A Reinforcement Learning Approach to Automated GUI Robustness Testing

Sebastian Bauersfeld and Tanja E. J. Vos

Universitat Politècnica de València,
Camino de Vera s/n, 46022, Valencia, Spain
{sbauersfeld,tvos}@pros.upv.es
http://www.upv.es

Abstract. Graphical User Interfaces (GUIs) can be found in almost all modern desktop, tablet and smartphone applications. Since they are the glue between an application's components, they lend themselves to system level testing. Unfortunately, although many tools promise automation, GUI testing is still an inherently difficult task and involves great manual labor. However, tests that aim at critical faults, like crashes and excessive response times, are completely automatable and can be very effective. These robustness tests often apply random algorithms to select the actions to be executed on the GUI. This paper proposes a new approach to fully automated robustness testing of complex GUI applications with the goal to enhance the fault finding capabilities. The approach uses a well-known machine learning algorithm called Q-Learning in order to combine the advantages of random and coverage-based testing. We will explain how it operates, how we plan to implement it and provide arguments for its usefulness.

Keywords: gui testing, automated testing, reinforcement learning

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. This makes them attractive for testers, since testing at the GUI level means testing from the user's perspective and is thus the ultimate way of verifying a program's correct behavior. However, current GUIs are large, complex and difficult to access programmatically, which poses great challenges for their testability. In an earlier work [1] we presented a framework called *GUITest* to cope with these challenges. GUITest allows to generate arbitrary input sequences even for large Systems Under Test (SUT). The basic procedure comprises three steps: 1) Obtain the GUI's state (i.e. the visible widgets and their properties like position, size, focus ...). 2) Derive a set of sensible actions (clicks, text input, mouse gestures, ...). 3) Select and execute an action. Figure 1 gives an example of how GUITest iteratively applies these three steps in order to drive Microsoft Word. The ultimate goal is to generate sequences that crash the GUI or make it unresponsive.

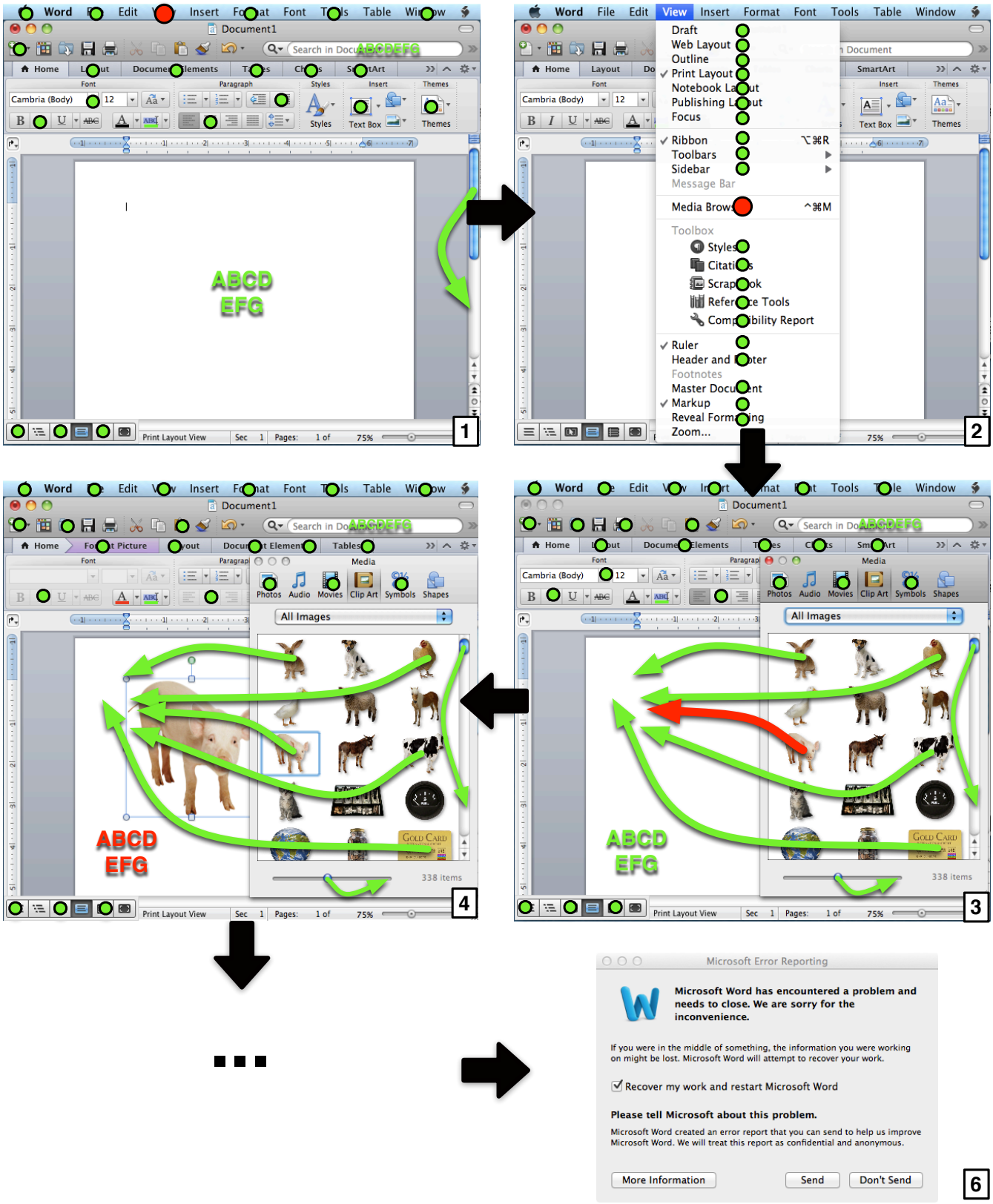

Fig. 1. Sequence generation by iteratively selecting from the set of currently available actions. The ultimate goal is to crash the SUT. (Not all possible actions are displayed in order to preserve clarity)

In its current form, GUITest selects the actions to be executed at random. We believe that this is a straightforward and effective technique of provoking crashes and reported on its success in [1]. We were able to find 14 crash sequences for Microsoft Word while running GUITest during 48 hours[1]. Some of the advantages and disadvantages of random action selection are:

+ No model or instrumentation required: This is an important aspect, since usually there exists no abstract model of the GUI. Generating one often involves manual effort and, due to size constraints, it is either not complete or not entirely accurate [2].
+ Easy to maintain: Tests continue to run in the presence of GUI changes.
+/− Unbiased: Each permutation of actions is equally likely to be generated. In the absence of domain knowledge about the SUT this is desirable. However, quite often certain aspects require more thorough testing than others.
− Not every action is equally likely to be executed.
− It can take a considerable amount of time until a sufficiently large percentage of the GUI has been operated.

In this paper we strive to improve on the last three characteristics. In large SUTs with many – potentially deeply nested – dialogs and actions, it is unlikely that a random algorithm will sufficiently exercise most parts of the GUI within a reasonable amount of time. Certain actions are easier to access and will therefore be executed more often, while others might not be executed at all.

The idea is to slightly change the probability distribution over the sequence space. This means that action selection will still be random, but seldom executed actions will be selected with a higher likelihood, in order to favor exploration of the GUI. The straightforward greedy approach of selecting at each state the action which has been executed the least number of times, might not yield the expected results: In a GUI it is often necessary to first execute certain actions in order to reach others. Hence, these need to be executed more often, which requires an algorithm that can "look ahead". This brought us to consider a reinforcement learning technique called Q-Learning. In the following section we will explain how it works, define the environment that it operates in and specify the reward that it is supposed to maximize.

2 The Algorithm

We assume that our SUT can be modeled as a finite *Markov Decision Process* (MDP). A finite MDP is a discrete time stochastic control process in an environment with a finite set of states S and a finite set of actions A [3]. During each time step t the environment remains in a state $s = s_t$ and a decision maker, called *agent*, executes an action $a = a_t \in A_s \subseteq A$ from the set of available actions, which causes a transition to state $s' = s_{t+1}$. In our case, A will refer

[1] Videos of these crashes are available at `https://staq.dsic.upv.es/sbauersfeld/index.html`

to the set of possible GUI actions, i.e. clicks on widgets, text input and mouse gestures and S will be the set of observable GUI states.

The state transition probabilities are governed by

$$P(a, s, s') = Pr\{s_{t+1} = s' | s_t = s, a_t = a\}$$

meaning, that the likelihood of arriving in state s' exclusively depends on a and s and not on any previous actions or states. This property, which is crucial to the application of reinforcement learning algorithms, is called *Markov Property*. We assume that it approximately holds for the SUT[2].

In an MDP, the agent receives rewards $R(a, s, s')$ after each transition, so that it can learn to distinguish good from bad decisions. Since we want to favor exploration, we set the rewards as follows:

$$R(a, s, s') := \begin{cases} r_{init} & \text{, if } x_a = 0 \\ \frac{1}{x_a} & \text{, else} \end{cases}$$

Where x_a is the amount of times that action a has been executed and r_{init} is a large positive number. Hence, the more often an action has been executed, the less desirable it will be for the agent. The ultimate goal is to learn a policy π which maximizes the agent's expected reward. The policy determines for each state $s \in S$ which action $a \in A_s$ should be executed. We will apply the Q-Learning algorithm in order to find π. Instead of computing it directly, Q-Learning first calculates a *value function* $V(s, a)$ which assigns a numeric quality value – the expected future reward – to each state action pair $(s, a) \in S \times A$. This function is essential, since it allows the agent to look ahead when making decisions. Eventually, it becomes trivial to derive the optimal policy: One selects $a^* = argmax_a\{V(s, a) | a \in A_s\}$ for each $s \in S$.

Algorithm 1 shows the pseudocode for our approach. The agent starts off completely uninformed, i.e. without any knowledge about the GUI. Step by step it discovers states and actions and learns the value function through the rewards it obtains. The quality value for each new state action pair is initialized to r_{init}. The heart of the algorithm is the value update in line 9: The updated quality value of the executed state action pair is the sum of the received reward plus the maximum value of all subsequent state action pairs multiplied by the discount factor γ. The more γ approaches zero, the more opportunistic and greedy the agent becomes (it considers only immediate rewards). When γ approaches 1 it will opt for long term reward. It is worth mentioning that the value function and consequently the policy will constantly vary, due to the fact that the rewards change. This is what we want, since the agent should always put emphasis on the regions of the SUT which it has visited the least amount of times.

Instead of always selecting the best action (line 6), one might also consider a random selection proportional to the values of the available state action pairs. This would introduce more randomness in the sequence generation process.

[2] Since we can only observe the GUI states and not the SUT's true internal states (*Hidden Markov Model*), one might argue whether the Markov Property holds sufficiently. However, we assume that this has no significant impact on the learning process.

Input: r_{init} /* reward for unexecuted actions */
Input: $0 < \gamma < 1$ /* discount factor */
begin
  start SUT
  $V(s, a) \leftarrow r_{init} \quad \forall (s, a) \in S \times A$
  repeat
    obtain current state s and available actions A_s
    $a^* \leftarrow argmax_a\{V(s, a) | a \in A_s\}$ /* select the best action */
    execute a^*
    obtain state s' and available actions $A_{s'}$
    $V(s, a^*) \leftarrow R(a^*, s, s') + \gamma \cdot max_{a \in A_{s'}} V(s', a)$ /* value update */
  until *stopping criteria met*
  stop SUT
end

Algorithm 1: Sequence generation with Q-Learning

Representation of States and Actions In order to be able to apply the above algorithm, we have to assign a unique and stable identifier to each state and action, so that we are able to recognize them, even across different runs of the SUT. Our testing library GUITest allows us to derive the complete GUI state of the SUT, so that we can inspect the property values of each visible widget. For example: It gives access to the title of a button, its position and size and tells us whether it is enabled. This gives us the means to create a unique identifier for a click on that button: It can be represented as a combination of the button's property values, like its title, help text or its position in the widget hierarchy (which parents / children does the button have). The properties selected for the identifier should be relatively "stable": The title of a window, for example, is quite often not a stable value (opening new documents in Word will change the title of the main window) whereas its tool tip or help text is less likely to change. The same approach can be applied to represent GUI states: One simply combines the values of all stable properties of all widgets on the screen. Since this might be a lot of information, we will only save a hash value generated from these values[3]. This way we can assign a unique and stable number to each state and action.

3 Related Work

Artzi et al. [4] 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

[3] Of course this could lead to collisions. However, for the sake of simplicity we assume that this is unlikely and does not significantly affect the optimization process.

necessary arguments. To direct their search, they use prioritization functions: They select event handlers at random, but prefer the ones for which they have achieved only low branch coverage during previous sequences.

Marchetto and Tonella [5] generate test suites for AJAX applications using metaheuristic algorithms. They execute the applications to obtain a finite state machine, whose states are instances of the application's DOM-tree (Document Object Model) and whose 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, 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 does not modify nor require the SUT's source code, which makes it applicable to a wide range of programs.

4 Future Work

In this paper we proposed a technique for improving the effectivity of GUI robustness testing. We believe in the ability of random testing to provoke crashes in GUI-based applications. However, due to the fact that modern GUIs often have deeply nested dialogs and actions, which are unlikely to be executed by a random algorithm, we think that a more explorative approach could improve the fault finding effectivity. We are therefore integrating an algorithm known as Q-Learning into our test framework GUITest. We think that this algorithm, which was designed to solve problems in the presence of uncertainty and lack of environmental knowledge, is suitable for testing large and complex GUIs without additional information provided by formal models. We expect to finish the implementation soon and look forward to report on the results.

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

References

1. Bauersfeld, S., Vos, T.E.J.: GUITest: A Java Library for Fully Automated GUI Robustness Testing. To appear in ASE 2012.
2. Huang, S., Cohen, M.B., Memon, A.M.: Repairing GUI Test Suites using a Genetic Algorithm. ICST 2010: pp. 245-254
3. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA, 1998.
4. Artzi, S., Dolby, J., Jensen, S.H., Møller, A., Tip, F.: A Framework for Automated Testing of Javascript Web Applications. ICSE 2011: pp. 571-580
5. Marchetto, A., Tonella, P.: Using Search-Based Algorithms for Ajax Event Sequence Generation during Testing. Empirical Software Engineering Volume 16 Issue 1, pp. 103-140. (Kluwer Academic Publishers Hingham, MA, USA). 2011