User interface level testing with TESTAR; what about more sophisticated action specification and selection?

Sebastian Bauersfeld and Tanja E. J. Vos

Research Center on Software Production Methods (PROS)
Universitat Politècnica de València
Valencia, Spain

Abstract. Testing software applications at the Graphical User Interface (GUI) level is a very important testing phase to ensure realistic tests because the GUI represents a central juncture in the application under test from where all the functionality is accessed. In earlier works we presented the TESTAR tool, a Model-Based approach to automate testing of applications at the GUI level whose objective is to generate test cases based on a model that is automatically derived from the GUI through the accessibility API. Once the model has been created, TESTAR derives the sets of visible and unblocked actions that are possible for each state that the GUI is in and randomly selects and executes actions in order to drive the tests. This paper, instead of random selection, we propose a more advanced action specification and selection mechanism developed on top of our test framework *TESTAR*. Instead of selecting random clicks and keystrokes that are visible and unblocked in a certain state, the tool uses a Prolog specification to derive sensible and sophisticated actions. In addition, it employs a well-known machine learning algorithm, called Q-Learning, in order to systematically explore even large and complex GUIs. This paper explains how it operates and present the results of experiments with a set of popular desktop applications.

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 behaviour. Current GUIs can account for 45-60% of the entire source code [14] and are often large and complex. Consequently, it is difficult to test applications thoroughly through their GUI, especially because GUIs are designed to be operated by humans, not machines. Moreover, they are inherently non-static interfaces, subject to constant change caused by functionality updates, usability enhancements, changing requirements or altered contexts. This makes it very hard to develop and maintain test cases without resorting to time-consuming and expensive manual testing.

In previous work, we have presented TESTAR [5,6,3], a Model-Based approach to automate testing at the GUI level. TESTAR uses the operating system's Accessibility API to recognize GUI controls and their properties and enables programmatic interaction with them. It derives sets of possible actions for each state that the GUI is in (i.e. the visible widgets, their size, location and other properties such as whether they are enabled or blocked by other windows etc.) and randomly selects and executes appropriate ones in order to drive the tests. In completely autonomous and unattended mode, the oracles can detect faulty behaviour when a system crashes or freezes. Besides these free oracles, the tester can easily specify some regular expressions that can detect patterns of suspicious titles in widgets that might pop up during the executed tests sequences. For more sophisticated and powerful oracles, the tester can program the Java protocol that is used to evaluate the outcomes of the tests.

The strength of the approach is that the technique does not modify nor require the SUT's source code, which makes it applicable to a wide range of programs. With a proper setup and a powerful oracle, TESTAR can operate completely unattended, which saves human effort and consequently testing costs. We believe that TESTAR is a straightforward and effective technique of provoking crashes and reported on its success describing experiments done with MS Word in [5]. We were able to find 14 crash sequences while running TESTAR during 48 hours[1] applying a strategy of random selection of visible/unblocked actions.

In this paper we will investigate more sophisticated ways of action specification and selection. Instead of clicking randomly on visible and unblocked locations within the GUI, we enable the tester to define the set of visible actions that he or she wants to execute. The definitions are written in Prolog syntax and allow the specification of thousands of actions – even complex mouse gestures – with only a few lines of code. Moreover, we will use a machine learning algorithm called Q-Learning [4] to explore the GUI in a more systematic manner than random testing. It learns about the interface and strives to find previously unexecuted actions in order to operate even deeply nested dialogs, which a random algorithm is unlikely to discover. In the next section we will explain how TESTAR obtains the GUI state and executes actions.

This paper is structured as follows. Section 2 presents the TESTAR approach for testing at the GUI level and describes the extensions for action specification and selection. Section 3 presents the results of a first experiment in which we applied TESTAR to a set of popular applications to test its ability in finding reproducible faults. Section 4 lists related work, section 5 reviews the approach and section 6 describes future work.

2 The TESTAR Approach

Figure 1 shows how TESTAR would operate on MS Word. At each step of a sequence, TESTAR (A) determines the state of the GUI, i.e. the visible widgets,

[1] Videos of these crashes are available at `http://www.youtube.com/watch?v=PBs9jF_pLCs`

their size, location and other properties (such as whether they are enabled or blocked by other windows etc.). From that state it (B) derives a *set of feasible actions* (the green dots, letters and arrows, which represent clicks, text input and drag and drop operations, respectively) from which it then (C) selects one (marked red) and finally executes it. By repeating these steps, TESTAR will be able to generate arbitrary input sequences to drive the GUI. The following subsections will explain this in more details, together with the new more sophisticated ways of deriving actions and selection them for execution.

2.1 Determine the GUI State

All of our experiments described in this paper are conducted on MacOSX. For this platform the Accessibility API – which simplifies computer usage for people with disabilities – is used to obtain the SUT's GUI state. It allows to gather information about the visible widgets of an application and gives TESTAR the means to query their property values. Since it is a native API written in *ObjectiveC*, we make use of the *Java Native Interface* (JNI) to invoke its methods. After querying the application's GUI state, we save the obtained information in a so-called *widget tree* which captures the structure of the GUI. Figure 2 displays an example of such a tree. Each node corresponds to a visible widget and contains information about its type, position, size, title and indicates whether it is enabled, etc. The Accessibility API gives access to over 160 attributes which allows us to retrieve detailed information such as:

- The **type** of a widget.
- The **position** and **size** which describe a widget's rectangle (necessary for clicks and other mouse gestures).
- It tells us whether a widget is **enabled** (It does not make sense to click disabled widgets).
- Whether a widget is **blocked**. This property is not provided by the API but we calculate it. For example, if a message box blocks all other widgets behind it, then TESTAR can detect those and other modal dialogs (like menus) and sets the blocked attribute accordingly.
- Whether a widget is **focused** (has keyboard focus) so that TESTAR knows when it can type into text fields.
- Attributes such as **title, help** and other descriptive attributes are very important to distinguish widgets from each other and give them an identity. We will make use of this in the next subsection when we describe our algorithm for action selection.

This gives us access to almost all widgets of an application, if they are not custom-coded or drawn onto the window. We found that the Accessibility API works very well with the majority of native applications (since the API works for all standard widgets, the developers do not have to explicitly code for it to work).

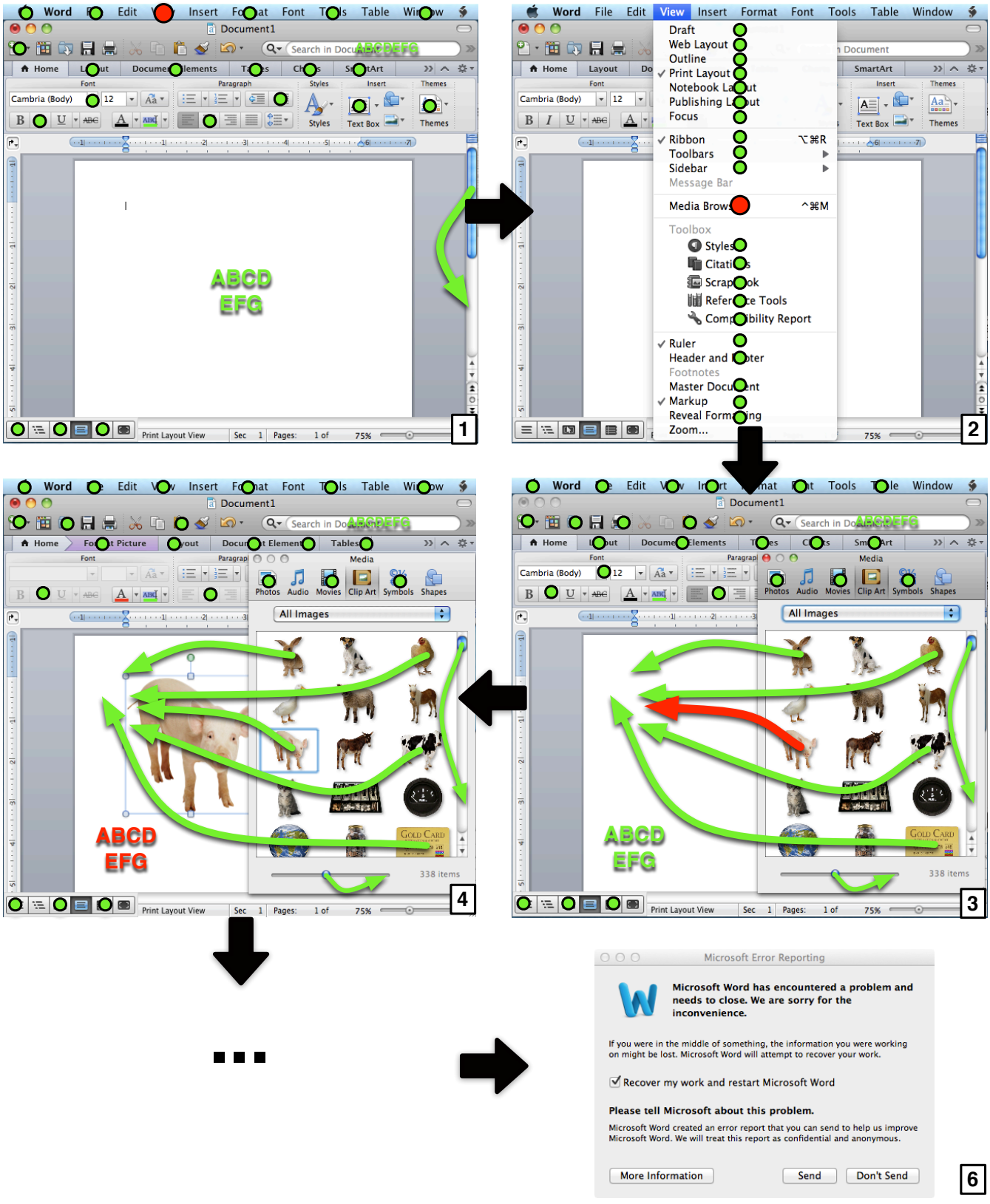


Fig. 1. Sequence generation by iteratively selecting from the set of currently available actions. The ultimate goal is to crash the SUT. (In order to preserve clarity the graphic does not display all possible actions.)

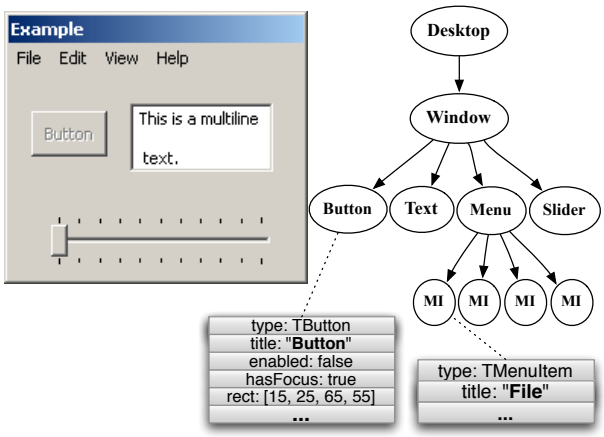


Fig. 2. The state of a GUI can be described as a widget tree which captures property values for each control.

2.2 Derive Actions

Having obtained the GUI's current state, we can go on to derive a set of actions. One thing to keep in mind is that the more actions TESTAR can choose from, the bigger the sequence space will be and the more time the search for faults will consume. Ideally, TESTAR should only select from a small set of actions which are likely to expose faults. Thus, our intention is to keep the search space as small as possible and as large as necessary for finding faults. For each state we strive to generate a set of *sensible* actions which should be appropriate to the widgets that they are executed on: Buttons should be clicked, scrollbars should be dragged and text boxes should be filled with text. Furthermore, we would like to exercise only those widgets which are *enabled* and not *blocked*. For example: it would not make sense to click on any widget that is blocked by a message box. Since the box blocks the input, it is unlikely that any event handling code (with potential faults) will be invoked.

One way to tell TESTAR how to use specific widgets would be to provide a set of rules so that it can generate actions according to a widget's type, position and other attribute values. This would work reasonably well for many GUIs, since most widgets are standard controls. However, it is not very flexible. There might be non-standard widgets which TESTAR does not know how to use or which are used in a different way than they were designed for (Microsoft Word uses static text labels as table items and has lots of custom coded widgets). Moreover, on screen 3 of Figure 1 how would TESTAR know that it can drag images from the media browser into the document or that it can draw onto the canvas in Figure 4? Finally, a tester might want to execute only specific parts of the application, e.g. click only buttons and menu items or leave out certain actions which could trigger "hazardous" operations that delete or move files.

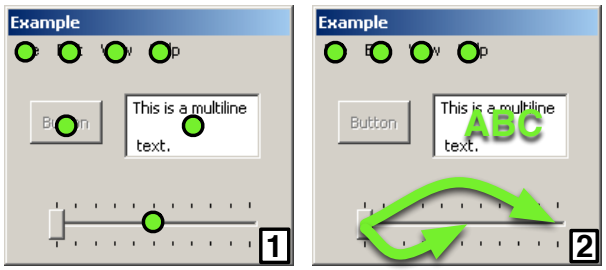
Due to these reasons, we let the tester specify which actions TESTAR should execute. Since modern GUIs can be very large and complex, with potentially ten thousands of widgets, the tester needs a comfortable and efficient way of defining actions. We found that action specification in Java is often verbose and not very concise. Definitions such as "Drag every image within the window titled 'Images' onto the canvas with help text 'Document'." often require many lines of code. Therefore, we were looking for a more declarative language, which allows the tester to specify actions in a more natural way. Eventually, we decided to integrate a *Prolog* engine into our framework. Prolog is a programming language that has its roots in first-order-logic and is often associated with the artificial intelligence and computational linguistics communities [9]. In Prolog the programmer writes a database with facts and rules such as

```
parent(bruce, sarah).
parent(sarah, tom).
parent(gerard, tom).
ancestor(X,Y):- parent(X,Y);
                (parent(X,Z), ancestor(Z,Y)).
```

where the first three lines state that Bruce is Sarah's parent and that Sarah and Gerard are Tom's parents. The third line is a recursive rule and reads: "X is Y's ancestor, if X is the parent of Y or there exists a Z so that X is a parent of Z and Z is an ancestor of Y" (the semicolon and comma represent disjunction and conjunction, respectively). The programmer can now issue queries such as `?- ancestor(X,tom)` ("Who are Tom's ancestors?") against this database. Prolog will then apply the facts and rules to find the answers to this question, i.e. all possible substitutions for `X` (Sarah, Gerard, Bruce). This can be applied to much more complex facts and hierarchies and Prolog is known to be an efficient and concise language for those kind of relationship problems [9].

In our case we reason over widgets that also form a hierarchy. However, we do not have an explicitly written fact database. Instead, the widget tree acts as this database, as it describes the relationships among the widgets and contains their property values. The tester can then use the Prolog engine to define actions for the current GUI state. Figure 3.1 shows an example of how this is done. Let us assume we want to generate clicks for all widgets within our example dialog. The corresponding Prolog query is listed under the image and reads: "For all widgets W, which have an ancestor A, whose title is 'Example', calculate the center coordinates X and Y and issue a left click". Since this also generates clicks for the disabled button widget, the text box and the slider, we might want to improve this code, in order to obtain a more appropriate action set: Figure 3.2 shows the adapted Prolog code and its effects. We implemented predicates such as `widget(W)`, `enabled(W)`, `type(W, R)` and many others to allow the tester to reason over the widget tree.

Besides traditional clicks, text typing and drag and drop operations, the Prolog engine also facilitates the definition of more complex mouse gestures such as the ones depicted in Figure 4. The tester simply defines a set of points that the cursor is supposed to pass through and TESTAR then uses cubic splines to




```
1) widget(W),ancestor(A,W),title(A,"Example"),center(W,X,Y),lclick(X,Y).

2) widget(W),ancestor(A,W),title(A,"Example"),enabled(W),type(W,T),(
     ((T='Button'; T='MenuItem'), center(W, X, Y), lclick(X, Y));
     (T='Text', center(W, X, Y), click(X, Y), type('ABC'));
     (T='Slider', center(W, 'Thumb', X1, Y1),
        rect(W, Left, Top, Width, Height), Y2 is Top + Height / 2,
        (X2 is Left + Width / 2; X2 is Left + Width), drag(X1,Y1,X2,Y2))
   ).
```

Fig. 3. Action specification with Prolog queries.

calculate the cursor's trajectory. This allows for complex drawing gestures and handwriting.

The ability to define the set of actions makes it possible to customize TESTAR test and to restrict the search space to the actions that the tester thinks are likely to trigger faults. As our experiments will show, a reasonably sized set of "interesting" actions, can be more effective in finding faults than unconstrained mouse and keyboard inputs. The Prolog engine and the fact that GUIs usually consist of a limited set of widget types with similar functionality, allow the tester to specify thousands of these actions with only a few lines of code. Thus, even large and complex applications can be configured quickly.

For our Prolog engine we used a modified version of the *PrologCafe*[2] implementation, which compiles and transforms Prolog into Java code.

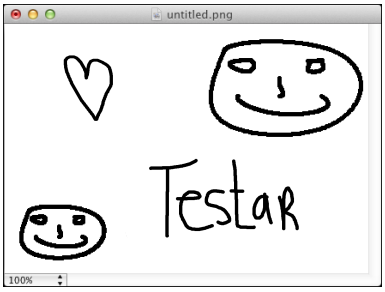


Fig. 4. TESTAR is capable of executing complex mouse gestures.

[2] `http://code.google.com/p/prolog-cafe`

2.3 Action Selection

The action selection strategy is a crucial feature within TESTAR. The right actions can improve the likelihood and decrease the time necessary for triggering crashes. Consequently, we want it to learn about the GUI and be able to explore it thoroughly. 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.

Ideally, TESTAR should exercise all parts of the GUI equally and execute each possible action at least once. Instead of pursuing a systematic exploration like Memon et al. [13], which use a depth first search approach, we would like to have a mixture between a random and systematic approach that still allows each possible sequence to be generated. Thus, our idea is to change the probability distribution over the sequence space, so that seldom executed actions will be selected with a higher likelihood than others, in order to favor exploration of the GUI. To achieve this, a straightforward greedy approach would be to select at each state the action which has been executed the least number of times. Unfortunately, this 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*. We will now explain how it works, define the environment that it operates in and specify the reward that it tries to maximize.

We assume that our SUT can be modelled 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 [16]. 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, the agent is TESTAR, the set A will refer to the possible GUI actions and S will be the set of observable GUI states.

An MDP's state transition probabilities are governed by

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

meaning, that the likelihood of arriving in state s' exclusively depends on a and s and not on any previous actions or states. This is called *Markov Property* which we assume holds approximately for the SUT[3]. In an MDP, the agent receives rewards $R(s, a, s')$ 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

[3] 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 the algorithm will still perform reasonably well.

follows:

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

Where x_a is the amount of times that action a has been executed and r_{init} is a large positive number (we want actions that have never been executed before, to be extraordinarily attractive). 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: The agent then simply selects $a^* = argmax_a\{V(s,a)|a \in A_s\}$ within each state $s \in S$.

Algorithm 1 shows the pseudo-code for our approach. Since it does not have any prior knowledge about the GUI, the agent starts off completely uninformed. 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[4].

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. For our experiments, however, we sticked to the presented version of the algorithm.

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. For this, we can use the structure and the property values within the widget tree. For example: To create a unique identifier for a click on a button we can use a combination of the button's property values, such as its title, help text or its position in the widget hierarchy (parents / children / siblings). The properties selected for the identifier should be

[4] Again, this is contrary to the Markov Property where rewards are supposed to be stationary. We counter the problem of constantly changing rewards with frequent value updates, i.e. we sweep more often over the state actions pairs within the generated sequence (i.e. line 9 of the algorithm).

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
    $a^* \leftarrow argmax_a\{V(s,a)|a \in A_s\}$
    execute a^*
    obtain state s' and available actions $A_{s'}$
    $V(s,a^*) \leftarrow R(s,a^*,s') + \gamma \cdot max_{a \in A_{s'}} V(s',a)$
  until *stopping criteria met*
  stop SUT
end

Algorithm 1: Sequence generation with Q-Learning [4]

relatively "stable": The title of a window, for example, is quite often not a stable value (opening new documents in a text editor will change the title of the main window) whereas its 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[5]. This way we can assign a unique and stable number to each state and action.

3 Evaluation

We performed two different experiments. In the first one, we compared three different approaches with TESTAR summarized in Table 1 to find out: ***RQ1*** - *which of these testing approaches needs the least amount of time to crash applications?*.

In the second experiment we used only approach C. At first we had it generate a set of short sequences – with at most 200 actions – with the goal to trigger crashes for each SUT. We then tried to replay these crashing sequences to determine the reproducibility of the revealed faults and find out: ***RQ2****: What fraction of the generated crash sequences are reproducible, i.e. trigger the crash again upon replay?*

3.1 The variables: What is being measured?

In the first experiment we measured the average time it took each approach to crash an application, which is equivalent to what Liu et al. [12] do in their experiments. We considered an SUT to have crashed if a) it unexpectedly terminated or b) it did not respond to inputs during more than 60 seconds.

[5] 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.

Table 1. The approaches of TESTAR that were compared.

Approach	Description
A	TESTAR's default configuration which randomly selects actions, but is informed in the sense that it uses the Accessibility API to recognize the visible/unblocked actions.
B	TESTAR with random selection of actions specified by the tester using Prolog as described in Section 2.2. This approach also randomly selects actions, but it is "informed", in the sense that it makes use of Prolog action specifications tailored to each SUT.
C	This approach is equivalent to B, but uses the Q-learning algorithm instead of random selection.

In the second experiment we measured the absolute number of crashes that we found and the percentage of the crashing sequences that we were able to reproduce.

3.2 The Systems Under Test (SUTs)

We applied the three approaches to a set of popular MacOSX applications as listed in the first column of Table 2. We tried to include different types of SUTs such as office applications (Word, Excel, Mail, iCal ...) an instant messenger (Skype), a music player (iTunes) and a drawing application (Paintbrush), to find out whether our approach is generally applicable.

3.3 The protocol

We carried out all of our experiments on a 2.7 GHz Intel Core i7 MacBook with 8GB RAM and MacOSX 10.7.4. To verify and analyze the found crashes, we used a *frame grabber* which recorded the screen content of our test machine during the entire process. Thus, many of the found crashing sequences can be viewed on our web page[6]. Before running any tests, we prepared each application by performing the following tasks:

– Write scripts which restore configuration files and application data before each run: This is a crucial step, since we compared different approaches against each other and wanted to make sure that each one starts the application from the same configuration. Most applications save their settings in configuration files, which needed to be restored to their defaults after each run. In addition, applications like *Mail* or *Skype* employ user profile

[6] http://www.pros.upv.es/testar

Table 2. Competition between approaches A, B and C..

Application	**Avg. time T (minutes) to crash (3 runs per SUT and approach)**		
	A	**B**	**C**
Excel 2011 v14.1.4	20.95	9.24	15.06
iTunes v10.6.1	1072.55*	53.86	49.23
PowerPoint 2011 v14.1.4	21.18	9.77	8.61
Skype v5.8.0.945	1440*	144.26	130.79
iCal v5.0.3	1099.81*	103.82	146.53
Calculator v10.7.1	62.93	18.75	19.93
Word 2011 v14.1.4	50.33	14.41	12.56
Mail v5.2	1276.71*	134.02	122.31
Paintbrush v2.0.0	1440*	239.82	234.04
Overall Average	720.5	80.89	82.12

data such as mailboxes or contact lists. During a run, mails and contacts might be deleted, added, renamed or moved. Thus, each approach had a list of directories and files which they automatically restored upon application start.

- Setup a secure execution environment: GUI testing should be done with caution. Unless they are told otherwise, the test tools will execute every possible action. Thus, they might print out documents, add / rename / move / delete files, printers and fonts, install new software or even shut down the entire machine. We experienced all of these situations and first tried to counter them by disallowing actions for items such as "Print", "Open", "Save As", "Restart", etc. However, applications such as Microsoft Word are large and allow many potentially hazardous operations which are difficult to anticipate. Therefore, we decided to run our tests in a sandbox (a MacOSX program called *sandbox-exec*) with a fine-grained access control model. This allowed us to restrict read and write access to certain directories, block specific system calls and restrict internet access. The latter was necessary when we were testing *Skype* and *Mail*. Since we employed some of our own contact lists and mail accounts, we could have contacted people and might have transmitted private data.
- For approaches B and C, we moreover defined a set of *sensible* actions: As described in Section 2.A and 2.B, we took care to specify actions appropriate to the specific widgets: We generated clicks on buttons and menu items as well as right-clicks, input for text boxes, drag operations for scrollbars, sliders and other draggable items. For one of the tested applications (*Paintbrush*, a Microsoft Paint clone) we also generated mouse gestures to draw figures as shown in Figure 4. For each application we strived to define a rich set of actions, comprising the ones that a human user would apply when working

with the program. The LOCs of the Prolog specifications can be found in Table 3.

Table 3. Size of Prolog action specifications in Lines Of Code (LOC).

Application	**LOC**
Excel 2011 v14.1.4	29
iTunes v10.6.1	25
PowerPoint 2011 v14.1.4	29
Skype v5.8.0.945	30
iCal v5.0.3	28
Calculator v10.7.1	11
Word 2011 v14.1.4	29
Mail v5.2	30
Paintbrush v2.0.0	39

For each approach we set the delay between two consecutive actions to 50 ms, to give the GUI some time to respond. During the entire testing process, the SUTs were forced into the foreground, so that even if they started external processes, these did not block access to the GUI.

In the first experiment we applied all three approaches to the applications in Table 2, let each one run until it caused a crash and measured the elapsed time. This process was repeated three times for each application, yielding 81 runs in total and 27 for each approach. Since we had only limited time for the evaluation, we stopped runs that failed to trigger crashes within 24 hours and took 24 hours as the time. This only happened with approach A and we marked the corresponding cells in Table 2 with "*". We used our video records to examine each crash and to determine whether an SUT was really frozen or did only perform some heavy processing, such as importing music or mailboxes in the case of iTunes and Mail.

In the second experiment we had approach C exercise each of the object SUTs again. This time we limited the amount of actions that were generated to 200, since we strived to generate short and comprehensible crashing sequences. If the SUT did not crash after 200 actions, it was restarted. We run C for 10 hours on each application in order to generate several short crashing sequences. After that, we replayed each of the crashing sequences 5 times. We considered the crash to be reproducible if during one of these playbacks the application crashed after the same action as during the initial crash.

3.4 The results

Table 2 lists our findings for the first experiment. It shows that approach B and C were able to trigger crashes for all applications and needed significantly less

time than the default TESTAR (we performed a t-test with significance level $\alpha = 0.05$). These findings are consistent with [8], where the authors carried out their experiments on Motorola cell phones. Consequently, during the experiment, with advanced TESTAR action specification in Prolog, approaches B and C were found to be faster in finding crashes than the default setting (A). The additional effort of writing the Prolog specifications was relatively small for us. As can be found in Table 3, most of the specifications consisted of less than 30 LOCs. Only the specification for Paintbrush is slightly more complex, since we added a few mouse gestures to draw figures into the drawing area. A future additional study should be done with real testers to see if the learning curve is not to steep.

Unfortunately, approach C is not significantly faster in crashing the application than approach B, so we do not have a clear outcome about our Q-learning approach for action selection. Each of the two algorithms seems to perform better for specific applications. We will need to investigate on the cause of this outcome in future work since a quick analysis already found that approach C on the average executes about 2.5 times as many different actions as B, as we expected.

Table 4 shows the results of the second experiment. We were able to reproduce 21 out of 33 triggered crashes, which is more than 60% and yields the answer to RQ2. Some of the reproducible crashes we found were indeterministic so that during some playbacks the application did not crash, whereas during others it did. This might be caused by the fact that the execution environment during sequence recording and sequence replaying is not always identical. Certain factors, which might have been crucial for the application to crash, are not entirely under our control. Such factors are the CPU load, memory consumption, thread scheduling, etc. For future research we consider the use of a virtual machine which might further improve the reproducibility, because it would allow to guarantee the same environmental conditions during recording and replaying.

Table 4. Reproducibility of crashes.

Application	**Crashes**	**Reproducible**
Skype v5.8.0.945	2	1
Word 2011 v14.1.4	8	5
Calculator v10.7.1	4	4
iTunes v10.6.1	2	1
iCal v5.0.3	2	2
PowerPoint 2011 v14.1.4	4	2
Mail v5.2	1	0
Excel 2011 v14.1.4	9	5
Paintbrush v2.0.0	1	1
Total	33	21
Percentage	-	**63.64%**

3.5 Threats to Validity

Some of the crashes that were generated might have been caused by the same faults. Unfortunately, we could not verify this, since we did not have access to the source code. To determine the general reproducibility in RQ2, one would need a set of sequences which trigger crashes which are known to be caused by different faults of different types.

The capabilities of approaches B and C depend on the Prolog specifications and thus on the tester's skills in defining a set of fault sensitive actions. In addition, the experiments in this paper were executed by the researcher that developed the specification functionality and so it could be expected that specifications of a person less familiar with these facilities, and hence the additional effort, could be larger.

Finally, we executed our experiments on a single platform. This might not be representative for other desktop platforms such as Windows or even mobile operating systems like Android and iOS.

4 Related Work

Amalfitano et al. [1] perform crash-testing on Android mobile apps. Before testing the application, they generate a model in the form of a *GUI tree*, whose nodes represent the app's different screens and whose transitions refer to event handlers fired on widgets within these screens. The model is obtained by a so-called GUI-Crawler which walks through the application by invoking the available event-handlers with random argument values. From the GUI-tree they obtain test cases, by selecting paths starting from the root to one of the leaves. They automatically instrument the application to detect uncaught exceptions which would crash the app. They test their approach on a small calculator application.

Liu et al. [12] employ Adaptive Random Testing (ART) to crash Android mobile applications or render them unresponsive. Their algorithm tries to generate very diverse test cases, which are different from the already executed ones. They define distance measures for input sequences and strive to generate new test cases which have a high distance to the ones which are already in the test pool. They test their approach on six small applications among which an address book and an SMS client. In addition to normal user input, like keystrokes, clicks and scrolling, they also simulate environmental events like the change of GPS coordinates, network input, humidity or phone usage.

Hofer et al. [10] apply a smart testing monkey to the Microsoft Windows calculator application. Before the testing process, they manually build an abstract model of the GUI relevant behavior using a language that is based on finite state machines and borrows elements from UML and state charts. The resulting *Decision Based State Machine* acts as both, an orientation for walking through the GUI by randomly selecting event transitions and as test oracle which checks certain properties for each state. They use the Ranorex Automation Framework to execute their test cases.

Artzi et al. [2] 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, such as 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 achieved only low branch coverage during previous sequences.

Huang et al. [11] concentrate on functional GUI testing. 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. In their experiments they test Java applications (some of them are synthetic, some are part of an office suite developed by students) which they execute by performing clicks. Sometimes they have problems with the execution of their sequences, since the GUI model they are derived from is an approximation. Thus, they repair their sequences by applying a genetic algorithm which strives to restore the coverage of the initial test suite. Their techniques are available as part of the *GUITAR* framework[7].

Miller et al. [15] conducted an interesting empirical study on the efficiency of dumb monkey testing for 30 MacOS GUI applications. They developed a monkey that issues random (double) clicks, keystrokes and drag operations on the frontmost application. They even report on the causes of the crashes for the few applications that they have the source code for. Unfortunately, they do not mention how long they were running the tests for in order for the programs to hang or crash.

Among the tools for dumb monkey testing there are: *anteater*[8] which performs crash testing on the iPhone by issuing random clicks for any view (screen) that it finds. *UI/Application Exerciser Monkey*[9] is a similar program for Android and generates random streams of clicks, touches or gestures, as well as a number of system-level events. Other monkey tools are *Powerfuzzer* (an HTTP based web fuzzer written in Python), *GUI Tester*[10] (a desktop monkey for Windows which uses a taboo-list to avoid cycling on the same events) and *MonkeyFuzz*[11] (Windows desktop monkey developed in C#).

5 Conclusion

In this paper we presented TESTAR, a tool for automated testing at the GUI level, together with some new approaches for action selection and specification. Moreover we presented the results of evaluating these new approaches for crash

[7] `http://sourceforge.net/projects/guitar`
[8] `http://www.redant.com/anteater`
[9] `http://developer.android.com/tools/help/monkey.html`
[10] `http://www.poderico.it/guitester/index.html`
[11] `http://monkeyfuzz.codeplex.com`

testing of a set of real-world applications with complex GUIs. The strengths of our tool are:

- Easy specification of even complex actions: This enables the tester to restrict the search space to the interesting operations. It also allows to go beyond simple clicks and keystrokes to generate mouse gestures which drive even complex GUIs and exercise the majority of their functionalities.
- Revealed faults are reproducible: Since crashing sequences are often relatively short and their actions are parameterized with the widgets they are executed on, they can be reliably replayed, which makes the majority of the crashes reproducible. This allows the developer to better understand the fault and to collect additional information during playback. He may even replay the sequence in slow motion to observe the course of events.
- Since we employ the operating system's Accessibility API the SUT does not require any instrumentation which makes the approach feasible for many technologies and operating systems: Many applications are not designed with testing in mind and it can be difficult to add testing hooks later on [7]. Our framework still allows to test those applications, without any instrumentation effort. We deliberately do not make use of any coverage techniques based on source code or bytecode instrumentation. For many large applications it is impractical or even impossible to measure code coverage, especially if the source code is not available. And by far not all applications run in virtual machines such as the JVM or Microsoft's CLR. Techniques that rely on bytecode instrumentation can certainly make use of additional information to guide the test case generation more effectively, but they are restricted to certain kinds of technologies. We strive for general applicability.

The results that we obtained from our experiments are encouraging and show the suitability of the approach for nontrivial SUTs. We proved that complex popular applications can be crashed and that those crashes are reproducible to a high degree. Once setup, the tests run completely automatic and report crashes without any additional labour.

6 Future Work

Although first results are encouraging, more experimentation needs to be done to find out why the Q-learning approach did not work as expected and how difficult writing Prolog specifications would turn out for real test practitioners.

Our current implementation runs on MacOSX, TESTAR is not restricted to any particular platform and we are in the process of developing support for Microsoft Windows. In addition, we plan implementations for other operating systems such as Linux and Android. Our approach benefits from the fact that many platforms provide an Accessibility API or one to access window manager information (such as the WinAPI).

The approach presented in this paper currently only targets critical faults such as crashes. However, we plan to extend our framework to develop techniques

for automated regression testing. Our ambitious goal is to completely replace the fragile and inherently labor intense capture and replay method and to develop a more effective and automated approach to regression testing. Therefore, we will have to use a powerful oracle which allows us to detect not only crashes but also functional faults such as incorrect output values in text fields, layout problems, etc. One way to achieve this is to perform back to back testing with two consecutive versions of an application. In this scenario, the old version serves as the oracle for the new one. The difficulty lies in detecting the intended (new features) and unintended (faults due to modifications) differences between the widget trees in each state in order to reduce the amount of false positives.

Acknowledgment

This work is supported by EU grant ICT-257574 (FITTEST) and the SHIP project (SMEs and HEIs in Innovation Partnerships) (reference: EACEA / A2 / UHB / CL 554187).

References

1. D. Amalfitano, A.R. Fasolino, and P. Tramontana. A gui crawling-based technique for android mobile application testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 252–261, March 2011.
2. Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. A framework for automated testing of javascript web applications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 571–580, New York, NY, USA, 2011. ACM.
3. S. Bauersfeld, A de Rojas, and T.E.J. Vos. Evaluating rogue user testing in industry: An experience report. In *Research Challenges in Information Science (RCIS), 2014 IEEE Eighth International Conference on*, pages 1–10, May 2014.
4. Sebastian Bauersfeld and Tanja Vos. A reinforcement learning approach to automated gui robustness testing. In *In Fast Abstracts of the 4th Symposium on Search-Based Software Engineering (SSBSE 2012)*, pages 7–12. IEEE, 2012.
5. Sebastian Bauersfeld and Tanja E. J. Vos. Guitest: a java library for fully automated gui robustness testing. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 330–333, New York, NY, USA, 2012. ACM.
6. Sebastian Bauersfeld, Tanja E. J. Vos, Nelly Condori-Fernández, Alessandra Bagnato, and Etienne Brosse. Evaluating the TESTAR tool in an industrial case study. In *2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, Torino, Italy, September 18-19, 2014*, page 4, 2014.
7. S. Berner, R. Weber, and R.K. Keller. Observations and lessons learned from automated testing. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 571–579, May 2005.
8. C. Bertolini, G. Peres, M. d' Amorim, and A. Mota. An empirical evaluation of automated black box testing techniques for crashing guis. In *Software Testing Verification and Validation, 2009. ICST '09. International Conference on*, pages 21–30, April 2009.

9. Ivan Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
10. B. Hofer, B. Peischl, and F. Wotawa. Gui savvy end-to-end testing with smart monkeys. In *Automation of Software Test, 2009. AST '09. ICSE Workshop on*, pages 130–137, May 2009.
11. Si Huang, Myra Cohen, and Atif M. Memon. Repairing gui test suites using a genetic algorithm. In *ICST 2010: Proceedings of the 3rd IEEE International Conference on Software Testing, Verification and Validation*, Washington, DC, USA, 2010. IEEE Computer Society.
12. Zhifang Liu, Xiaopeng Gao, and Xiang Long. Adaptive random testing of mobile application. In *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*, volume 2, pages V2–297–V2–301, April 2010.
13. Atif Memon, Ishan Banerjee, Bao Nguyen, and Bryan Robbins. The first decade of gui ripping: Extensions, applications, and broader impacts. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*. IEEE Press, 2013.
14. Atif M. Memon. *A comprehensive framework for testing graphical user interfaces*. 2001. Advisors: Mary Lou Soffa and Martha Pollack; Committee members: Prof. Rajiv Gupta (University of Arizona), Prof. Adele E. Howe (Colorado State University), Prof. Lori Pollock (University of Delaware).
15. Barton P. Miller, Gregory Cooksey, and Fredrick Moore. An empirical study of the robustness of macos applications using random testing. In *Proceedings of the 1st International Workshop on Random Testing*, RT '06, pages 46–54, New York, NY, USA, 2006. ACM.
16. Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.