Evaluating TESTAR's effectiveness through code coverage

Aaron van der Brugge[1], Fernando Pastor Ricós[2], Pekka Aho[1], Beatriz Marín[2], and Tanja E.J. Vos[1,2]

[1] Open Universiteit, The Netherlands
[2] Universitat Politècnica de València, Spain

Abstract. Testing is of paramount importance in assuring the quality of software products. Nevertheless, it is not easy to judge which techniques or tools are the most effective. A commonly used surrogate metric to evaluate the effectiveness of testing tools is code coverage, which has been widely used for unit and integration testing. However, for GUI testing approaches, this metric has not been sufficiently investigated. To fill this gap, we run experiments with the TESTAR tool, a scriptless testing tool that automatically generates test cases at the Graphical User Interface (GUI) level. In the experiment, we analyze and compare the obtained code coverage when using four different action selection mechanisms (ASMs) in TESTAR that are used to test three SUTs.

Keywords: Effectiveness · Code Coverage · Experiment · GUI Testing.

1 Introduction

Nowadays there are a lot of methods with different activities, roles and artifacts to develop software. In all of them, the testing phase is of paramount importance for quality assurance. The best way to evaluate the quality of testing would be to use the percentage of failures that were found executing the tests cases. However, it is not possible to know in advance how many bugs exist in the software, and if we don't find any, this doesn't mean that they don't exist. Consequently, in order to analyse the effectiveness of testing, surrogate measures [8] are used. A commonly used surrogate measure is code coverage [2]. Code coverage has been widely used for measuring the quality of unit and integration testing, however for system testing approaches that test at the GUI level, this metric have been less investigated.

We want to fill this gap by evaluating the coverage of a GUI testing tool in an experiment. We select the TESTAR tool [25] since it is an open-source tool that allows to automatically generate the test cases from the GUI. To do that, TESTAR implements a scriptless approach, meaning that the test cases do not have to be defined prior to test execution. Instead, each test step is generated during the test execution, based on (1) the actions that are available in that specific time and state of the GUI, and (2) the action selection mechanism (ASM). Selecting and executing the most suitable actions can both improve the likelihood and

decrease the time required for finding failures. Thus, depending on the ASM that the tool uses, different coverage measures can be obtained.

In the experiment, we aim to compare and gain knowledge of the effectiveness of various ASMs implemented in the open source TESTAR tool. Moreover, in this experiment we evaluate the effectiveness of two ASMs that have not been evaluated previously: prioritize new actions and unvisited actions. The contribution of our experiment is interesting for researchers and practitioners who are concerned in implementing more effective ASMs in the tools they are researching/developing or using in their projects.

The rest of the paper is organized as follows. Section 2 presents some relevant related work and the main characteristics of TESTAR. Section 3 presents the experiment set-up and Section 4 presents the results obtained. Finally, Section 5 presents our conclusions and future work.

2 Related work

Most publications that investigate coverage criteria for GUI testing are based on models like event-flow graphs, event-interaction graphs and state machines. In those cases, the coverage criteria are defined based on the models.

In [16] a hierarchical relationship is defined between the components of a GUI that is represented by an integration tree. The inter-component coverage criteria are used to evaluate the adequacy of test sequences that cross components. The event-flow graphs and integration tree for a given GUI, are used to evaluate the coverage of a given test suite with respect to these new coverage criteria.

A genetic algorithm based coverage approach has been published in [20]. A genetic algorithm searches for the optimal test parameter combinations that satisfy a pre-defined test criterion that covers an event model.

With respect to code coverage testing, there are various surveys presented in [27], [23], [22], and [21]; but very little is written about measuring code coverage in GUI testing.

In [24], a comparison of the effectiveness of model-based GUI testing tools for Android applications is presented. Authors introduced a stochastic model-based testing tool - called Stoat - and evaluated its effectiveness by comparing different code coverage measures with regard to A3E [5], and Sapienz [12]. This work differs from our work since it is specific for Android applications without focus in the ASMs. In our study we are centered in the evaluation of the different action selection mechanisms that can be used by an automated GUI testing tool.

Another empirical study of automated GUI testing tools is presented in [19]. This work evaluates two ASMs of TESTAR(Random and Q-learning) with GUITAR[17], AutoBlacktest [13] and Augusto [14]. As far as we know, this is the only study that has measured code coverage for TESTAR. In this work, we want to provide more evidence of the coverage of a GUI testing tool (TESTAR) by taking into account 4 different ASMs for 3 SUTs. In addition to the previously evaluated ASMs, Random and Q-learning, we will introduce 2 new ASMs: prioritize new actions and unvisited actions.

Of the industrial studies that have evaluated the effectiveness of TESTAR [6] [7] [15] [9] [3] [18], only one [7] has included the measurement for code coverage of PHP code. All the other studies concentrated on fault-finding capabilities and other functional coverage criteria.

2.1 TESTAR

TESTAR is an open source tool that carries out automated testing without the need of scripts, falling into the category of smart monkey testing tools. It implements a scriptless approach, meaning that the test cases do not have to be defined prior to test execution. Instead, each test step is generated during the test execution, based on the actions that are available in that specific time and the state of the GUI.

The underlying principle of TESTAR is simple: generate test sequences of (state, action)- pairs by starting up the SUT in its initial state and continuously selecting an action to bring the SUT in another state. The action selection characterizes the most basic problem of intelligent systems: what to do next.

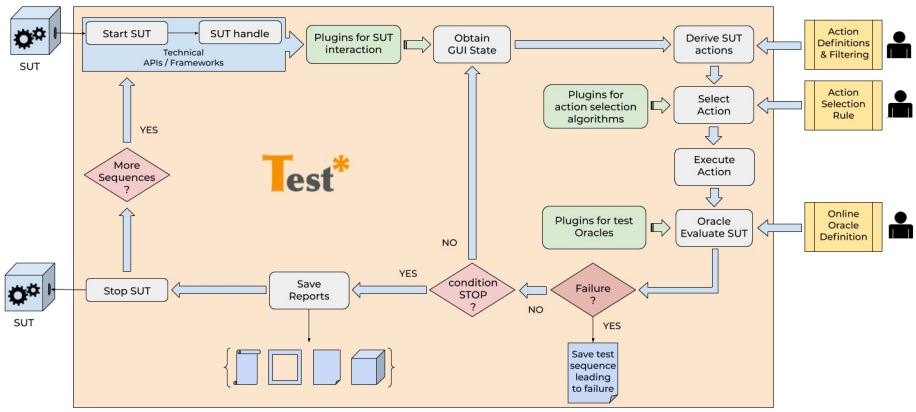


Fig. 1: High level logical flow of TESTAR.

A high level illustration of TESTARs logical flow is shown in Fig. 1. Everything inside the large square is automated. The three activities on the right side of the square represent where the users can improve the flow by configuring or extending TESTAR if desired. After starting up the SUT, the tool goes into the loop of continuously selecting and executing an action to bring the SUT from one state to another state, until some stopping criteria has been met, after which the SUT is closed.

In order to specify which Action Selection Mechanism (ASM) will be used during the testing process, specific protocols are defined. TESTAR distribution 2.2.14 supports the following ASMs:

– **Random** is based on random selection of the next action in a particular state of the system.

- **Q-learning** uses a reinforcement learning algorithm described in [11, 10] using the best values obtained in the results of the experiments $R_{max} = 9999999$ and $\gamma = 0.95$. This ASM requires state abstraction, because it requires some kind of a state model and has to be able to compare the state-action identifiers to map the Q-learning values of state-action pairs.
- **Prioritize new actions** prioritizes new actions by comparing the available actions in the current state to the available actions in the previous state. This ASM does not require state abstraction because it only uses the available actions and their description property to compare the actions.
- **Unvisited actions** infers and uses a state model that is stored in an OrientDB database. This ASM evidently also requires state abstraction to identify states. It will look at the current state in the abstract state model for actions that have not yet been visited by TESTAR. From this collection of unvisited actions, it returns a random one. If there are no unvisited actions left in the current state, it uses the model to obtain a sequence of actions that leads to a state with unvisited actions.

3 Research method

We want to investigate the effectiveness of TESTAR using different ASMs. Code coverage is a metric used to determine how much code of a program has been exercised during the active period of a program. In our study, we can use instruction coverage and branch coverage to determine how much of the code of a System Under Test (SUT) has been executed by TESTAR using different ASMs. Thus, we have formulated the following research question:

> RQ1. Which of the action selection mechanisms of TESTAR are the most effective, measured by code coverage?

To answer the research question, we design an experiment following the guidelines presented in [26, 4]. The goal of the experiment is to analyse four different ASMs of TESTAR for the purpose of evaluating their effectiveness measured by code coverage from the point of view of the researchers.

To make statistical analysis possible, we formulate the following null hypothesis that is based on the research question.

> H0 : The evaluated action selection mechanisms of TESTAR do not show a difference in the reached code coverage.

3.1 Objects: Selection of SUTs

The SUTs that we can select for the experiment should comply with the following: 1) The SUTs have a GUI; 2) TESTAR is able to detect the widgets on the GUI of the SUTs; 3) The SUTs are written in Java and JaCoCo [1] can be used to measure the code coverage reached during testing of the SUT; 4) The SUTs must be published under open source license.

Information on the selected SUTs			
Metric	Spaghetti	SwingSet2	Rachota
Packages	1	1	3
Java Classes	1	31	52
Methods	45	290	934
LLOC	350	7029	2722
Application	Example	Swing Demo	Business
Java Swing	Yes	Yes	Yes
Classes incl. Inner classes	14	138	327

Table 1: Details on the three SUTs used for the experiment

We selected the following SUTs (see Table 1). **Spaghetti** is a simple SUT that consists of only one jar especially made for this experiment. The name was given since the SUT is a jumble of code and graphics, and had no further practical use. **SwingSet2** [3] is a Java application for demonstrating the Swing features, a set of building blocks for creating cross platform GUIs. **Rachota**[4] is a Java application for time tracking different projects. It displays time data in a diagram form, creates customized reports and invoices or analyses measured data and suggests hints to improve user's time usage. It was also used for evaluation in [19].

We prepared SUT-specific TESTAR configuration for each SUT (see Table 2), which could be counted as an independent variable, as the same configuration was used for all ASMs.

Spaghetti	- Top widgets	
	- (Add) Force Tree, ComboBox, List	
Swingset2	- Top widgets	
	- (Filter) isSourceCodeEditWidget	
	- (Add) forceWidgetTreeClickAction	
	- (Add) forceListElemetsClickAction	
Rachotta	- Top widgets	- (Add) isSpinBoxWidget
	- (Trigger) addFilenameReportAction	- (Filter) isDurationTableCell
	- (Trigger) forcePricePerHourAndFinish	- (Add) forceWidgetTreeClickAction
	- (Add) isEditToClickWidget	- (Add) forceListElemetsClickAction
	- (Filter) isEditableWidget	- (Filter) "Cancel" inside Report wizard
	- (Add) isCalendarTextWidget	

Table 2: SUT-specific TESTAR configuration

3.2 Variables

We use the following independent variables for the experiment, which are constant values and factors:

- The TESTAR action selection mechanisms (factor): Random, Q-learning, Prioritize new actions and Unvisited actions, with their corresponding TESTAR protocols to control them.

[3] https://github.com/openjdk/jdk/tree/master/src/demo/share/jfc/SwingSet2
[4] http://rachota.sourceforge.net/en/index.html

– The test.settings file:
  1. Suspicious widget oracles: none (blocking)
  2. State abstraction for Q-learning and Unvisited actions (see Table 3). State abstraction does not affect Random or Prioritize new actions.
  3. Minimum waiting time between executed actions: 0.3 seconds
  4. Action duration: 0.3 seconds
  5. SUT max startup time: 60 seconds
  6. Each test run consists of 10 sequences and 300 actions per sequence.

	Spaghetti and Swingset2	Rachotta
Q-learning	Path, ControlType, Title	Path, ControlType
Unvisited actions	Path, ControlType, Title	Path, ControlType

Table 3: State abstraction for state dependent ASMs

We use the following dependent variables to validate our hypothesis and answer our research question:

– Timestamp after each action, corresponding to the time elapsed since the start of the test sequence. This is measured after each action.
– Instruction coverage after each action, as measured by JaCoCo metric, i.e. the number of instructions executed, expressed as a percentage of the total number of instructions contained in the code of the SUT. At this point it is important to mention that instruction coverage is based on bytecode instead of the line coverage commonly used by developers.
– Branch coverage after each action, as measured by JaCoCo metric, i.e. the number of branches executed, expressed as a percentage of the total number of branches contained in the code of the SUT.
– Accumulated instruction coverage. The coverage values of the 10 test sequences are accumulated during the test run to get a final value of the total code covered during each test run.
– Accumulated branch coverage. The coverage values of the 10 test sequences are accumulated during the test run to get a final value of the total code covered during each test run.

3.3 Design of the experiment

Fig. 2 presents the design of the experiment. All three SUTs will be tested with the four ASMs. Coverage will be measured after every action. The results will be used to determine the point at which sufficient code coverage is achieved.

In this experiment we will use the general design principle of blocking. This means we will block the testing capacity on errors and exceptions of TESTAR, which could otherwise interrupt the test-runs when discovering such faults. So no oracles (that define errors and exceptions TESTAR is checking on) will be used during this experiment because finding an error or exception will interrupt the test sequence, and the goal of this coverage experiment is not to find possible programmers' mistakes and such.

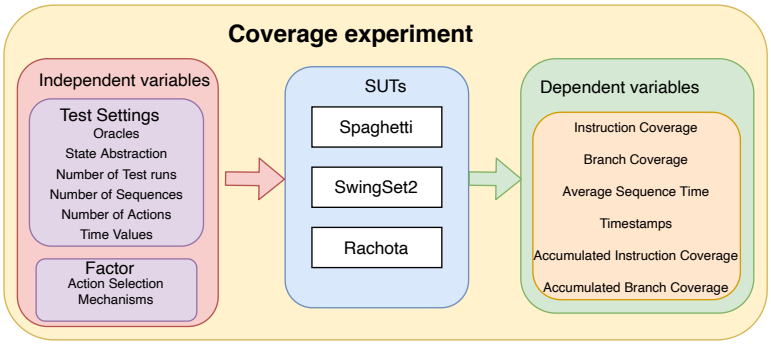


Fig. 2: Design of the experiment

By keeping the numbers of the most independent variables (except the factor) equal during the experiment, we create a balanced design that keeps the relation between cause and effect clear. For this reason, in this coverage experiment, we keep the number of actions equal. It seems plausible that more actions during a test run make it possible to achieve at least the same amount of coverage during the same test run than fewer actions. So, it is clear that a sufficient number of actions must be chosen in order to properly compare the measures obtained. Thus, we choose a test set-up of 10 sequences of 300 actions for each test run.

To be able to draw valid conclusions about the possible rejection of the null hypothesis, and to deal with the randomness of TESTAR ASMs, we will repeat all test runs of our experiment 30 times using concurrent Virtual Machines (VMs) running tests (see Fig. 3). Each combination of SUT and ASM will produce two samples. One consisting of the final instruction coverage values and another one consisting of final branch coverage, both reached after 10 sequences of 300 actions. Since each combination will have 30 test runs, each sample will have 30 values. We apply Mann-Whitney U-Test to compare two samples with each other, looking at whether the values of the data of one sample are higher or lower than that of the other sample.

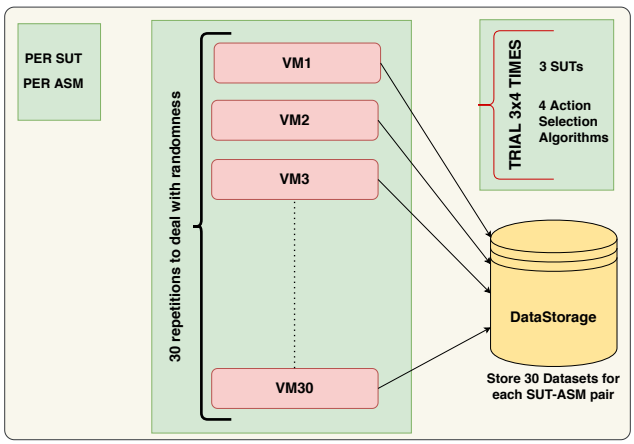


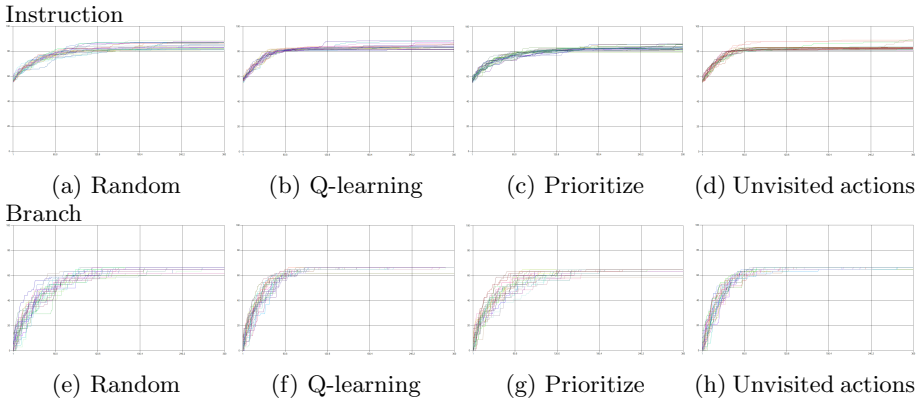
Fig. 3: Overall design of the experiment

4 Results

The results of the coverage experiment are presented in this section. The accumulated instruction and branch coverage percentage are presented in curves for instruction and branch coverage for each SUT. Mann-Whitney U p-values for instruction and branch coverage were carried out to discover the differences between the samples.

4.1 Analysing Spaghetti

Regarding instruction coverage, after already 120 actions the tests for all ASMs on Spaghetti reach a relatively high percentage of accumulated instruction coverage (see Fig. 4). Note that in Fig. 4, the number of actions is presented on axis X and the merged instruction/branch coverage per test run is presented on axis y. We do not present the mean value for each ASM for instruction and branch coverage due to randomized algorithms can yield a high variance probability distributions, so using the mean value can be misleading, as it is stated in [4].



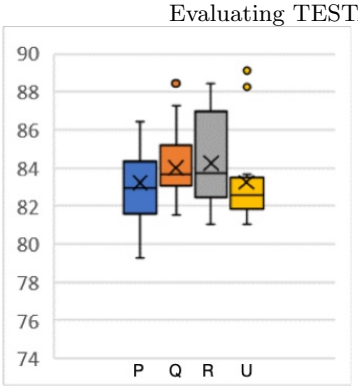
(a) Random (b) Q-learning (c) Prioritize (d) Unvisited actions

(e) Random (f) Q-learning (g) Prioritize (h) Unvisited actions

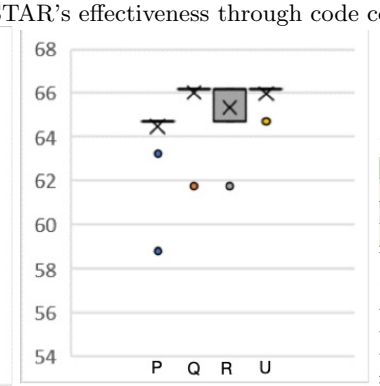
Fig. 4: Spaghetti: curves showing how coverage grows with every executed action

The results show a statistically significant difference between **Q-learning** and **Unvisited actions** (see Fig. 5c). Despite of this difference, there is a similarity in the course of the curves for both action selection mechanisms. Both start with a similarly steep curve. This is because there are still relatively many unvisited actions in the beginning, on which both protocols work in a similar way, favoring **Unvisited actions**.

The box plots for instruction coverage show the variation between the different ASMs (Fig. 5a). The **Random** ASM seems to perform the best on the first 300 actions. That makes sense if you consider the reached coverage. **Prioritize new actions** is not the worst ASM in terms of growth, but the worst in getting the highest instruction coverage value. **Unvisited actions** seems to only have the fastest growth at the first 100 actions, after which it is caught up by the other ASMs.



(a) Instruction



(b) Branch

Instruction

	R	Q	U
P	.103	.096	.604
R		.959	.069
Q			.010

Branch

	R	*Q*	*U*
P	.001	.001	.001
R		.001	.009
Q			.190

(c) p values

Fig. 5: Spaghetti values for 300 actions

Regarding the branch coverage, the growth of coverage over the first 300 actions of the four ASMs has similar behavior (Fig. 4). Also almost all ASMs have a significant difference for merged branch coverage on the first 300 actions, except for **Q-learning** compared to **Unvisited actions**, which have almost the same mean for both ASMs as it is shown in (Fig. 5b).

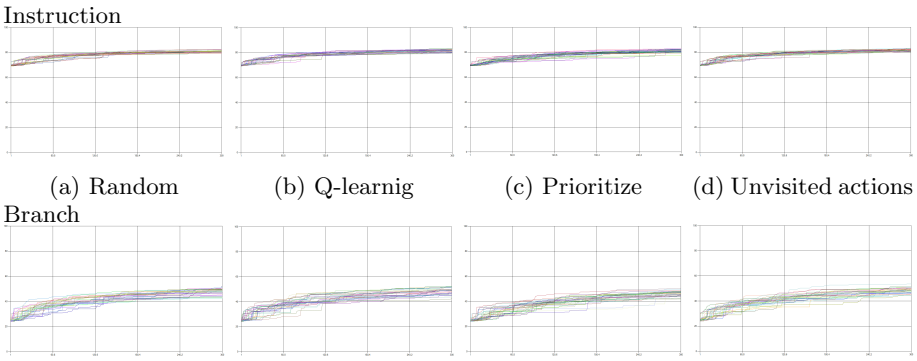
4.2 Analysing SwingSet2

For SwingSet2, after already 60 actions the tests for all ASMs reach a relatively high percentage of accumulated instruction coverage (see Fig. 6). We observe that there are differences in the steepness of the graphs on the first hundred actions, which indicates a difference in growth during that range. **Unvisited actions** and **Q-learning** seem to grow faster, while **Prioritize new actions** and **Random** have both a similar less steep curve. The **Unvisited actions** clearly performs best on the first 300 actions (see Fig. 7a). It outperforms **Prioritize new actions** and **Random** (Fig. 7c) with statistical significance.

The box plots in Fig. 7b show that **Unvisited actions**, when compared with the other ASMs, has a better performance during the first 300 actions. The p values show no improvement in differences between the branch coverage of ASMs. On the contrary, the tests over **Q-learning** and **Unvisited actions** are no longer significant.

4.3 Analysing Rachota

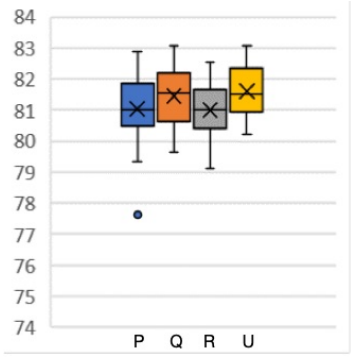
Regarding the instruction coverage, we observe that there is diverse behavior between the individual test runs. After the first 300 actions, we see the growth of the coverage is still rising at action 300 (Fig. 8). From Fig. 9a) we can observe that **Q-learning** is the most converged ASM. It also has achieved the highest coverage over the first 300 actions. **Random**, on the other hand, has the lowest coverage and the most dispersed values of coverage. This difference between these two ASMs is confirmed in Fig. 9c instruction p values. **Random** also has lower



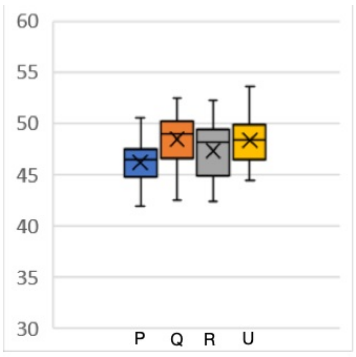
(a) Random (b) Q-learnig (c) Prioritize (d) Unvisited actions

(e) Random (f) Q-learnig (g) Prioritize (h) Unvisited actions

Fig. 6: SwingSet2 coverage curves



(a) Instruction



(b) Branch

Instruction

	R	Q	U
P	.790	.167	.033
R		.099	.009
Q			.515

Branch

	R	*Q*	*U*
P	.048	.001	.001
R		.126	.211
Q			.579

(c) p values

Fig. 7: Swingset values at 300 actions

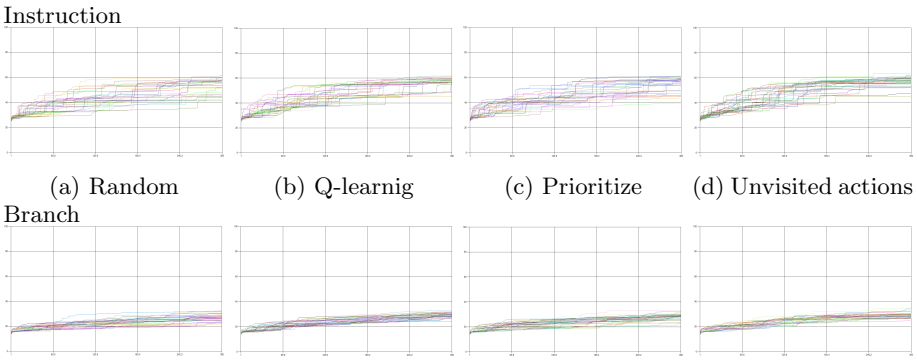
performance than **Unvisited actions** over the first 300 actions with statistical significance.

Regarding the branch coverage, **Unvisited actions** seems to be one of the highest scoring ASMs over the first 300 actions, both in terms of reached coverage and converged test run values. Similar to instruction coverage, **Unvisited actions** and **Q-learning** are the best performing ASMs on the field of branch coverage over the first 300 actions, when used with Rachota (see Fig. 9b).

4.4 Answering the research question

Regarding the null hypothesis, *H0 : The evaluated action selection mechanisms of* TESTAR *do not show a difference in the reached code coverage*, we can reject it and state that there is a difference in the reached code coverage of the ASMs.

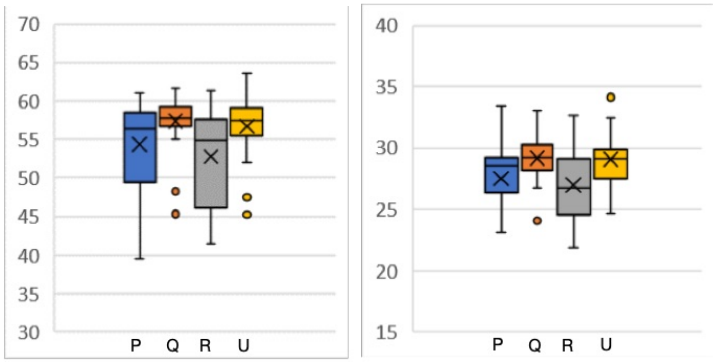
Regarding our research question *RQ1. Which of the action selection mechanisms of* TESTAR *are most effective, measured by code coverage?*, we observe that



(a) Random (b) Q-learnig (c) Prioritize (d) Unvisited actions

(e) Random (f) Q-learnig (g) Prioritize (h) Unvisited actions

Fig. 8: Rachota coverage curves



Instruction

	R	Q	U
P	.196	.044	.181
R		.001	.006
Q			.297

Branch

	R	*Q*	*U*
P	.102	.054	.128
R		.001	.004
Q			.297

(a) Instruction (b) Branch (c) p values

Fig. 9: Rachota values at 300 actions

Random is the best scoring ASM with Spaghetti. It's on the third place when used on SwingSet2. But it clearly performs worst with Rachota. **Q-learning** and **Unvisited actions**, on the other hand, exhibit the opposite behavior. They score best on Rachota, but **Unvisited actions** in particular scores the lowest on Spaghetti. From this, it could be concluded that **Random** scores best on small programs that are not very complex, but especially **Q-learning** scores best on programs that are relatively extensive. **Unvisited actions** seems to score best in term of speed, quickly increasing the coverage at the first actions.

Looking at the size and the complexity of the code, it seems that differences in growth and reached coverage better show up with the larger SUTs. Consequently, the most significant results are from Rachota that is the largest and most complex of the selected SUTs. Considering the statistical significance, the results from the coverage experiment are quite good with Rachota.

Considering the whole experiment, **Q-learning** seems to be the best performing ASM for both instruction coverage and branch coverage. However, over

the first 100 actions, **Unvisited actions** shows the fastest growth overall. This is especially the case with the tests on Spaghetti.

Summarizing, we observe that effectiveness of some ASMs appear to be better on larger applications and others work on smaller applications. We can therefore conclude that a measurement of effectiveness with code coverage is possible, provided that the properties of the SUT are taken into account. Moreover, we plan to run a follow-up experiment with a wider set SUTs to gain more evidence on the effectiveness of ASMs of TESTAR.

4.5 Threats to validity

We follow the guidelines of [26] to analyze some threats that could affect the validity of our results. Regarding to *construct validity*, JaCoCo will be used to measure code coverage data. We hence rely on the JaCoCo definitions of instruction and branch coverage and their accurate measurements.

Conclusion validity Code coverage as measured by JaCoCo is used as a surrogate measure to draw conclusions about the effectiveness of the randomized testing approach TESTAR. Since we cannot assume normal distribution of randomized testing approaches [4] we have applied Mann-Whitney U statistical tests to the 30 samples of code coverage measures.

External validity is concerned with generalization. The SUTs used in this study are small and only one application (Rachota) is a real application and not a demo or toy project. The threats to external validity should be reduced in future work by adding more complex and bigger SUTs.

5 Conclusions

This paper presents an empirical measurement and comparison of code coverage of 4 ASMs of TESTAR on 3 different SUTs. 2 ASMs of the experiment, unvisited actions and prioritize new actions, have not been evaluated with any metrics before. Even though the results of our experiment are inconclusive in terms of the best ASM for TESTAR, our results clearly show that the ASM affects the reached code coverage, so that we provide some helpful insights by showing the impact that the SUT's characteristics can have on different ASMs.

We observe that some ASMs perform better on some SUTs but worse on other SUTs. State abstraction and model inference have a strong impact on the behavior of unvisited actions and Q-learning algorithms. If the state abstraction is too concrete, TESTAR keeps finding new states and ASM keeps using the default values (random). If the state abstraction is too abstract, states with a lot of difference might be considered the same and some SUT functionality might not get tested at all. In addition, the state model gets very non-deterministic. The unvisited actions ASM turns into random after all the unvisited actions have been visited. This can be seen from the results as the coverage is growing quickly in the beginning, and slowly after the threshold has been reached.

Further work is related to the experimentation on a wider variety of SUTs, such as web applications or bigger SUTs, and without blocking the capacities of TESTAR to find exceptions. Furthermore, we want to identify correlations among the characteristics of SUTs and the performance of the ASMs; and also we want to analyze the magnitude of the improvements though the analysis of effect size using different ASMs.

In addition, we plan to continue the research by improving the ASMs, for instance adding execution counter to prioritize new actions and taking advantage of the state model would probably improve its performance in early exploration of the SUT. Moreover, optimizing the state abstraction for each SUT would probably improve the performance of unvisited actions ASM and Q-learning ASM. Another interesting future research topic would be hybrid ASMs and finding suitable criteria to change from one action selection algorithm to another. Changing from unvisited actions ASM to some combinatorial action selection algorithm after the state model has been visited could be a good solution.

Acknowledgements

This research has been funded by the following projects: H2020 EU project DECODER (www.decoder-project.eu), H2020 EU project iv4XR (www.iv4xr-project.eu) and ITEA project IVVES (www.ivves.eu).

References

1. Jacoco. https://www.jacoco.org, last accessed: 12 May 2021
2. Abran, A., et al.: Swebok. Guide to the Software Eng. Body of Knowledge (2004)
3. Aho, P., Vos, T.E.J., Ahonen, S., Piirainen, T., Moilanen, P., Pastor Ricos, F.: Continuous piloting of an open source test automation tool in an industrial environment. In: Jornadas de Ingeniería del Software y Bases de Datos (JISBD). pp. 1–4. Sistedes (2019)
4. Arcuri, A., Briand, L.: A practical guide for using statistical tests to assess randomized algorithms in software engineering. ACM, New York, NY, USA (2011)
5. Azim, T., Neamtiu, I.: Targeted and depth-first exploration for systematic testing of android apps. In: ACM SIGPLAN int. conference on Object oriented programming systems languages & applications. pp. 641–660 (2013)
6. Bauersfeld, S., de Rojas, A., Vos, T.E.J.: Evaluating rogue user testing in industry: An experience report. In: IEEE Eighth International Conference on Research Challenges in Information Science (RCIS), 2014. pp. 1–10 (May 2014)
7. Bauersfeld, S., Vos, T.E.J., Condori-Fernández, N., Bagnato, A., Brosse, E.: Evaluating the TESTAR tool in an industrial case study. In: ACM-IEEE International Symposium on Empirical Software Eng. and Measurement, ESEM. p. 4 (2014)
8. Briand, L.C.: A critical analysis of empirical research in software testing. In: 1st Int. Symposium on Empirical Soft. Eng. and Measurement (ESEM). pp. 1–8 (2007)
9. Chahim, H., Duran, M., Vos, T.E.J., Aho, P., Condori Fernandez, N.: Scriptless testing at the gui level in an industrial setting. In: Research Challenges in Information Science. pp. 267–284. Springer (2020)
10. Esparcia-Alcázar, A., Almenar, F., Martínez, M., Rueda, U., Vos, T.: Q-learning strategies for action selection in the testar automated testing tool. 6th Int. Conf. on Metaheuristics and nature inspired computing (META) pp. 130–137 (2016)

11. Esparcia-Alcázar, A., Almenar, F., Vos, T., Rueda, U.: Using genetic programming to evolve action selection rules in traversal-based automated software testing: results obtained with the TESTAR tool. Memetic Comput. **10**(3), 257–265 (2018)
12. Mao, K., Harman, M., Jia, Y.: Sapienz: Multi-objective automated testing for android applications. In: 25th Int. Symp. on Software Testing and Analysis. pp. 94–105 (2016)
13. Mariani, L., Pezze, M., Riganelli, O., Santoro, M.: Autoblacktest: Automatic black-box testing of interactive applications. In: Fifth International Conference on Software Testing, Verification and Validation. pp. 81–90. IEEE (2012)
14. Mariani, L., Pezzè, M., Zuddas, D.: Augusto: Exploiting popular functionalities for the generation of semantic gui tests with oracles. In: 40th International Conference on Software Engineering. pp. 280–290 (2018)
15. Martinez, M., Esparcia, A.I., Rueda, U., Vos, T.E.J., Ortega, C.: Automated localisation testing in industry with testar. In: Wotawa, F., Nica, M., Kushik, N. (eds.) Testing Software and Systems. pp. 241–248. Springer, Cham (2016)
16. Memon, A.M., Soffa, M.L., Pollack, M.E.: Coverage criteria for gui testing. In: 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering. pp. 256–267 (2001)
17. Nguyen, B.N., Robbins, B., Banerjee, I., Memon, A.: Guitar: an innovative tool for automated testing of gui-driven software. Automated software engineering **21**(1), 65–105 (2014)
18. Pastor Ricós, F., Aho, P., Vos, T., Torres Boigues, I., Calás Blasco, E., Martínez, H.: Deploying testar to enable remote testing in an industrial ci pipeline: A case-based evaluation. In: Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles. pp. 543–557. Springer (2020)
19. Pezzè, M., Rondena, P., Zuddas, D.: Automatic gui testing of desktop applications: an empirical assessment of the state of the art. In: Companion Proceedings for the ISSTA/ECOOP 2018 Workshops. pp. 54–62 (2018)
20. Rauf, A., Anwar, S., Jaffer, M.A., Shahid, A.A.: Automated gui test coverage analysis using ga. In: 7th International Conference on Information Technology: New Generations. pp. 1057–1062. IEEE (2010)
21. Shahid, M., Ibrahim, S., Mahrin, M.N.: A study on test coverage in software testing. Advanced Informatics School (AIS), Universiti Teknologi Malaysia, International Campus, Jalan Semarak, Kuala Lumpur, Malaysia (2011)
22. Sharma, S., Chandra, U., Jain, P.: A literature survey on automation of test data generation for branch coverage testing using genetic algorithm. International Journal of Computational Intelligence Research **13**(6), 1521–1531 (2017)
23. Smith, B., Williams, L.A.: A survey on code coverage as a stopping criterion for unit testing. Tech. rep., North Carolina State Univ. Dept. of Comp. Science (2008)
24. Su, T., Meng, G., Chen, Y., Wu, K., Yang, W., Yao, Y., Pu, G., Liu, Y., Su, Z.: Guided, stochastic model-based gui testing of android apps. In: 11th Joint Meeting on Foundations of Software Engineering. pp. 245–256 (2017)
25. Vos, T.E.J., Aho, P., Pastor Ricos, F., Rodriguez-Valdes, O., Mulders, A.: testar – scriptless testing through graphical user interface. Software Testing, Verification and Reliability **31**(3), e1771 (2021)
26. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in software engineering. Springer Science & Business Media (2012)
27. Yang, Q., Li, J.J., Weiss, D.M.: A survey of coverage-based testing tools. The Computer Journal **52**(5), 589–597 (2009)