Bauersfeld, S.; Vos ., TE. (2012). GUITest: a Java library for fully automated GUI robustness testing. En ASE 2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. ACM. 330-333. doi:10.1145/2351676.2351739.

GUITest – A Java Library for fully automated GUI Robustness Testing.

(Tool Demonstrations)

Sebastian Bauersfeld
Universitat Politècnica de València
Camino de Vera s/n
Valencia, Spain
sbauersfeld@pros.upv.es

Tanja E. J. Vos
Universitat Politècnica de València
Camino de Vera s/n
Valencia, Spain
tvos@pros.upv.es

ABSTRACT

Graphical User Interfaces (GUIs) are substantial parts of today's applications, no matter whether these run on tablets, smartphones or desktop platforms. Since the GUI is often the only component that humans interact with, it demands for thorough testing to ensure an efficient and satisfactory user experience. Being the glue between almost all of an application's components, GUIs also lend themselves for system level testing. However, GUI testing is inherently difficult and often involves great manual labor, even with modern tools which promise automation. This paper introduces a Java library called GUITest[1], which allows to generate fully automated GUI robustness tests for complex applications, without the need to manually generate models or input sequences. We will explain how it operates and present first results on its applicability and effectivity during a test involving Microsoft Word.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Software Testing

Keywords

gui testing, automated testing, robustness testing

1. INTRODUCTION

Graphical User Interfaces (GUIs) represent the main connection point between a software's components and its end users and can be found in almost all modern applications. Vendors strive to build more intuitive and efficient interfaces to guarantee a better user experience, making them more powerful but at the same time much more complex. Especially since the rise of smartphones and tablets, this complexity has reached a new level and threatens the efficient testability of applications at the GUI level. To cope with this challenge, it is necessary to *automate* the testing process.

Capture and Replay (CR) tools promise to automate testing at the GUI level. However, they have always been controversial, since they require a great amount of manual effort on the part of the testers[6], who still need to record and maintain input sequences. Especially in the context of frequently changing GUIs, CR tools generate high maintenance costs. Despite its disadvantages, the CR method can be valuable, since the test cases are generated by humans with domain knowledge. However, the complexity of today's GUIs, the manual labor associated with testing them and the resulting maintenance costs, call for a complementary method with a truly automatic test environment. In this environment test case generation, execution and evaluation should be performed without human intervention. This is quite difficult, because GUIs are designed to be accessed by humans, not programs. A program needs to programmatically access the GUI's state in order to be able to simulate human-like behavior in the form of clicks, keystrokes and gestures. Therefore, we developed *GUITest*, a Java library which allows to write automated robustness tests for complex graphical applications. In the following we will explain how GUITest works, why and how it is different from existing tools and libraries and how it performed during a first test involving a complex, non-Java desktop application.

2. THE APPROACH

Figure 1 visualizes the basic procedure that a human goes through when using a graphical user interface. In picture (a) we can see a phone's application panel with the corresponding icons. Just by looking at the screen, most humans intuitively know which actions can be executed in this particular state. One could for example tap one of the five app items or swipe to the left or right in order to reveal additional ones. If the user clicks on the lower left item (b) a browser app will start (c), offering a variety of actions to choose from (d). After for example tapping the search field, a virtual keyboard will appear (e) and again he will have to make a decision on which action to execute. By repeating

[1]Videos and screenshots available at `https://staq.dsic.upv.es/sbauersfeld/index.html`

ASE'12, September 3–7, 2012, Essen, Germany.

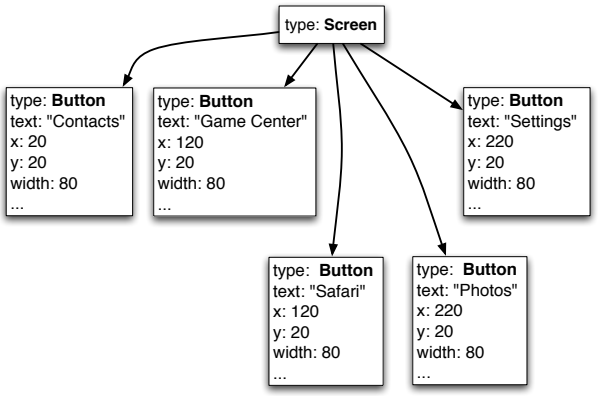

Figure 2: Widget Tree for picture a) of Figure 1.

this process, one can generate arbitrary input sequences and thus drive the GUI.

In order for a program to drive the GUI in a similar manner, it is necessary to gather enough information about the GUI's state, which constitutes the state of its widgets (i.e. control elements). To be able to perform sensible actions, it needs to determine the type, position, size and other attributes of all widgets visible in a certain state. GUITest can determine the current GUI state of the System Under Test (SUT) in the form of a *widget tree*. A widget tree is a hierarchical composition of the widgets currently visible on the screen, together with the values of associated widget attributes. Figure 2 shows an example of such a tree for picture (a) . With this information, an automatic testing program can compute sensible default actions: Enabled buttons, icons and hyperlinks can be tapped, text fields can be tapped and filled with text, the screen, scrollbars and sliders may be dragged, etc. GUITest allows to simulate simple (clicks, keystrokes) as well as complex actions (drag and drop operations, handwriting and other gestures, etc.) and can thus drive even sophisticated GUIs. Table 1 lists some of GUITest's features.

–	implemented in Java
–	currently supported OSs: MacOS 10.3 and newer
–	platform independent and extensible design
–	works with all native applications and applications that support the MacOSX Accessibility API
–	SUT requires no instrumentation
–	allows web testing through the *Safari* browser
–	derives sensible default actions for each GUI state
–	allows the definition of custom actions
–	generated sequences can be saved and replayed
–	allows the implementation of fine-grained oracles for fault detection

Table 1: GUITest Features

3. A FIRST TEST

To evaluate GUITest's functionality, we implemented a robustness test for *Microsoft Word for Mac*[2]. The goal was to develop a program which generates random input sequences and automatically detects crashes of the SUT. We considered the SUT to have crashed if a) it unexpectedly terminated or b) the GUI did not respond to inputs during more than 60 seconds. We leveraged GUITest's functionality to write a completely automated test which requires no supervision. Therefore we had to

- supply the name of Word's executable and the location of its configuration files: The configuration files need to be restored before each execution of Word in order to ensure identical startup conditions during sequence generation and playback. If these conditions differ (for example due to a previous run which changed values in the options dialog) one might not be able to replay crashing sequences.

- define the set of actions to execute: As mentioned earlier, GUITest already derives default actions for the majority of widgets. However, one might want to include additional actions, like dragging clip art symbols into the document (see Figure 3). It might also be necessary to disallow certain actions, for example: During our initial tests the program executed the "Shut Down" and "Restart" menu items in the system menu, which consequently terminated the entire machine. Other issues arose in the context of file dialogs: Within such dialogs the program can create, delete or move files, which might cause severe damage, depending on where in the file system it is browsing. Print dialogs and the like should also be treated carefully. Figure 3 shows typical actions which can be executed as well as certain widgets for which we intentionally disallowed actions (notice the *print*, *save* and *open* tool items as well as the *Apple* and *"Word"* menu items). Eventually, we ran the program in a dedicated user account with stricter access rights in order to prevent potential damage.

We limited the length of the generated sequences to 200 actions in order to be able to replay crash runs. After each run, the test program stopped the SUT, saved the sequence (if it caused a crash), restored the configuration files, restarted the SUT and went on to generate the next test case. Altogether, we wrote four Java class files with a total of 314 LOC. The automated test ran for 18 hours resulting in 672 generated sequences. Nine of these sequences caused Microsoft Word to crash and display an error message. Six of these we were able to reproduce. Three, however, were difficult to replay, as the timing during playback played an important role: During some runs a sequence crashed the SUT, yet during others it passed through without any problems. We believe that this is related to external factors (memory consumption, processor utilization, ...) which are hard to control. We are currently working to improve sequence playback and are considering the use of virtual machines which could give better control over those conditions. Another option would be to video record the entire testing process which would make sequence playback unnecessary.

4. OTHER TOOLS AND TECHNOLOGIES

There are various tools available for GUI testing, including commercial, open source and scientific products. A large part of them falls into the capture and replay (CR) or scripting categories. Popular commercial tools are *eggPlant*[3],

[2] `http://www.microsoft.com/mac/word`

[3] `http://www.testplant.com/products/eggplant/`

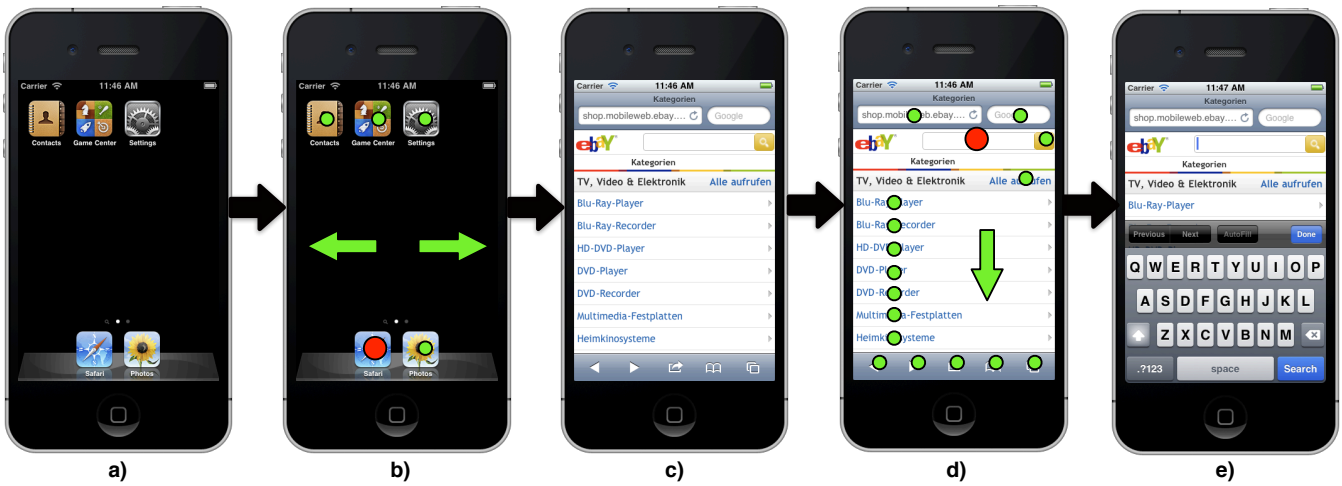

Figure 1: Sequence generation: Scan the GUI, derive sensible actions, select one and execute it.

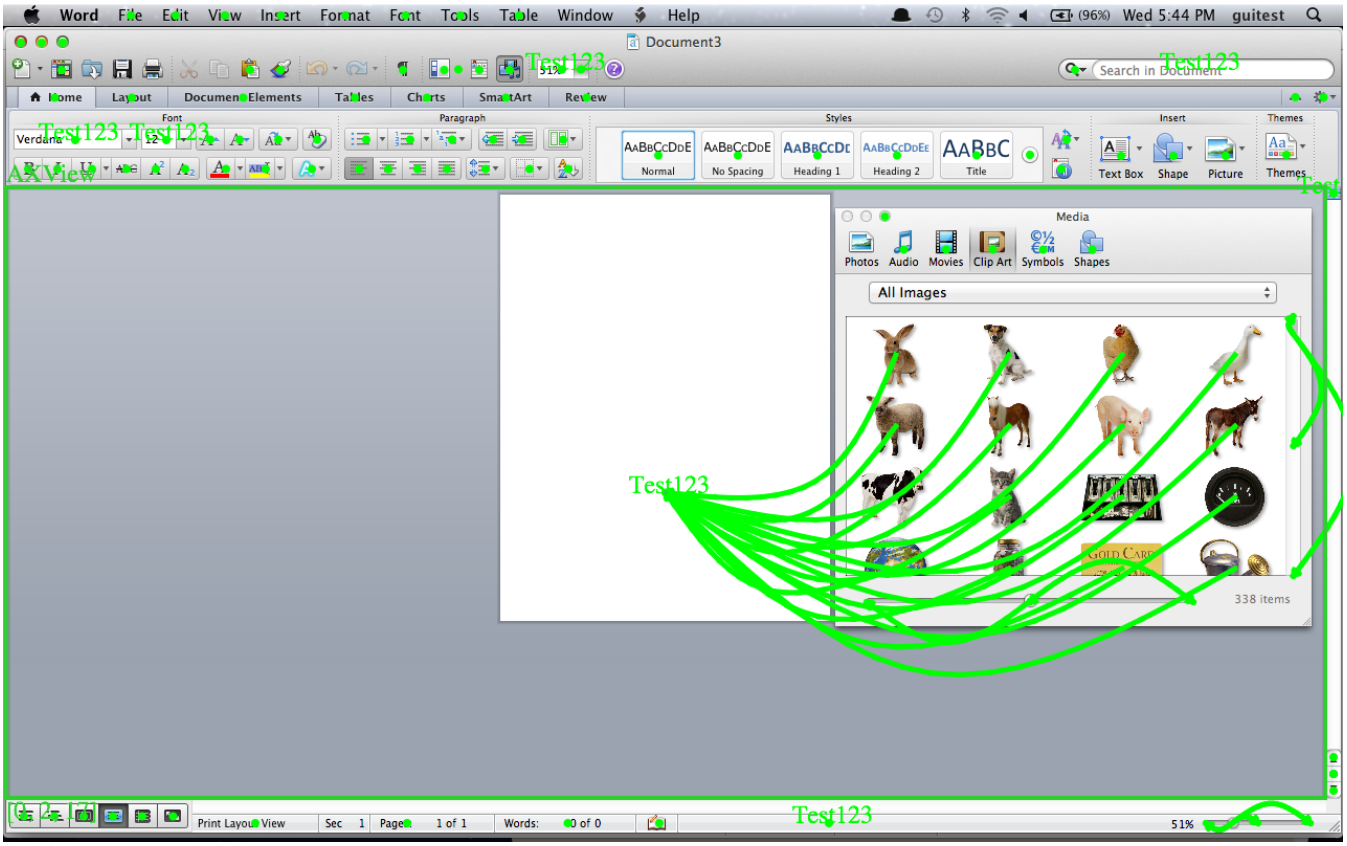

Figure 3: This is a screenshot of GUITest's demonstration mode, which displays possible actions within a particular GUI state. It includes clicks, text input ("Test123") and drag and drop operations.

TestComplete[4] or *QF-Test*[5]. Among the open source tools are *Abbot*[6], *Selenium*[7] and *SWTBot*[8]. As mentioned earlier, these tools often induce a lot of manual labor, especially when a GUI is subject to frequent changes so that recorded test cases break and need to be repaired. However, since the test cases are recorded by humans (potentially with good knowledge about the SUT), they can be very effective.

There have been a few scientific approaches to GUI testing which are more automatic than the CR method: An interesting one has been realized within the *GUITAR* framework[9] developed under the lead of Atif Memon. Their idea is to walk through the GUI (by systematically clicking on widgets) and to automatically generate a model (in the form of an event flow graph) from which they derive test cases by applying several coverage criteria. [4] gives an overview over their work. Unfortunately, in their experiments they only test small Java applications (some of them are synthetic, some are part of a small office suite developed by students) which they execute by performing clicks only. They have problems with the execution of their sequences, since the GUI model they are derived from is an approximation. Thus, they generate short sequences (3 to 20 actions) which they then automatically repair by applying a genetic algorithm.

Artzi et al.[1] perform feedback-directed random test case generation for JavaScript web applications. Their objectives are to find test suites with high code coverage as well as sequences that exhibit programming errors, like invalid-html or runtime exceptions. They developed a framework called *Artemis*, which triggers events by calling the appropriate handler methods and supplying them with the necessary arguments. To direct their search, they use prioritization functions: They select event handlers at random, but prefer the ones for which they have have achieved only low branch coverage during the past sequences.

Marchetto and Tonella [5] generate test suites for AJAX applications using metaheuristic algorithms. They execute the applications to obtain a finite state machine. The states in this machine are instances of the application's DOM-tree (Document Object Model) and the transitions are events (messages from the server / user input). From this FSM they calculate the set of *semantically interacting events*. The goal is to generate test suites with maximally diverse event interaction sequences, i.e. sequences where each pair of consecutive events is semantically interacting.

The strength of our approach is, that it works with large, native applications which it can drive using complex actions. The abovementioned approaches either invoke event handlers (which is not applicable to many GUI technologies) or perform only simple actions (clicks). Moreover, our technique neither involves manual labor (generation of input sequences or models) nor requires the SUT's source code for instrumentation. While this is also true for the approach presented in [4], they only generate short sequences of which many are invalid. Thus, these sequences need to be repaired, which, according to the authors, can take days or even weeks. Finally, the presented technique has very low maintenance costs, since the tests will continue to work even if the GUI changes. We are aware, however, that currently our approach is quite simple (random action selection) and only suitable for finding severe faults which cause crashes. It might benefit from ideas of the abovementioned approaches, as well as vice versa.

[4] `http://smartbear.com/products/qa-tools/automated-testing-tools`
[5] `http://www.qfs.de/en/qftest/index.html`
[6] `http://abbot.sourceforge.net`
[7] `http://seleniumhq.org/`
[8] `http://www.eclipse.org/swtbot`
[9] Download: `http://sourceforge.net/projects/guitar/`

5. CONCLUSIONS AND FUTURE WORK

In this paper we presented a robustness test for Microsoft Word, implemented with the help of our GUI testing library GUITest. Despite the simplicity of this test, the results are quite promising and show the applicability of this approach even for large scale GUI applications with complex interfaces. In the future we will focus on the following aspects:

1. More sophisticated action selection: We plan to apply machine learning techniques and metaheuristics to find sequences which are more likely to crash the SUT (e.g. sequences that consume a lot of memory or have long execution times). In earlier works [3, 2] we presented some ideas of how this could work.

2. Fine-grained oracles: We will strive to detect faults other than crashes. The focus is on automated techniques which learn to distinguish correct from erroneous behavior.

3. Other GUI technologies: Currently, GUITest only supports applications running under MacOSX. We have conducted feasibility studies for other platforms and in the future plan to support Windows. We also consider to implement the approach for a tablet or smartphone platform.

6. ACKNOWLEDGMENTS

This work is supported by EU grant ICT-257574 (FITTEST).

7. REFERENCES

[1] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A framework for automated testing of javascript web applications. In *ICSE'11*, 2011.

[2] S. Bauersfeld, S. Wappler, and J. Wegener. An approach to automatic input sequence generation for gui testing using ant colony optimization. In *GECCO'11*, 2011.

[3] S. Bauersfeld, S. Wappler, and J. Wegener. A metaheuristic approach to test sequence generation for applications with a gui. In *SSBSE'11*, 2011.

[4] S. Huang, M. B. Cohen, and A. M. Memon. Repairing gui test suites using a genetic algorithm. In *ICST'10*, 2010.

[5] A. Marchetto and P. Tonella. Using search-based algorithms for ajax event sequence generation during testing. *Empirical Softw. Engg.*, 2011.

[6] A. M. Memon. *A comprehensive framework for testing graphical user interfaces*. PhD thesis, University of Pittsburgh, 2001.

APPENDIX

A. PRESENTATION OUTLINE

Currently, we have a running test for Microsoft Word, which we will present during the conference. On our website (see below) we have several screenshots and videos which will be used for the presentation. During a live demonstration we will explain how the library obtains the GUI states and calculates possible actions and we will show what a test run looks like. Eventually, we will present the faults that we have found for Microsoft Word. By the time the conference will take place, we will also be able to present results from tests with other applications.

B. WEBSITE

On `https://staq.dsic.upv.es/sbauersfeld/index.html` you can download screenshots, videos and other files related to GUITest. We also have recordings of some of the crashes that occurred during our tests.

C. SCREENSHOTS

Figure 4: Our test caused Microsoft Word to crash or hang several times.

Figure 5: During our tests with Word the included equation editor also crashed.

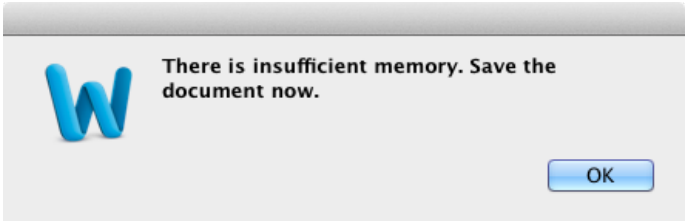

Figure 6: Sometimes Word ran out of memory.

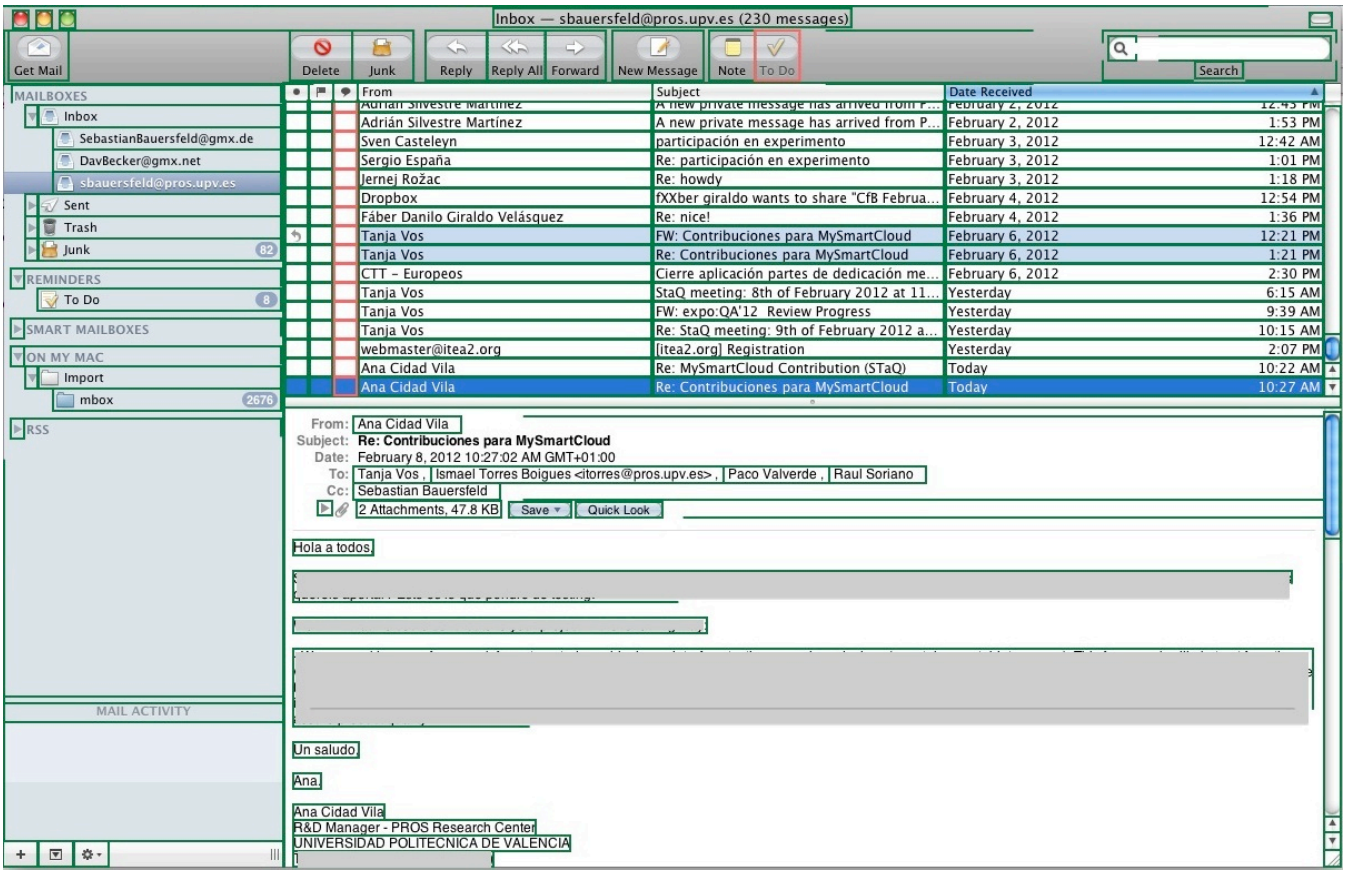

Figure 7: GUITest recognizes widgets of all native MacOSX applications, like the standard mail client.