Automated Testing at the User Interface level

Tanja E.J. Vos
Universidad Politécnica de Valencia
Open Universiteit
Email: tvos@pros.upv.es, tanja.vos@ou.nl

Urko Rueda Molina
Universidad Politécnica de Valencia
Email: urueda@pros.upv.es

Wishnu Prasetya
Universiteit van Utrecht
Email: s.w.b.prasetya@uu.nl

I. 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 [1] and are often large and complex. To be effective, UI testing should be automated.

A substantial part of the current state-of-the art for automating UI testing is still based on the Capture and Replay (CR) technique [2]. CR requires significant human intervention to record interactions (i.e. clicks, keystrokes, drag/drop operations) that are used as regression tests for new product releases. A known concern of CR is that it creates a critical maintenance problem because the test cases easily break when the UI evolves, which happens often. A more advanced technique, Visual testing [3], takes advantage of image processing algorithms to simulate step by step human interactions. Though visual approaches simplify the work of testers, they are slow, imprecise (prone to false positives with wrong UI element identification, and false negatives with missed UI elements), and also rely on the GUI stability.

We present a completely different approach to automated GUI testing called TESTAR[1] (Test Automation at the user inteRface level). TESTAR automatically and dynamically generates test sequences based on a tree model (automatically derived from the UI through the Accessibility API). No test cases are recorded and the tree model is inferred for every state, this implies that tests will run even when the GUI changes. This reduces the maintenance problem that threatens the techniques mentioned earlier.

II. TESTAR engine for Automated User Interface Testing

TESTAR performs the steps as is shown in Fig. 1: (1) start the SUT; (2) obtain the GUI's *State* (a widget tree[2]); (3) derive a set of sensible actions that a user could execute in a specific SUT's state (i.e. clicks, text inputs, mouse gestures); (4) select one of these actions (random or using some search-based optimization criteria); (5) execute the selected action; (6) apply the available oracles to check (in)validness of the new

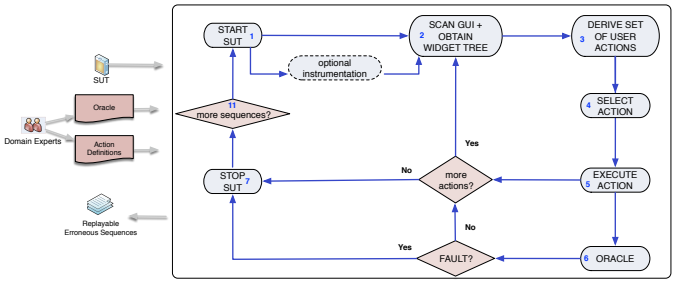

Fig. 1. *TESTAR testing flow*

UI state. If a fault is found, stop the SUT (7) and save a replayable sequence of the test that found the fault. If not, keep on testing if more actions are desired within the test sequence.

Using TESTAR, you can start testing immediately and you do not need to specify test cases in advance. TESTAR automatically generates and executes test sequences based on a structure that is automatically derived from the UI through the accessibility API. Without specifying anything, TESTAR can detect the violation of general-purpose system requirements through implicit oracles like those stating that the SUT should not crash, the SUT should not find itself in an unresponsive state (freeze) and the GUI state should not contain any widget with suspicious words like *error*, *problem*, *exception*, etc.

To ease testers work, the tool provides a basic testing protocol that can be customized for concrete needs in particular SUTs. Protocol customization is performed through an end-user compilable Java class file. The class contains a method for each of the steps 1 to 7 in Fig. 1). For step 6, the method `getVerdict` defines the test oracles in the form of predicates over the SUT's GUI states. These oracles are the requirements that are required to hold in all the GUI states that are reachable from the SUT initial states. Out of the box, TESTAR provides an implementation of `getVerdict` that contains oracles for checking *general-purpose requirements*, i.e. those that need to hold for any software system independent of its context and specific objectives. These are for example requirements that state that: the SUT should not crash, the SUT should not find itself in an unresponsive state (freeze), the GUI state should not contain any widget with suspicious words like *error*, *problem*, *exception*, etc. to the user". Currently, oracles in TESTAR are expressed in Java and Figure 2 shows the code for implementing the general-purpose oracles.

III. Incremental Oracle and Requirements constructions

While our application is automatically being tested for stability, crashes and undesired outputs, we can start adding

[1] http://www.testar.org

[2] TESTAR uses the Operating System's Accessibility API, which has the capability to detect and expose a GUI's widgets, and their corresponding properties like: display position, widget size, ancestor widgets, etc.

```
protected Verdicts getVerdicts(State state){
   Assert.notNull(state);
   verdicts = new Verdicts():
   verdicts.add(oracle_Crash (state));
   verdicts.add(oracle_Responsiveness(state));
   verdicts.add(oracle_ SuspiciousTitles(state));
   return verdicts;
}
protected Verdict oracle_Crash (State state){
   if(!state.get(IsRunning,false))
    return new Verdict("System offline! It crashed?");
}
protected Verdict oracle_Responsiveness (State state){
   if(state.get(NotResponding, true))
    return new Verdict("System is unresponsive!");
}
protected Verdicts oracle_SuspiciousTitles(State state){
    verdicts = new Verdicts():
    String titleRegEx = settings().get(SuspiciousTitles);

   // search all widgets for suspicious titles
   for(Widget w : state){
        String title = w.get(Title, "");
        if(title.matches(titleRegEx)){
        verdicts.add(new Verdict("......"));
   }
   return verdicts;
 }
```

Fig. 2. *Default oracles for general-purpose requirements in TESTAR*

more and more oracles that test more specific requirements of our application. This way we incrementally create the requirements, this is something that turns out to be very helpful when dealing with legacy systems.

To test other requirements, we express them as oracles and add them to the method `getVerdict`. The idea is that the tester only specifies *what* quality properties should hold. This way we build up the requirements *while we test* and testing can start early and even in the absence of clear requirements.

As an example, think we want to test an accessibility requirement from the W3C WAI that states *all images should have an alternate textual description*. We just need to add an oracle like the one below:

```
protected Verdicts oracle_ImagesTextDescripWAI(State state){
    verdicts = new Verdicts():
    Role role = w.get(Tags.Role);
    if (role.equals("UIAImage") && title.isEmpty())
        verdicts.add(new Verdict(0.1,
          "No alternate text descr");
   return verdicts;
 }
```

Other typical examples of requirements that we can express and test: all colour combinations between foreground and background are legible; the content of text box T always matches a regular expression R; all hyperlinks have a valid target site; the font, size and colour of each label are coherent with some accessibility standard; all widgets of type T in a range R share the same colour attributes (style guide rule); the content of text box T always matches a regular expression R; all hyperlinks have a valid target site, etc.

IV. Technology agnostic

TESTAR adopts the hypothesis that the majority of UIs are conceptually very similar. The only thing that varies is the underlying technology and the look and feel. But if sufficient state information is available i.e. the types, positions and properties of all widgets on the screen then testing an iPhone App is not much different from testing a Windows desktop application or an HTML website.

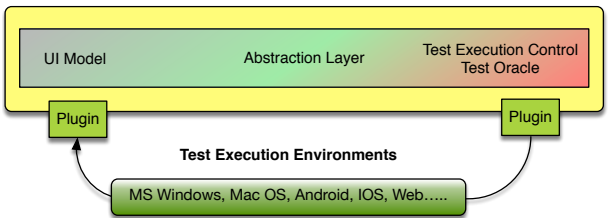

Fig. 3. *Abstraction layer and the extensible plugin architecture of TESTAR*

TESTAR's abstraction layer and the extensible plugin architecture of TESTAR (see Figure 3) makes it highly technology agnostic. The plugins will deal with the process of fetching the state information and executing UI actions for different platforms. The abstraction Layer, which will have a uniform interface that allows to access the UI state information in a standardised way. It will allow to simulate input in the form of clicks, drag and drop operations, swipes, pinches, audio input, etc. in order to operate the UI. The abstraction layer abstracts from different technologies, shields other components from technologic details and allows testers to concentrate on strategic parts of sequence and test suite generation.

V. Conclusions

TESTAR has been successfully applied in different industrial context [4], obtaining feedback for improved innovation transfer and eventual market-adopting.

Currently, TESTAR is presented as one of the research results that are transferred to industry as part of the Spanish (and soon also the European) Software Testing Innovation Alliance[3]. This alliance brings together key stakeholders in software testing to jointly work to improve innovation and technology transfer from University to Industry. The objective is to increase research impact in practice, education, business and also feedback into the research.

Acknowledgments.: FITTEST project, ICT-2009.1.2 no 257574, SHIP project (EACEA/A2/UHB/CL 554187) and the PERTEST project (TIN2013-46928-C3-1-R).

References

[1] A. 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).

[2] B. N. Nguyen, B. Robbins, I. Banerjee, and A. M. Memon, "Guitar: an innovative tool for automated testing of gui-driven software," *Autom. Softw. Eng.*, vol. 21, no. 1, pp. 65–105, 2014.

[3] E. Alegroth, M. Nass, and H. Olsson, "Jautomate: A tool for system- and acceptance-test automation," in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, March 2013, pp. 439–446.

[4] T. E. J. Vos, P. M. Kruse, N. Condori-Fernández, S. Bauersfeld, and J. Wegener, "TESTAR: Tool support for test automation at the user interface level," *Int. J. Inf. Syst. Model. Des.*, vol. 6, no. 3, pp. 46–83, Jul. 2015. [Online]. Available: http://dx.doi.org/10.4018/IJISMD.2015070103

[3] http://innovationalliance.eu