Visualization of automated test results obtained by the TESTAR tool

Urko Rueda, Anna I. Esparcia-Alcazar, and Tanja E.J. Vos

Research Center on Software Production Methods (PROS)
Universitat Politècnica de València
Camino de vera s/n 46022, Valencia, Spain
{urueda,aesparcia,tvos}@pros.upv.es
http://www.testar.org

Abstract. Bigger and more complex software systems demand quality practices that are seldom carried out in real industry. A common practice is to provide a post-release maintenance service of products to correct defects reported by the end user. In previous work we presented TESTAR, a technology-agnostic tool for automated testing of applications from their GUI. Here we introduce state-transition graph models derived from TESTAR test results as a tool for visualisation of what has been tested, to which extent and which software defects were found. We discuss how such models enable to perform quality assessment of software products by inspecting and debugging the system behaviour from the GUI perspective. This constitutes a step forward in aid of software developers and testers, since the User Interface is commonly the means end-users encounter potential software defects.

Keywords: Automated Testing, User Interface Models, Quality Assessment, Visualization

1 Introduction

Visualisation is a historical mechanism that contributes to human understanding of large sets of information. From city, bus and metro maps to UML models for software design, presentation slides to communicate to attendees, hardware resources graphs to display their usage by different applications processes, the list is numerous. In computer science it does not matter how smart a system could be, human control is still critical in most scenarios, including software design and software testing.

In previous work we presented TESTAR [8] and its successful application to various industrial systems [9]. TESTAR is capable of automated test generation and executes test cases based on a tree model that is automatically derived from the application's GUI through accessibility API technologies. Since this structure is built automatically during testing, the GUI is not assumed to be fixed and tests still run even when the GUI has been modified. This reduces the maintenance problem that threatens most current state-of.the-art approaches in GUI testing, like Capture-Replay [26, 24, 14, 15, 11] and Visual Testing [29, 3].

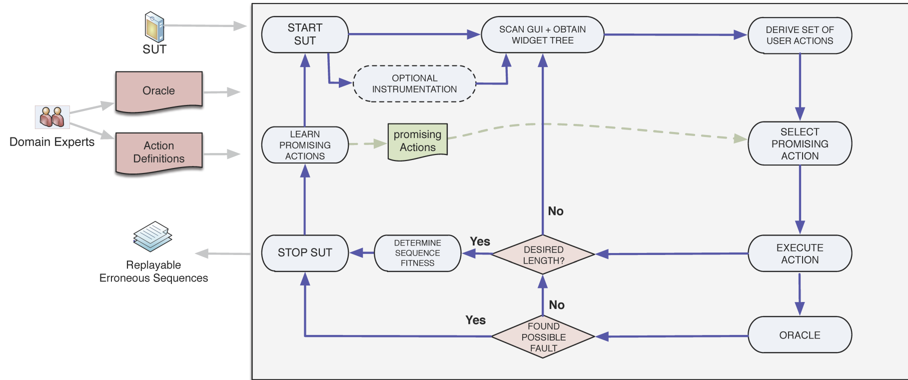

Fig. 1. TESTAR testing cycle

The TESTAR tool [25] carries out automated testing, Figure 1, on the SUT's GUI by deriving sets of possible actions for each state that the GUI is in, and automatically selecting and executing appropriate ones until a stopping criteria is reached. A tester can take full control of tests, by modifying a TESTAR default provided testing protocol which enable to establish the stopping criteria, which actions are available from the GUI, oracles, etc. Oracles are used to detect faulty behaviour when a system crashes or freezes. Besides these simple,"free" oracles, the tester can easily specify regular expressions that detect patterns of suspicious titles in widgets that might pop up during the execution. For more sophisticated and powerful oracles, the tester can enrich the default TESTAR (Java-based) protocol that is used to drive the tests.

SUT GUI states are computed as the set of hierarchical widgets (user interface elements) that conform its GUI, called widget trees in TESTAR. These widget trees are gathered through the Operating System's Accessibility API, an assistive technology that has the capability to detect and expose the GUI widgets and their corresponding properties[1].To illustrate the concept, the left part of Figure 2 shows the GUI of a very simple application, with its corresponding widget tree on the right. It has a titlebar (with text *Example*), a menu bar, a button, a text-field and a slider. It also includes an example of actions that could be performed by the user: dots for clicks, ABC text for text-input, arrows for drag&drop operations in the slider.

TESTAR has been deployed in various industrial environments [25], which has allowed us to receive feedback from the end users. Companies have been keen to take up TESTAR, yet they would appreciate a visualisation aid in order to see what has happened and what has been tested. We are also aware such an aid would enable us to debug testing performance, check which parts of the application GUI were exercised, and how TESTAR achieved to crash or reveal software defects.

[1] Such as display position, widget size, ancestor widgets, etc.

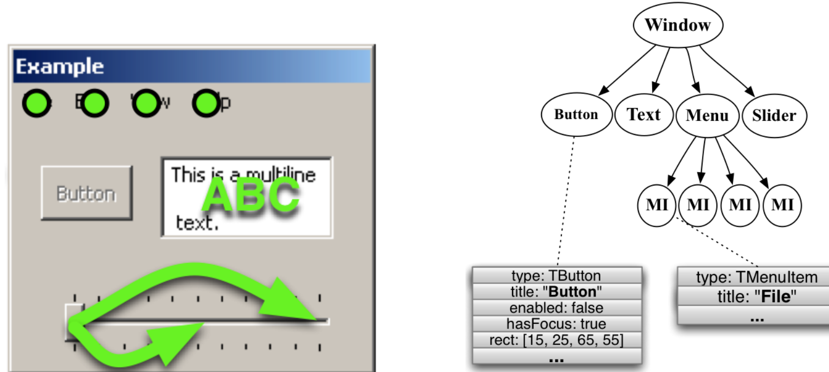

Fig. 2. An example GUI state (left) and its corresponding widget tree (right)

Here we present the results of our work on visualisation of the TESTAR outputs. The functionality described in the paper is already supported by the tool (https://github.com/STaQ-PROS-UPV/TESTAR). Section 2 gives an overview of related work in the area. Section 3 introduces the main contribution of this paper, namely the TESTAR graphical models, including a description of how they can be used for quality assessment. Finally, in section 4 we present some conclusions and outline areas for future work.

2 Related work

Model extraction using dynamic GUI crawling is a popular research topic when adopting model-based testing (MBT). It addresses the challenge of the effort and expertise required on crafting the application models. In [5] an overview is given of the existing body of knowledge on GUI testing, showing that 52% of the studied papers were about Model-Based GUI Testing approaches. It was found that the models that were mostly used to generate tests were Event Flow Graphs (EFG) and Finite State Models (FSM). Moreover, most approaches are limited to specific programming language environments.

Work related to EFG models has mostly been carried out by the GUITAR team [23] and is based on reverse engineering (called ripping [18]) of a EFG GUI model that represents all possible interaction sequences with the GUI widgets. Such a model is then used for test case derivation. GUITAR (available at http://guitar.sourceforge.net/wiki/index.php) was initially developed for Java applications, but extensions have been made for iPhone, Android and Web applications. Murphy Tools [1, 2] is a GUI Driver tool based on GUITAR, that presents a dynamic reverse engineering tool for Java GUI applications, and an iterative process of manually providing valid input values and automatically improving the created state-models. Murphy Tools automatically extracts models of the GUI of applications as the user interacts with them. Then, the models are

used to drive testing. This approach differs from TESTAR in two ways. First, the manual work required to exercise the applications for models population. Secondly, tests are guided by the extracted models. The effectiveness of the tests would rely directly on the goodness of extracted models. In contrast, TESTAR approach performs automated testing without the need of models. It is true that models are still extracted and, what is more, the models can be applied for more intelligent testing (i.e compared to randomly selecting GUI actions) when GUI space exploration is a main concern.

FSM models are used in [16] where tests are generated for AJAX applications using Hill-Climbing and Simulated Annealing on an approximate model that is obtained from the applications Document Object Model tree. [20, 21] present a tool called Crawljax that uses dynamic analysis to construct a FSM of an AJAX application using web crawling, from which a set of test cases is generated. WebMate [10] is another model extractor for web applications. It extracts a so-called a usage model, a graph where nodes correspond to states of the application and a transition represents a single interaction with the application. In [19] we can find automated crawling of web applications' models. They focus on Javascript and DOM to dynamically analyse candidate events to perform on the GUI and that would change the application state. Yet, in contrast to TESTAR, the tester does not have control over the events that would be interesting to perform. This is mainly because such approaches are focused to craft a model of the application, while TESTAR seamlessly drives testing on tester controlled testing protocol.

In [22] an FSM based approach is presented called GUI Test Automation Model (GUITam) for constructing the state models by dynamic analysis. The GUITam Runner works for C++ applications.

With the increased availability of mobile platform applications in our daily life there is a trend to switch testing from traditional desktop platform to the mobile context. In [28] static analysis is combined with dynamic GUI crawling to test Android applications. They use static analysis of the application source code to extract the actions supported by the GUI of the application. Next, they use dynamic crawling to build a FSM of the application by systematically exercising the extracted events on the live application. They compare their approach to the Android version of GUITAR.

The iCrawler tool [13] is a reverse engineering tool for iOS mobile applications that also uses an FSM.

Other types of models are used too. In [4] a so-called GUI tree is used as a model to perform crash testing on Android mobile apps (the nodes represents user interfaces of the app, while edges describe event-based transitions). Similar to [20], the model is obtained by a so-called GUI-Crawler that walks through the application by invoking the available event-handlers with random argument values. From the GUI-tree test cases are obtained by selecting paths starting from the root to one of the leaves. The application under test is instrumented to detect uncaught exceptions that would crash the app. Morgado et al. present

ReGUI tool [12] that uses dynamic analysis to generate Spec# and GraphML GUI models to test windows applications.

In [17] a tool called AutoBlackTest is presented that does dynamic analysis for model extraction and test suite generation for GUI applications. The tool uses IBM Rational Functional Tester (RFT) to extract the list of widgets present in a given GUI, to access their state and to interact with them. RFT supports a range of technologies to develop GUIs and offers a programmatic interface that is independent from the technology used to implement the GUI of the application under test.

Maintenance is one of the major drawbacks when using capture & replay tools, as changes to the user interface can result in laborious manual adaption work. Therefore, approaches to automating test suite maintenance [11] become crucial. In this line, work by Leotta et al. e.g. focuses on making existing approaches more robust to changes to the application [15]. Sun and Jones perform analysis of the underlying application programmable interface (API) in order to generate GUI tests [27].

3 TESTAR State-Transition Graphs

TESTAR testing cycle iterates a pair of action selection and execution for each UI state the SUT is in. Each UI state is represented by an unique identificator, and each action from that concrete state is also represented by an unique identifier. TESTAR populates the graphs by pairs of state-ID and action-ID as actions are being executed in a test sequence. A TESTAR graphing utility retrieves such pairs of states and actions and populates the test graphs.

To be able to visualise TESTAR test runs, we have defined the TESTAR State-Transition Graph, in which each node represents a unique GUI state and has an associated unique state identifier. The graph edges, which have an associated unique action identifier, represent a state transition caused by a user-action, such as left/right clicks, text input and drag&drop operations. An example is given in Figure 3, which shows a fragment of the corresponding TESTAR State-Transition Graph Model for a test run.

Each node contains a label xxx (y), where *xxx* represents the state identifier and *y* the number of times the state was traversed on a test execution. Similarly, labels for actions include identifier, number of repetitions and an extra parameter between square brackets which indicates the action sequence number in the test execution. In this way we have the ability to explore the executed test action by action. We discuss the quality assessment of applications in Section 3.5.

Nodes[2] and links in dashed lines, which include the label *unexplored*, refer to actions that are available from a given GUI state, but that were not executed during a test. The number of actions that were not executed appears between brackets.

[2] The number following the label *unexplored* within nodes merely refers to the count of states not fully explored

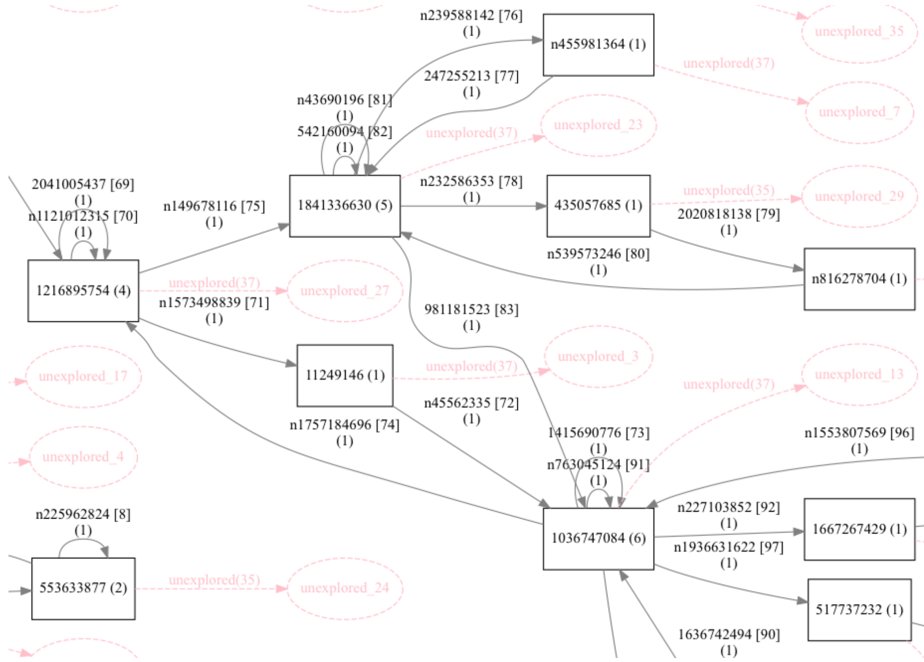

Fig. 3. Fragment of the TESTAR graph (tiny) for a 99-action test sequence for Windows calculator

3.1 Unique identifiers for states and actions

TESTAR reports test runs as a sequence of actions, each of which is a transition from two GUI states that might be the same (the GUI does not change) or different (the action produced an GUI change). Providing unique identifiers to those states and actions could potentially help in the inspection and analysis of test results. Several applications can be think of: quick search and jump to concrete states and actions by their unique id, cross test run matching were a state from test run X can be localised into a test run Y, etc. In the end, we believe these identifiers are essential to support debugging when there is a need to jump into concrete states and actions.

To make states and actions uniquely identified even across different runs of the SUT, we need to assign a unique and stable identifier to each state and action. TESTAR allows us to derive the complete GUI state of the SUT (i.e. the widget tree as explained in Section **??**), so that we can inspect the property values of each visible widget. For instance, 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 combination of the button's property values, like its title, help text or its position in the widget hierarchy (which parents / children the button has). 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.

Each action type (i.e. a click or a keystroke) may have parameters. For example, a click action has two parameters: the button (i.e. left or right) and the clicking position (x and y coordinates). The action identifier used for the graph link takes parameters into account. Hence, the action that types the text *foo* will have a different identifier than the action that types *boo*. What is more, the identifier also depends on the state the action is performed. This approach will enable to precisely locate an action in the graph by its unique identifier. 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.

3.2 Reusing the TESTAR graphs

TESTAR GUI models are state-transition diagrams/graphs described with plain text graph description language (DOT), which makes them reusable by third party tools. The DOT format is illustrated in the following sample code.

```
digraph DOT { rankdir=LR;
START [shape=point, height=0.3, style=solid, color=black];
START -> state_1;
state_1 -> state_2 [label="transition"];
state_2 -> state_2 [label="loop"]; state_2 -> END; }
```

It mainly consists of links between nodes, e.g. transition from state 1 to state 2; graphical properties can be set for different graphical representations.

The rationale behind using this simple dot format is that it offers enough capabilities for the representation of the UI test sequences from the perspective of acquiring an overview of the execution. Node labels can be applied to UI states identifiers and link labels to state' actions identifiers. The graphical properties of the dot format would also enable to remark special characteristics of the test sequences, as for example a varying color to indicate a higher repetition of certain actions. At a glance, the colors would be indicators of how much were the different parts of the UI exercised.

3.3 Test results graphs

TESTAR graphs comes in three varying details: *minimal*, without states and actions identifiers, *tiny*, which incorporates identifiers to the minimal version and *screenshoted*, which embeds GUI screenshot to states and GUI action target screenshot to transitions.

A *minimal* detailed version will help improve visualisation and get an overview of the SUT GUI space explored by a test sequence; an example of this can be found in Figure 4. This is done by removing the identifiers from states and actions, and the order of actions in the test which happen to crowd the graphs

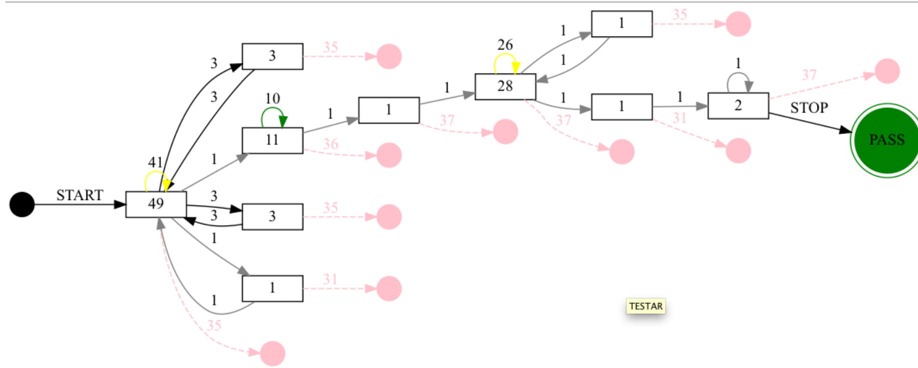

Fig. 4. Windows calculator TESTAR Graph (minimal) for 99 actions test sequence

with too much information for a simple overview of the complete test sequence. This is indeed more crucial the bigger the graphs are.

For a regular *tiny* version, the graphs provide enough information to search and identify the states and actions from test runs. The order, between brackets, in the actions enable to trace a test run action by action. This version is less readable than the minimal one due to the extra information displayed, but will enable to debug and match a test run to its corresponding GUI state-transition graph.

TESTAR also keeps a *screenshot-version* of the graphs, see Figure 5, that can be directly used for test documentation. Embedding the GUI screenshots for graph states and actions GUI target screenshots (e.g. the widget screenshot for a button click) provides more meaningful information about test runs. It allow to read which GUI states were exercised by a test run and the SUT behaviour after each executed action.

3.4 Abstracting the graphs

Some actions tend to populate the graphs almost to infinite, which would be problematic for graph visualisation due to their large extension. For example, a text-field widget might accept any kind of text input, which translates into nearly infinite ways of populating the text-field. The corresponding graph would consist of a different state for each different text input: *stateX, input-a, state-a, input-b, state-b, ..., input-n, state-n*. In this concrete sequence, if we start at state *X*, giving the input "a" for a text field will end in the state *b*, and so on with the input "b" to input "n". To avoid such expansion of the graph and maintain it more scalable, we present next an **abstraction mechanism**.

We have shown above how the graph states and actions identifiers are computed to uniquely identify them. States consider the widget properties *Role*, *Title* and *Shape*, while actions do take into account the GUI state from which it is performed and its event parameters (i.e. *some text* in a typing action).

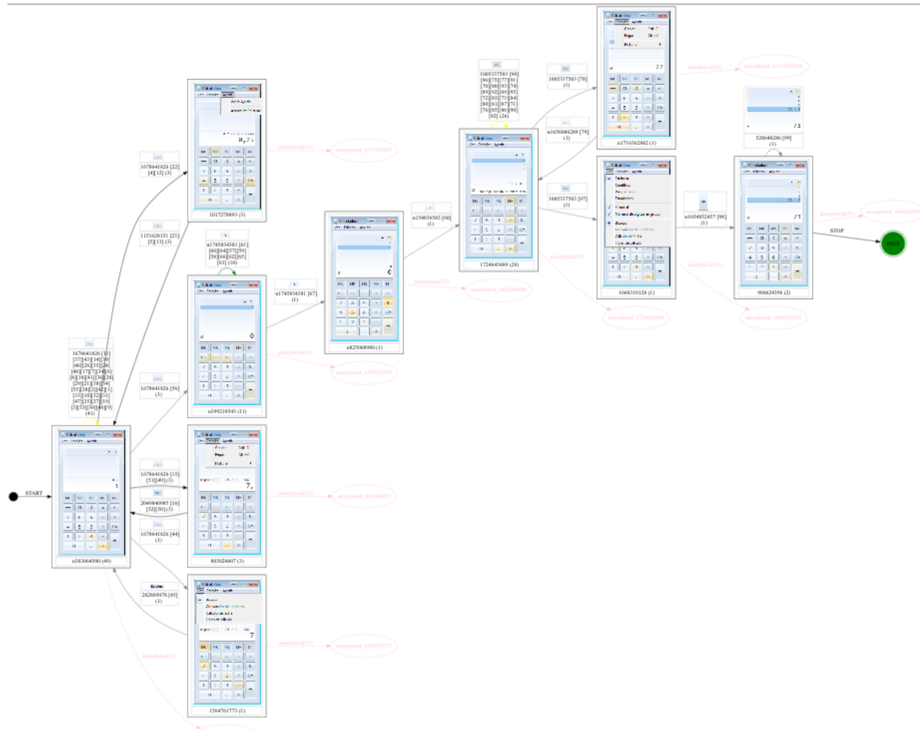

Fig. 5. Windows calculator TESTAR Graph (screenshot) for 99 actions test sequence

We can achieve abstraction by recoding the identifiers. Abstract state identifiers are computed by only considering the *Role* property of a widget. Thus, we can abstract the GUI by looking into its widgets hierarchy and widget types. For actions, the abstract identifier is calculated by discarding event parameters, concretely *mouse position* and typed texts. For the former, in the example, a left/click mouse action is performed at a fixed coordinate (the mouse position), but we are only interested on the action type (thus, ignoring the coordinates). For the latter, typing some text is abstracted ignoring what is typed.

Then, each unique state and action has an associated abstract identifier (a cluster of states or actions respectively) that enables TESTAR to cluster the graphs, collapsing states and actions into a simpler representation of the GUI model. Clusters are reported together with graphs, for example (cluster uses the abstract identifier):

```
  Cluster (5) 1151628151 contains:
(1) n1626056442 (2) 1669305354 (3) n86735519
```

We would like to remark that the current version of TESTAR screenshot version of the abstracted graphs are merely illustrative because each abstracted state and action is the combination of many states and actions, respectively. The

combination of several screenshots into a single abstracted screenshot is out of the scope of this paper. However, we can jump into the non-abstracted graphs to get a precise scenario of states and actions for our test sequence.

3.5 An example of application: quality assessment

The TESTAR approach aims to automate testing in industry, independently of the size and complexity of applications and the technology underneath. Such a technology-agnostic automation approach needs to be largely independent of the source code, especially as companies are reluctant to share their source code and are not keen on instrumentation or injection. Consequently, we need to complement our approach with quality metrics that are also independent of the source code. This way we can provide industry with quality indicators about the testing activities done with TESTAR. Being aware of the extent at which our software applications are being tested, it can provide and indicator of how reliable we could expect our software to be. While traditional testing provides metrics on code coverage and mutation metrics, indicators of how much of the code was exercised, a technology-agnostic black-box (testing outside the code or binary) testing approach like TESTAR requires alternative approaches to report the extent at which software are exercised/stressed.

The TESTAR graphing utility presented in this paper provides SUT GUI space exploration metrics. It reports, every X actions, on the number of unique and abstracted states and actions that were traversed/executed during a test. For example, a test run of 50 actions with a sampling every 10 actions may report the next exploration curve data:

```
________________UNIQUE ________ABSTRACT ___________TOTAL
   #, states, actions, states, actions, unique, abstract
   1,      4,      10,      2,       2,     14,        4
   2,      7,      20,      3,       3,     27,        6
   3,     13,      30,      5,       6,     43,       11
   4,     16,      40,      6,       7,     56,       13
   5,     22,      50,      6,       7,     72,       13
```

Additionally, TESTAR also reports the number of states for which there were discovered actions that were not explored/executed. The corresponding test report of the previous exploration curve data would include metrics for unexplored states and actions and a test verdict (PASS or FAIL):

```
_____________________________STATES
total, unique, abstract, unexplored (with unexplored actions)
   51,     22,        6,         21 (total = # actions + 1)

____________________________ACTIONS
total, unique, abstract, unexplored (discovered but unexecuted)
   50,     50,        7,        638
```

```
____________TOTAL
 unique, abstract, ... VERDICT
     72,       13, ...    PASS
```

Looking to this sample metrics we can make several observations. First, the exploration curve indicates how fast a test sequence is exploring the UI of the application. The last 10 executed actions does not increase the number of abstract states and actions explored (6 and 7 respectively). It could suggest a possible stopping criteria of the test sequence when we want to broad exercise the UI of the application. In contrary, the last 10 actions also suggest that a specific part of the UI is being stressed. This might be also interesting to discover faults that may arise by hard stressing a concrete part of the UI.

Secondly, the unexplored states and actions metrics provide indicators of un-exercised parts of the UI. Again, this may contribute to decide on the tests stopping criteria. The ideal testing would continue until no unexplored states or actions are reported. Yet, the screenshot version provides information on which unexplored UI parts should be further tested.

A TESTAR graph, from an executed test, depicts which SUT GUI states are reached by concrete actions. It provides (specially the screenshot versions) a straightforward documentation of how our system behaves. Most importantly, this documentation can be obtained automatically as soon as a new product version is available for testing. The graph will sketch how exactly the system reacts to user actions. The feedback received from our industrial partners indicates that this is a good way to proceed with test quality reporting in TESTAR.

4 Conclusions

TESTAR is a technology-agnostic automated testing solution that enables analyses of the quality of software from their GUI through a customised tester testing protocol, which enable to control which actions can be performed on the GUI, which actions should be executed and how to detect defects or suspicious behaviour through implementing SUT specific oracles.

In this paper we have presented an approach to visualisation that enables TESTAR to provide visual information of test runs. Test visualisation helps testers to directly inspect and analyse how the SUT was exercised in the search of potential failures. Without this feature we will only know whether a test pass or fails and, if reported, the raw sequence of actions that were performed. However, raw actions force to reproduce them in the GUI to check what was executed. Mechanisms to visually check how the tests performed are helpful if we want to get a deep insight of how the SUT was tested and what made a SUT to reveal a failure. In this line, TESTAR screenshoted graphs enable to inspect and analyse the tests. Even more, these graphs can be used as test documentation, from which human testers might perform manual tests (e.g. for failure reproducibility) Other than testers, developers can also benefit from this

information which makes them aware of concrete test sequences that break the SUT. TESTAR is also able to measure the size and complexity on GUI basis based on the number of GUI states and transitions explored by test runs. We have also introduced how to assess the quality of software products looking into the reported data and graphs models.

We are confident that TESTAR will help software industry to adopt testing automation and that it will enable to reduce the maintenance costs on new software releases. TESTAR is not oriented to update test artifacts, but to keep a working testing protocol (i.e. widgets/actions recognition, oracles) that will yet run on new releases. However, we are aware that there is still manual work to perform, the most critical part being the oracles definition. We foresee that oracles should be a new trending topic in testing development that will be mixed into the traditional software development methodologies. For example, a SUT working with a database should provide the mechanisms to check the data consistency: a) in the database itself, b) in the data sent from the SUT to the database and c) in the data retrieved from the database to the SUT GUI.

We are aware that one potential problem with our visualisation approach stems from the fact that bigger systems would produce bigger graphs with a higher amount of nodes/states. This is a critical scalability problem, specially on screenshoted versions of graphs, that could potentially introduce technical difficulties for inspecting and assessing the models. Addressing this potential problem is left for future work. Yet, our plan is to break the graphs into small interconnected portions. Each portion could be associated with concrete portions of the whole application GUI. We have already worked with graphs comprising of more than 250 abstract states and more than 500 abstract actions, due to test sequences of 5000 actions over the windows calculator. Although the graphs are big, we can easily jump to interesting parts by searching the identifiers of states and actions. More advanced inspecting features will be also incorporated (e.g. state centered re-graphing, displaying the closest interconnected states and actions).

Finally, we acknowledge that the TESTAR approach relies on the applications accessibility compliance, as TESTAR makes use of the platform native Accessibility API to retrieve the widget-trees presented in this paper. However, we expect to address the issue by supplying specific drivers that provides application widget-trees and events handling. We have already succeed on this line by supporting the Android platform for mobile phones.

An empirical evaluation of the visualization approach is also left for a consequent publication.

5 Acknowledgement

This work was partly funded by 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. Aho, P., Menz, N., Rty, T., Schieferdecker, I.: Automated java gui modeling for model-based testing purposes. In: Information Technology: New Generations (ITNG), 2011 Eighth International Conference on. pp. 268–273 (April 2011)
2. Aho, P., Suarez, M., Kanstren, T., Memon, A.: Murphy tools: Utilizing extracted gui models for industrial software testing. In: The Proceedings of the Testing: Academic & Industrial Conference (TAIC-PART). IEEE Computer Society (2014)
3. Alegroth, E., Nass, M., Olsson, H.: Jautomate: A tool for system- and acceptance-test automation. In: Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on. pp. 439–446 (March 2013)
4. Amalfitano, D., Fasolino, A., Tramontana, P.: A gui crawling-based technique for android mobile application testing. In: Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on. pp. 252–261 (March 2011)
5. Banerjee, I., Nguyen, B., Garousi, V., Memon, A.: Graphical user interface (gui) testing: Systematic mapping and repository. Information and Software Technology (2013)
6. Bauersfeld, S., Vos, T.E.J.: User interface level testing with TESTAR; what about more sophisticated action specification and selection? In: Proceedings of the Seventh Seminar on Advanced Techniques and Tools for Software Evolution, SATToSE 2014. pp. 60–78 (July 2014)
7. Bauersfeld, S., Wappler, S., Wegener, J.: A metaheuristic approach to test sequence generation for applications with a GUI. In: Proceedings of the Third International Symposium on Search Based Software Engineering (SSBSE 2011). pp. 10–12 (September 2011)
8. Bauersfeld, S., Vos, T.E.J.: Guitest: a java library for fully automated gui robustness testing. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. pp. 330–333. ASE 2012, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2351676.2351739
9. Bauersfeld, S., Vos, T.E.J., Condori-Fernández, N., Bagnato, A., Brosse, E.: Evaluating the TESTAR tool in an industrial case study. In: 2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2014, Torino, Italy, September 18-19, 2014. p. 4 (2014)
10. Dallmeier, V., Pohl, B., Burger, M., Mirold, M., Zeller, A.: Webmate: Web application test generation in the real world. 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops 0, 413–418 (2014)
11. Grechanik, M., Xie, Q., Fu, C.: Maintaining and evolving gui-directed test scripts. In: Proceedings of the 31st International Conference on Software Engineering. pp. 408–418. ICSE '09, IEEE Computer Society, Washington, DC, USA (2009), http://dx.doi.org/10.1109/ICSE.2009.5070540
12. I.Morgado, Paiva, A., Faria, J.: Dynamic reverse engineering of graphical user interfaces. Int. Journal on Advances in Software 5(3 and 4), 224–246 (2012)
13. Joorabchi, M., Mesbah, A.: Reverse engineering ios mobile applications. In: Reverse Engineering (WCRE), 2012 19th Working Conference on. pp. 177–186 (Oct 2012)
14. Kaner, C.: Avoiding shelfware: A managers view of automated gui testing. http://www.kaner.com/pdfs/shelfwar.pdf (2002)
15. Leotta, M., Clerissi, D., Ricca, F., Spadaro, C.: Comparing the maintainability of selenium webdriver test suites employing different locators: a case study. In:

Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to testing Automation. pp. 53–58. JAMAICA 2013, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2489280.2489284
16. Marchetto, A., Tonella, P.: Using search-based algorithms for ajax event sequence generation during testing. Empirical Software Engineering 16(1), 103–140 (2011), http://dx.doi.org/10.1007/s10664-010-9149-1
17. Mariani, L., Pezzè, M., Riganelli, O., Santoro, M.: Autoblacktest: A tool for automatic black-box testing. In: Proceedings of the 33rd International Conference on Software Engineering. pp. 1013–1015. ICSE '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1985793.1985979
18. Memon, A., Banerjee, I., Nguyen, B., Robbins, B.: 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)
19. Mesbah, A., van Deursen, A., Lenselink, S.: Crawling AJAX-based web applications through dynamic analysis of user interface state changes. ACM Transactions on the Web 6(1), 1–30 (2012)
20. Mesbah, A., van Deursen, A.: Invariant-based automatic testing of ajax user interfaces. In: Proceedings of the 31st International Conference on Software Engineering. pp. 210–220. ICSE '09, IEEE Computer Society, Washington, DC, USA (2009), http://dx.doi.org/10.1109/ICSE.2009.5070522
21. Mesbah, A., van Deursen, A., Lenselink, S.: Crawling ajax-based web applications through dynamic analysis of user interface state changes. ACM Trans. Web 6(1), 3:1–3:30 (Mar 2012), http://doi.acm.org/10.1145/2109205.2109208
22. Miao, Y., Yang, X.: An fsm based gui test automation model. In: Control Automation Robotics Vision (ICARCV), 2010 11th International Conference on. pp. 120–126 (Dec 2010)
23. Nguyen, B.N., Robbins, B., Banerjee, I., Memon, A.: Guitar: an innovative tool for automated testing of gui-driven software. Automated Software Engineering pp. 1–41 (2013)
24. Nguyen, B.N., Robbins, B., Banerjee, I., Memon, A.M.: Guitar: an innovative tool for automated testing of gui-driven software. Autom. Softw. Eng. 21(1), 65–105 (2014)
25. Rueda, U., Vos, T.E.J., Almenar, F., Martínez, M.O., Esparcia-Alcázar, A.I.: TESTAR: from academic prototype towards an industry-ready tool for automated testing at the user interface level. In: Canos, J.H., Gonzalez Harbour, M. (eds.) Actas de las XX Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2015). pp. 236–245 (2015)
26. Singhera, Z.U., Horowitz, E., Shah, A.A.: A graphical user interface (gui) testing methodology. IJITWE 3(2), 1–18 (2008)
27. Sun, Y., Jones, E.L.: Specification-driven automated testing of GUI-based Java programs. In: Proceedings of the 42nd annual Southeast regional conference. pp. 140–145. ACM (2004)
28. Yang, W., Prasad, M.R., Xie, T.: A grey-box approach for automated gui-model generation of mobile applications. In: Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering. pp. 250–265. FASE'13, Springer-Verlag, Berlin, Heidelberg (2013)
29. Yeh, T., Chang, T.H., Miller, R.C.: Sikuli: Using gui screenshots for search and automation. In: Proceedings of the 22Nd Annual ACM Symposium on User Interface Software and Technology. pp. 183–192. UIST '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1622176.1622213