had the acronym "OPFOR" added. However, Group 2 names remain unchanged. Finally, the primary issue that prevented further automation of test cases is that after viewing the task listing for the BTR-80, when the user goes back to view the task listing of the M3A3 and MARDER1A3, their order was changed to be the same order as the BTR-80. This is shown in Figure 11. This happens only after viewing the task listing for the BTR-80. If the user goes back to view the task listing for the M3A3 or the MARDER1A3, they will see the listing shown in Figure 11, not the original listing shown in Figure 9. Because of this behavior, full automated testing was not possible since it is difficult, if not impossible, to predict when and how the task listing will change at run-time for all entities provided by OTB.

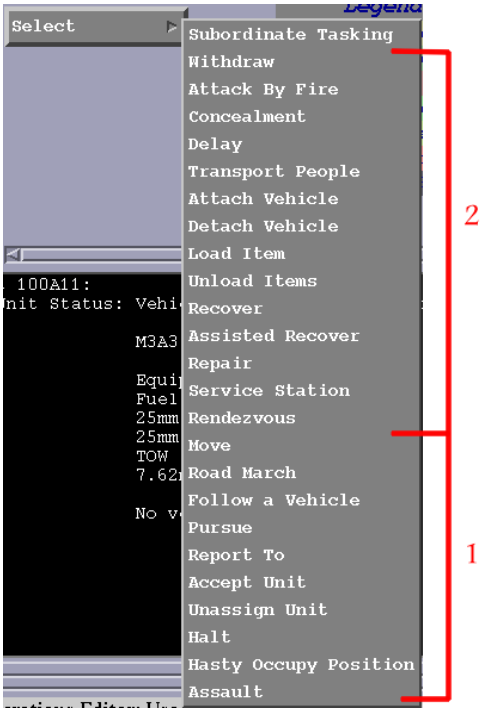

Figure 11. M3A3 (shown) and MARDER1A3 modified task listing

In addition to the problems just described, full automation is also hindered by the fact that there is no easy method for automatically selecting and assigning tasks to vehicles. For example, suppose an M1A1 entity was placed on the map. Once created, the only way to assign tasks to the entity is to select the entity on the map. This will cause the "Unit Operations" editor to appear. There is no other way to view this editor except to select the entity on the map. Unfortunately, there is no way of determining the screen coordinates of the entity on the map. Therefore, it requires a human to locate and select the entity.

OBSERVATIONS ABOUT TEST AUTOMATION

There are two primary phases to executing the test case scenarios with OTB. These phases are shown in Figure 12. As shown, there is a set-up and execution phase. During setup, OTB is started using the appropriate initialization data. This is the Start-up step. To use the test harness, the OTB interface may need to be resized in order for the harness to use the interface accurately.

Next, the actors to be used in the scenario are placed on the map and initialized with the appropriate data. This is the Force Laydown step. In the next step, the appropriate tasks are assigned to the actors. Finally, the scenario is saved to facilitate regression testing. Once the Scenario Set-up phase is complete, it is then executed by clicking "On Order" on the appropriate tasks. This is the Scenario Execution phase.

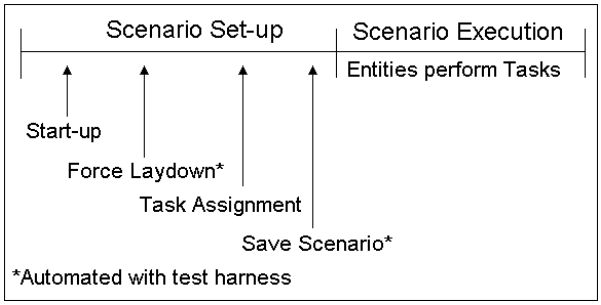

Figure 12. Two primary phases for executing test cases

For this application, there were 200 test cases to be executed. For test cases 1 – 125, all of the steps in the set-up phase were executed manually. For test cases 126 – 200, the start-up and task assignment steps were executed manually, while the force laydown and saving the scenario were executed using the test harness. Table 1 and Figure 13 show the time required to execute the test cases.

It is important to note that these test execution times are based on a single tester in a single test environment. The results would be expected to vary widely depending on the tester's experience with OTB, the tester's knowledge of and understanding of the test environment. In addition, since only 200 test cases were executed, it is possible that the apparent leveling off of the trends was a temporary phenomenon, rather than the end of a learning curve. Thus, these results should be considered to provide merely an illustration of benefits that can occur with test automation.

Table 1. Time Required for Test Sets

Test Cases	Time	Technique	Improved
1 – 25	6 hours	Manual	--
26 – 50	5 hours	Manual	120 %
51 – 75	4 hours	Manual	150 %
76 – 100	4 hours	Manual	150 %
101 – 125	4 hours	Manual	150 %
126 – 150	3 hours	Automated	200 %
151 – 175	3 hours	Automated	200 %
176 – 200	2 hours	Automated	300 %

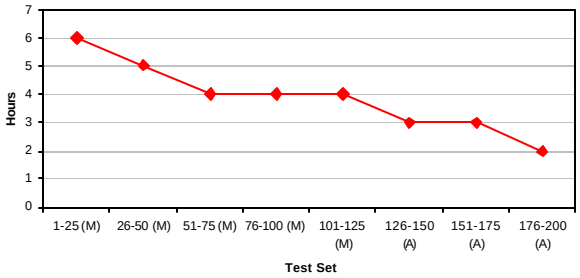

Figure 13. Total test time for each set of 25 test cases

As shown in Figure 13, there was an initial learning curve (between the first set of test cases and the third), resulting in more familiarity with the workflow of executing the test cases. By the third test set, the amount of time required to execute 25 test scenarios leveled off at 4 hours. (All the testing to this point was manual.)

Starting with test case 126, the test harness was used. There was an immediate improvement in the time (1 hour) and effort (reduced usage of the keyboard and mouse) required by the tester to execute the test cases. In addition, there was a second learning curve (between the seventh set of test cases and the eighth set), resulting in more familiarity with using the test harness as part of the testing process. Ultimately, the time required to execute 25 test cases was reduced to 2 hours. Overall, test execution required 31 hours for all 200 test cases. The times shown for each test set in Table 1 and Figure 13 represent the total time for setting up, execution, and evaluating all 25 test cases in the test set. Any reduction in time occurred during the Scenario Set-up phase (as shown in Figure 12). Further reduction in time and effort could have been achieved through automation, in this particular application, if not for the inconsistencies of the OTB GUI described in this paper.

SUGGESTIONS & CONCLUSIONS

In light of the difficulties described here with the OTB GUI and the potential benefits of automated testing at the system level, what can be done to improve the automated testing experience for this and other simulation systems? The ultimate answer to this question requires an additional perspective, an automated testing perspective, for GUI development. GUI development should primarily focus on the user. Often times, this is the only perspective taken. What may be easy or tolerable for a person is not so easy or

tolerable for a machine (and vice versa). As a result, GUIs have proven very challenging for automated testing.

GUI development should therefore also include an automated testing perspective that would look at the GUI from the machine's perspective. The challenge then becomes optimization of the GUI for both person and machine. The GUI should first be optimized for the user. However, there may be some leeway in the constraints formed by the user's needs, whereby the GUI can also be optimized for automated testing without negatively affecting the user's experience.

A good example of such a situation is shown in Figure 4 and Figure 5. In Figure 4, the selection box under the heading "Experimental:" is set to "Operational." From the user's perspective, this may be convenient and increase the user's productivity. This becomes a constraint under which automated testing must conform. However, suppose that changing the default settings of Figure 4 to be those shown in Figure 5 would not degrade the user's experience or, in any way, affect the user's productivity. This becomes an improvement that can be made to the GUI that would help optimize automated testing without negatively affecting the user (i.e., optimizing the GUI for automated testing within the constraints of the user's needs).

To optimize the GUI for both person and machine, some questions to be considered during GUI development include:

1. Is the GUI easy and intuitive for the user (user's perspective)?
2. What are the problems in the GUI that complicate the user's experience?
3. Can these problems be fixed without negatively affecting the user's experience?
4. Is the GUI easy to automate for testing purposes (machine's perspective)?
5. What are the problems in the GUI that complicate automated testing?
6. Can these problems be fixed without negatively affecting the user's experience? If not, do they simply become constraints for automated testing or do they impede automated testing such that it cannot be done?

Questions 1 – 3 are concerned with optimizing the user's experience. These questions are concerned with human-computer interfacing and have been and currently are being extensively investigated. Questions 4 – 6, however, are concerned with optimizing the automated testing experience, but within the constraints formed by the user's needs. Question 6 is perhaps the most difficult as it cannot be answered without first answering the other five. Questions 1 – 3 are intended to optimize the GUI for the user. Questions 4 and 5 are intended to optimize the GUI for the machine. Question 6 then tries to combine the solutions from both perspectives by giving priority to the user's perspective. Careful consideration of these questions during GUI development should ultimately result in a GUI that is both easy to use and facilitates test automation.

ACKNOWLEDGEMENTS

This work was funded in part by NAWC-TSD Contract N61339-01-D-002. The authors would like to thank Doug Parsons of the U.S. Army's PEO STRI for his support and insight that made this work possible.

REFERENCES

[1] Kaner, Cem. (1997). Pitfalls and strategies in automated testing. *IEEE Computer, Vol. 30, Issue 4*, pages 114 – 116.

[2] Memon, Atif M. (2002). GUI testing: pitfalls and process. *IEEE Computer, Vol. 35, Issue 8*, pages 87 – 88.

[3] Patton, Robert M. (2002). *Application of Intelligent Methods for Improved Testing and Evaluation of Military Simulation Software*. Ph.D. dissertation, University of Central Florida. Retrieved June 2, 2003, from http://rmpatton.net/dissertation

[4] U.S. Army's OneSAF Testbed Baseline System. Retrieved June 2, 2003, from http://www.onesaf.org/onesafotb.html

[5] Sun's Java Robot class. Retrieved June 2, 2003, from http://java.sun.com/j2se/1.4.1/docs/api/java/awt/Robot.html

[6] Usage Model Draw. Retrieved June 2, 2003, from http://rmpatton.net/projects/umd/