



Using genetic programming to evolve action selection rules in traversal-based automated software testing: results obtained with the TESTAR tool

Anna I. Esparcia-Alcázar^{1,2} · Francisco Almenar^{2,4} · Tanja E. J. Vos^{2,3} · Urko Rueda²

Received: 10 September 2017 / Accepted: 24 May 2018 / Published online: 9 June 2018
© Springer-Verlag GmbH Germany, part of Springer Nature 2018

Abstract

Traversal-based automated software testing involves testing an application via its graphical user interface (GUI) and thereby taking the user's point of view and executing actions in a human-like manner. These actions are decided on the fly, as the software under test (SUT) is being run, as opposed to being set up in the form of a sequence prior to the testing, a sequence that is then used to exercise the SUT. In practice, random choice is commonly used to decide which action to execute at each state (a procedure commonly referred to as monkey testing), but a number of alternative mechanisms have also been proposed in the literature. Here we propose using genetic programming (GP) to evolve such an action selection strategy, defined as a list of IF-THEN rules. Genetic programming has proved to be suited for evolving all sorts of programs, and rules in particular, provided adequate primitives (functions and terminals) are defined. These primitives must aim to extract the most relevant information from the SUT and the dynamics of the testing process. We introduce a number of such primitives suited to the problem at hand and evaluate their usefulness based on various metrics. We carry out experiments and compare the results with those obtained by random selection and also by Q-learning, a reinforcement learning technique. Three applications are used as Software Under Test (SUT) in the experiments. The analysis shows the potential of GP to evolve action selection strategies.

Keywords Automated software testing via the GUI · Genetic programming · Action selection for testing · Testing metrics

1 Introduction

Testing a software application at the Graphical User Interface (GUI) level is an important step when ensuring software quality, mainly because it implies taking the user's perspective and is thus the ultimate way of verifying a program's

correct behaviour. GUIs can account for 45–60% of the source code [2] in any application and are often large and complex. Automating the process of testing via the GUI is therefore a very relevant issue in order to minimise time-consuming and tedious manual testing, a task hindered by the fact that GUIs are designed to be operated by humans, not machines.

Traversal-based automated software testing is a technique for automatically testing applications by means of their GUI [1]. Its name derives from the fact that the GUI is *traversed* using information obtained from it (GUI reflection), which can be used to check some general properties. This approach differs from other approaches to testing via the GUI that can be found in the literature, such as *capture-and-replay* (C&R), which involves recording user interactions and converting them into a script that can be replayed repeatedly, and *visual-based* testing, which relies on image recognition techniques to visually interpret the images of the target UI [3]. Of the three approaches, traversal-based testing is the closest to the user's perspective and is also the one that copes

✉ Anna I. Esparcia-Alcázar
aesparcia@ieee.org

Francisco Almenar
falmenar.etraid@grupoetra.com

Tanja E. J. Vos
tvos@testar.org

¹ Department of Systems Engineering and Control, Universitat Politècnica de València, Valencia, Spain

² Software Production Methods Research Centre, Universitat Politècnica de València, Valencia, Spain

³ Open Universiteit, Heerlen, The Netherlands

⁴ Etra I+D, Valencia, Spain

better with the dynamic nature of applications, which can be subject to frequent changes due to updates in functionality, improvements in their usability, or modifications in the requirements.

Traversal-based testing tools often resort to random choice in order to decide what action to execute given the current state (or window) the system is in, a procedure known as *monkey testing*. Attempts to add intelligence to the action selection process have involved using metaheuristics or machine learning techniques, such as Q-learning [11] and Ant Colony Optimisation [8]. Here we extend the findings of [12], where we used genetic programming to evolve action selection rules for the traversal-based software testing tool TESTAR.¹ GP has previously been used in software testing, e.g. by [19,21] and more recently by [14] but these works focused on evolving test cases for object oriented software; they neither deal with testing via the GUI, nor with evolving an action selection mechanism.

In [12] we use GP to evolve a population of rules whose quality (or *fitness*) is evaluated by using each new rule as the action selection mechanism in TESTAR. The fitness is based on a suitable metric of the testing efficiency. Defining such a metric (or combination of metrics) is not an easy task; for instance, the aim of testing is to find faults, but not finding them is not necessarily a proof that the testing process was adequate. Different approaches have been taken in the literature, also depending on the type of SUT. For instance, in [10] metrics are proposed for event driven software; [17] defines a coverage criteria for GUI testing, while in [20] the number of crashes of the SUT, the average time it takes to crash and the reproducibility of these crashes are used. In this work we will follow the approach taken in [11], where we proposed four metrics which are suitable for testing web applications, based on the assumption that source code is not available.

Here we extend the work done in [12] by introducing new primitives (nodes and terminals) and allowing longer, more elaborate, decision strategies, consisting of a list of IF-THEN rules. We have also adapted the crossover and mutation operators so as to better handle these kinds of rules. The aim is to ascertain which are the interesting primitives to use when evolving action selection rules with GP and also if the improvements have a positive effect on the performance.

The rest of this paper is structured as follows. Section 2 describes the TESTAR tool and the way it works; Sect. 3 describes the use of genetic programming to evolve action selection algorithms. Section 4 introduces the metrics used for quality assessment of the testing procedure. Section 5 summarises the experimental set up, the results obtained and the statistical analysis carried out; it also highlights the problems encountered. Finally, in Sect. 6 we present some conclusions and outline areas for future work.

2 TESTAR

TESTAR (or Test*) is an open source tool that performs automated testing via the GUI, using the operating system's Accessibility API to recognise GUI controls (or *widgets*) and their properties,² and enabling programmatic interaction with them. It derives sets of possible actions for each state the GUI is in and selects and executes appropriate ones, thus creating a test sequence on the fly. TESTAR has been successfully applied to various commercial and open source applications, both desktop and web-based, as shown in e.g. [5–7,18].

TESTAR performs the following steps (as shown in Fig. 1): (1) start the SUT; (2) obtain the GUI state (a widget tree); (3) derive a set of sensible actions that a user could execute in a specific SUT state (i.e. clicks, text inputs, mouse gestures); (4) select one of these actions; (5) execute the selected action; (6) apply the available oracles to check (in)validness of the new UI state. If a failure 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.

TESTAR can detect the violation of general-purpose system requirements through implicit oracles [4] like those stating that (1) the SUT should not crash, (2) the SUT should not find itself in an unresponsive state (freeze) and (3) the GUI state should not contain any widget with suspicious words like *error*, *problem*, *exception*, etc.

3 Evolution of action selection rules by genetic programming

Traditional genetic programming [15] involves the evolution of a population of individuals, or candidate solutions, that can be represented as expression trees, given suitable nodes (functions) and leaves (terminals) are defined for the problem at hand. In this work rather than represent individuals as trees, we will follow the linear GP approach initiated by [9]; our action-selection strategies will be a list of IF-THEN rules that, given the current state of the SUT, pick the next action to execute. An example rule would be something like this:

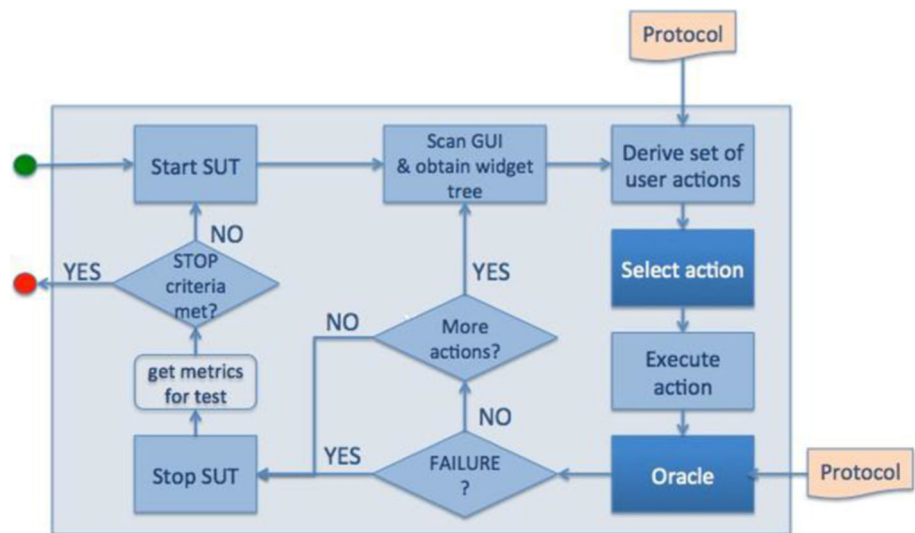
```

IF condition1
THEN action1
IF condition2
THEN action2
...
IF conditionN
THEN actionN
ELSE
  pick random action
  
```

¹ <https://testar.org>.

² E.g. display position, widget size, ancestor widgets, etc.

Fig. 1 How TESTAR works



Hence, conditions are evaluated and actions taken in consequence, and if no conditions are met a random action is chosen by default.

In [12] we gave the first steps towards GP-evolution of action-selection rules. Here we aim at finding answers to the following research questions:

1. *Is Genetic programming a good option for obtaining action-selection rules in traversal-based testing?*
2. *Which are the relevant features to take into consideration when selecting the next action to execute?*
3. *Do the evolved rules generalise to ‘unseen’ SUTs, i.e. software for which they were not evolved?*

With this aim in mind, we have run two sets of experiments, one with a simple set of functions and terminals (as described in Table 1) and fast execution times, and the other which uses a larger set (given by Tables 1 and 2) and for which both the evolution and validation are left to run for a longer time.

4 Testing performance metrics

As stated in Sect. 1, a number of metrics have been defined in the literature to assess the quality of the testing, e.g. those given by [17] or [20]. However, two main issues can be found with them: namely, that they either imply having access to the SUT source code (which is not always the case) or that they focus on errors encountered and reveal nothing about to what extent the SUT was explored (which is particularly relevant if no errors are detected). For these reasons, we decided on the following metrics, as defined by [11]:

- **Abstract states** This metric refers to the number of different states, or windows in the GUI, that are visited in

the course of an execution. An abstract state does not take into account modifications in parameters; for instance, a window containing a text box with a given text would be considered the same abstract state as the same window and text box containing a different text.

- **Longest path** This is defined as the longest sequence of non-repeated consecutive states visited.
- **Minimum and maximum coverage per state** The *state coverage* is defined as the rate of executed over total available actions in a given state/window; the metrics are the highest and lowest such values across all windows.

Longest path and maximum coverage are in a way opposed metrics, one measuring exploration and the other exploitation of the SUT. It must be noted that the metrics given above can be used to compare the efficiency of different testing methods, but not to assess the overall goodness of a method in isolation, because we do not know the global optima for each metric; for instance, we cannot know exactly how many different states there are as a consequence of the assumption of not having access to the source code.

5 Experiments and results

5.1 Experiment 1: Reduced primitives set

In this first experiment we have taken a simplified approach which involves testing three SUTs; one of them, which we will refer to as the *sandbox* SUT, or SUT_0 , will be used in the evolutionary process. For this, the fitness will be given by the Abstract States metric. Then, the best evolved rule is validated by using it to test two more SUTs, SUT_1 and SUT_2 . For this latter phase we carried out 30 runs of 500 actions each, for each SUT. In this way we can ascertain how well the

Table 1 Functions and terminals—short version (used in experiments 1 & 2)

Name	Arity (#args)	How it works	Returns
IFT	3	If 1st argument is true return 2nd	action
pickSameAs	1	Returns a random action of the specified type arg1 is an action type	action
pickUnexecuted	1	Returns a random unexecuted action of the specified type arg1 is an action type if there are none, return a random action	action
pickAnyUnexecuted	0	Returns a random unexecuted action; if there are none, return a random action	action
typeInto	0	Text field	action type
leftClick	0	Left click	action type
previousAction	0	The type of the last executed action	action type
AND, OR	2	Boolean operators	boolean
\leq , EQ	2	args are boolean Comparison operators	boolean
NOT	1	Negation arg is boolean	boolean
nActions	0	Number of actions in current state	integer
nTypeInto	0	Number of typeInto actions in current state	integer
nLeftClick	0	Number of leftClick actions in current state	integer
RND	0	Random value	double

Table 2 Functions and terminals—extended version (experiment 2 only)

Name	Arity	How it works	Returns
pickLeastExecuted	2	From the list of least executed actions of type = arg1 , return the one in position $\text{ceil}(\text{arg2} * \text{listSize})$ arg1 is an action type arg2 is a double $\in [0, 1]$	action
pickMostExecuted	2	From the list of most executed actions of type = arg1 , return the one in position $\text{ceil}(\text{arg2} * \text{listSize})$ arg1 is an action type arg2 is a double $\in [0, 1]$	action
pickDifferentFrom	1	Pick a random action of type \neq arg1	action
<	2	Less than args are numeric	boolean
RAC	0	Random action	action

Table 3 Genetic programming parameters for Experiment 1

Feature	Value
Population size	20
Max rule size	20 nodes
Functions	Pick, PickAny, PickAnyUnexecuted, AND, OR, LE, EQ, NOT
Terminals	nActions, nTypeInto, nLeftClick, previousAction, RND, typeLeftClick, typeTypeInto, Any
Evolutionary operators	Mutation (probability = 0.05) Crossover (probability = 1)
Evolutionary method	Steady state
Selection method	Tournament of size 5
Termination criterion	Generating more than 30 different states

GP-evolved rule generalises to SUTs not encountered during evolution. In the validation phase all metrics are calculated and used for comparison.

The best evolved rule was as follows:

```

IF nLeftClick LT nTypeInto
PickAny leftClick
ELSE
PickAnyUnexecuted

```

For the sake of comparison, the validation process was also carried out using random and Q-learning-based action selection. Q-learning [22] is a model-free reinforcement learning technique in which an agent, at a state s , must choose one among a set of actions A_s available at that state. By performing an action $a \in A_s$, the agent can move from state to state. Executing an action in a specific state provides the agent with a reward (a numerical score which measures the utility of executing a given action in a given state). The goal of the agent is to maximise its total reward, since it allows the algorithm to look ahead when choosing actions to execute. It does this by learning which action is optimal for each state. The action that is optimal for each state is the action that has

the highest long-term reward. The choice of the algorithm's two parameters, maximum reward, R_{max} and discount γ , will promote exploration or exploitation of the search space. In our case we chose those that had provided best results in [11]

A summary of the experimental settings is given in Table 3 and 4.

The software under test (SUT) We used three different applications in order to evaluate our action selection approach, namely PowerPoint—which will be the *sandbox* SUT—Odoo, and Testona. **PowerPoint** is a slide show presentation program part of the productivity software Microsoft Office. It is currently one of the most commonly used presentation programs available. **Odoo** is an open source Enterprise Resource Planning software consisting of several enterprise management applications; of these, we installed the mail, calendar, contacts, sales, inventory and project applications in order to test a wide number of options. **Testona** (formerly known as *Classification Tree Editor*) is a software testing tool that runs on Windows. It implements tree classification, which involves classifying the domain of the application under test and assigning tests to each of its leaves.

Figure 2 shows the experimental procedure; genetic programming is used as the evolutionary engine, that calls the testing tool TESTAR to evaluate the individuals and calculate their fitness. Once the evolutionary process ends, the best action selection rule found is evaluated with 30 runs of TESTAR in SUT_1 and SUT_2 ; the metrics obtained are subject to a statistical comparison.

Statistical analysis We run the Kruskal–Wallis non parametric test, with $\alpha = 0.05$, on the results for the three action selection mechanisms. The test shows that all the metrics have significant differences among the sets. Running pair-wise comparisons by means of the Mann–Whitney–Wilcoxon test provides the results shown in Table 5, where the bold column is the best option. It can be seen that the GP approach wins in the abstract states and longest path metrics for Odoo and comes second in Testona, where, surprisingly, random testing performs best.

5.1.1 Failure detection

One metric we have not considered in the statistical analysis is the number of failures encountered, shown in Table 6. Here we can see that in general, the GP approach finds the

Table 4 Experimental set up for Experiment 1

Set	Action selection algorithm	Parameters	Max. actions per run	Runs
Ev	GP-evolved rule	See Table 3	500	30
Qlearning	Q-learning	$R_{max} = 9999999$; $\gamma = 0.95$	500	30
RND	Random	N/A	500	30

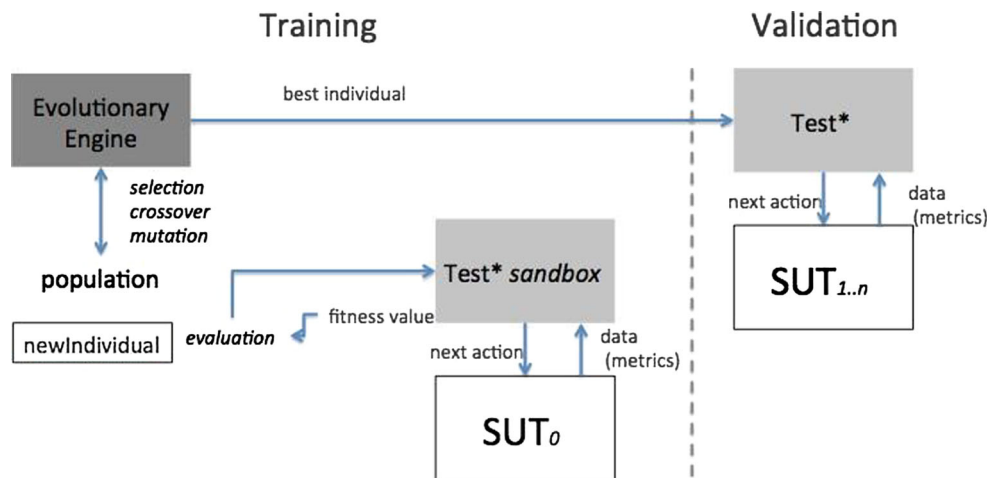


Fig. 2 The evolutionary (training) process and the subsequent validation

Table 5 Results of the statistical comparison for all algorithms and metrics, in SUT_1 and SUT_2 (Experiment 1)

Odoo	Set		
Abstract states	GP rule	Q-learning	RND
Longest path	GP rule	Q-learning	RND
Maximum coverage per state	GP rule	Q-learning	RND
Minimum coverage per state	RND	Q-learning	GP rule
Testona	Set		
Abstract states	RND	GP rule	Q-learning
Longest path	RND	GP rule	Q-learning
Maximum coverage per state	GP rule	Q-learning	RND
Minimum coverage per state	GP rule	Q-learning	RND

The bold column represents the best choice, the remaining ones are in order of preference

most real failures.³ In addition, we consider a false positive the situation where the oracle detects a failure where there is not actually one. For instance, a text message addressed to the end user that contains the words *error*, *problem* or *exception* would be flagged as a failure, even when it is not. This would be detected by the human tester. So, finding false positives can actually be interpreted as an advantage, because their detection allows human testers to improve the oracles. Finally, we term *freeze* the situation when the SUT enters an unresponsive state. For instance, in a web-based application, it can be due to a page that takes a long time to load. This is not necessarily related to a failure in the SUT.

A conclusion to this first experiment is that, even with this small set of functions and terminals, the GP-evolved action selection rule proves to be superior to random testing and also to Q-learning-based testing, even when validated in SUTs different from that with which it has been evolved, i.e. displaying a good generalisation ability. The evolved rule is

³ Note that ascertaining whether these failures are associated to any defects is beyond the scope of the *TESTAR* tool.

Table 6 Number of failures encountered per SUT and algorithm

SUT	Algorithm	Failures	Freezes	False positives
Odoo	GP rule	4	0	2
	RND	0	0	4
	Q-learning	1	1	6
Testona	GP rule	2	2	3
	RND	0	3	6
	Q-learning	1	1	3

however, very simple, and often resorts to random selection of actions, so there seems to be room for improvement with regard to the choice of nodes (functions and terminals) with which rules are built. This is the aim of the second experiment.

5.2 Experiment 2: Extended primitives set

The set up for the experiments is given in Table 7 and 8 and that of the GP runs in Table 7. As in the previous experi-

Table 7 Genetic programming parameters for Experiment 2

Feature	Value
Population size	100
Max no. of rules per strategy	14
Functions & terminals	See Tables 1 and 2
Evolutionary operators	Crossover (probability = 1) Mutation (probability = 0.05)
Evolutionary method	Steady state
Selection method	Tournament of size 11
Termination criterion	Time limit (10 hours)

Table 8 Experimental set up for Experiment 2

Action selection algorithm	Parameters	Max. actions per run	Runs
GP-evolved rule	See Table 7	500	30
Random	N/A	500	30

ment, the sandbox SUT is used during evolution, while the fitness of each individual is, as before, calculated by using it as the action selection rule for the traversal-based tool TESTAR. Because in the previous experiment the Maximum and Minimum Coverage metrics did not prove very useful, they have been discarded in this experiment; hence, only two of the previously defined metrics are collected, namely Abstract States and Longest Path, with the former being used as the fitness value.

The best evolved rule, after simplification, was as follows:

```

IF NOT nTypeInto < RND
THEN pickLeastExecuted (previousAction, 0.6615-47369064)
ELSE RAC

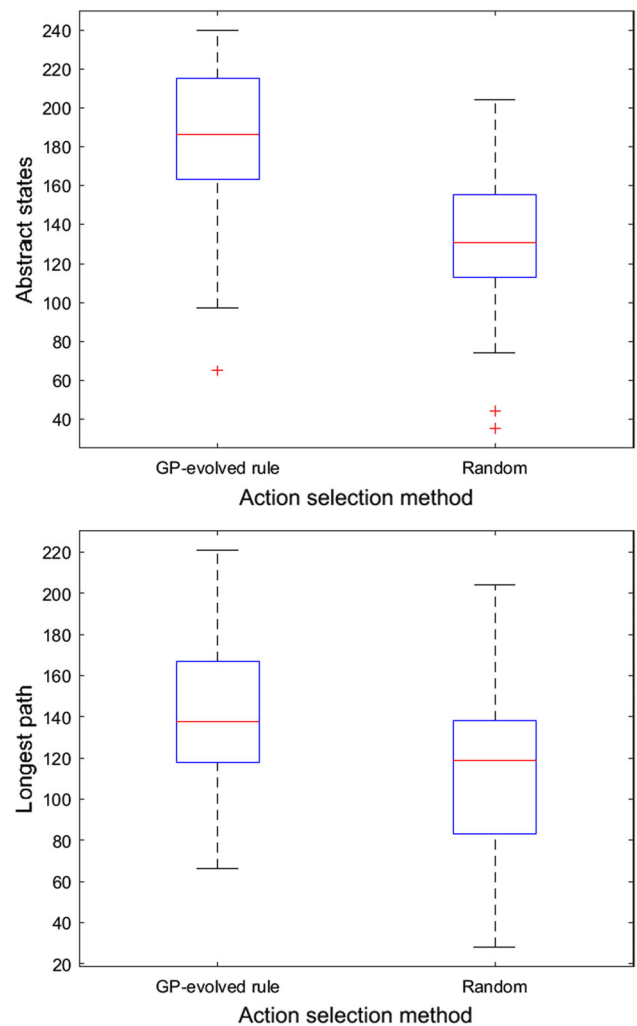
```

This solution is indeed making use of the extended set of primitives described in Table 2.

In order to carry out statistical comparisons, the validation process was repeated using random action selection.

A summary of the experimental settings is given in Table 8.

Statistical analysis As in the previous experiment, we run the Mann–Whitney–Wilcoxon non parametric test, with $\alpha = 0.05$, on the results for the two action selection mechanisms. The test shows that both metrics used have significant differences among the sets for PowerPoint and Odoo, although they are indistinguishable for Testona. The results of the tests are shown in the boxplots contained in Figs. 3, 4, and 5, which clearly indicate the superiority of the GP approach in the case of PowerPoint. For Odoo, on the other hand, random selection performs better.

**Fig. 3** Boxplots for the Abstract States and Longest Path metrics with the results obtained for PowerPoint

5.3 Discussion

It could be argued that in the best solutions evolved by the GP approach the random component introduced by many of the primitives described may lead to a discrepancy between the observed quality of the individual at run time and the fitness attributed to it during evolution. While this could indeed be the case, the statistical tests show nevertheless the superiority of the solutions evolved with the proposed method. A way to minimise the effect of randomness in the measured fitness and hence counteract the possible discrepancy between measured fitness and actual performance would be to systematically re-evaluate the best individuals at specified times; this would, however, be costly in terms of computational effort so it has not been implemented in this work; this is left for future study.

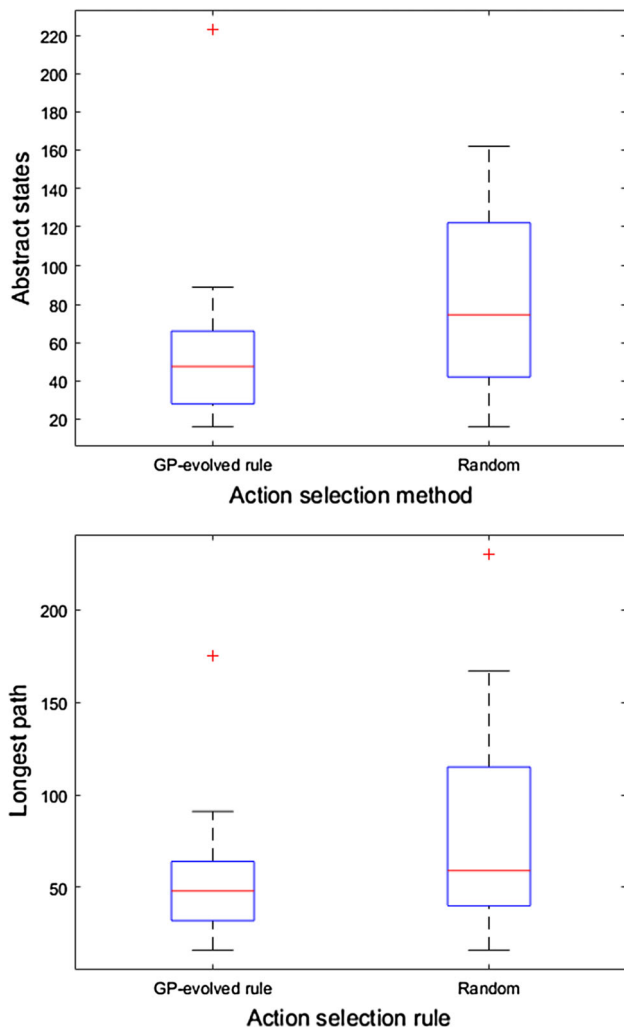


Fig. 4 Boxplots for the Abstract States and Longest Path metrics with the results obtained for Odoo

6 Conclusions and further work

We have shown the successful application of a genetic programming-evolved action selection rule within the automated testing tool TESTAR. We have carried out two sets of experiments, one with a simple primitives set and another one using more primitives.

In the first experiment an action selection strategy evolved by GP using the commercial software PowerPoint as the *sandbox* SUT was compared to Q-learning and random (or *monkey*) testing. The performance was evaluated on a commercial, desktop SUT (Testona) and an open source, web-based one (Odoo) and according to four metrics. Statistical analysis reveals the superiority of the GP approach in Odoo, although not in Testona.

In the second experiment, involving a larger set of primitives, the action selection strategy was also evolved by GP using PowerPoint as the *sandbox* SUT. Its performance was

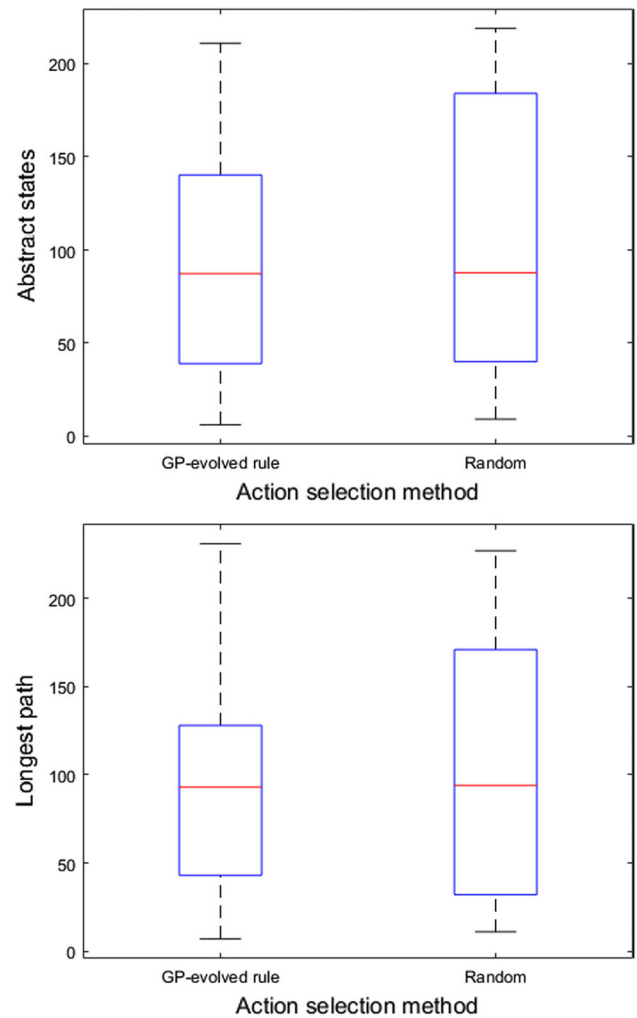


Fig. 5 Boxplots for the Abstract States and Longest Path metrics with the results obtained for TESTONA

compared to random testing, by evaluation on PowerPoint, Testona and Odoo and according to two metrics (abstract states visited and longest path traversed). The statistical analysis reveals the superiority of the GP approach over random testing when the validation is carried out on the same SUT as used for the evolution. Also, the best evolved solution, when simplified, includes some of the newly introduced functions.

Further work will thus involve analysing other options for new functions and terminals.

By analysing the best evolved solutions obtained by GP in experiments 1 and 2, we can see that there is an important component of randomness in both, as they both rely on random action selection as a last resort. This is not considered to be undesirable, as random testing has often proved to be the best option in many circumstances; but the GP-evolved rules can improve over plain random selection, as the results of the tests show.

In response to our research questions formulated in Sect. 3:

1. *Is genetic programming a good option for obtaining action-selection rules in traversal-based testing?*
→ Yes, provided the computational resources are available for the evolution process
2. *Which are the relevant features to take into consideration when selecting the next action to execute?*
→ The metrics used bias the evolutionary search towards those features related to unexecuted actions
3. *Do the evolved rules generalise to ‘unseen’ SUTs, i.e. software for which they were not evolved?*
→ Not always, so if possible, a better strategy would be to evolve SUT-specific rules.

Always an issue when using evolutionary algorithms is the computational expense. Evaluating the fitness of the solutions using TESTAR is time-costly, especially if a large number of actions is desired in the testing sequence. A possible solution would be to limit the number of fitness evaluations and use similarity to obtain an approximate fitness measure, as done in [13].

An additional conclusion regards the relevance of the metrics used; if the source code is not available direct measures of code coverage cannot be used and we are forced to use surrogate measures, as the ones proposed here. However this might not be the best option, so a step ahead would involve eliminating the fitness function completely (and hence the reliance on metrics) and guiding the evolution based on novelty only [16].

Acknowledgements The authors wish to thank Xara Sharman for her help with the graphics.

References

1. Aho P, Menz N, Rty T (2013) Dynamic reverse engineering of GUI models for testing. In: Proceedings of 2013 international conference on control, decision and information technologies (CoDIT’13)
2. Aho P, Oliveira R, Algroth E, Vos T (2016) Evolution of automated testing of software systems through graphical user interface. In: Procs. of the 1st international conference on advances in computation, communications and services (ACCSE 2016), Valencia, pp 16–21
3. Alegroth E, Feldt R, Ryrholm L (2014) Visual GUI testing in practice: challenges, problems and limitations. *Empir Softw Eng* 20:694–744. <https://doi.org/10.1007/s10664-013-9293-5>
4. Barr ET, Harman M, McMinn P, Shahbaz M, Yoo S (2015) The oracle problem in software testing: a survey. *IEEE Trans Softw Eng* 41(5):507–525
5. Bauersfeld S, Vos TEJ (2012) A reinforcement learning approach to automated GUI robustness testing. In: Fast abstracts of the 4th symposium on search-based software engineering (SSBSE 2012), pp 7–12
6. Bauersfeld S, de Rojas A, Vos T (2014) Evaluating rogue user testing in industry: an experience report. In: 2014 IEEE eighth international conference on research challenges in information science (RCIS), pp 1–10. <https://doi.org/10.1109/RCIS.2014.6861051>
7. Bauersfeld S, Vos TEJ, Condori-Fernández N, Bagnato A, Brosse E (2014) 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
8. Bauersfeld S, Wappler S, Wegener J (2011) A metaheuristic approach to test sequence generation for applications with a GUI. In: Cohen MB, Ó Cinnéide M (eds) Search based software engineering: third international symposium, SSBSE 2011, Szeged, Hungary, September 10–12, 2011. Proceedings. Springer Berlin Heidelberg, Berlin, Heidelberg, pp 173–187
9. Brameier MF, Banzhaf W (2010) Linear genetic programming, 1st edn. Springer, New York
10. Chaudhary N, Sangwan O (2016) Metrics for event driven software. *Int J Adv Comput Sci Appl* 7(1):85–89
11. Esparcia-Alcázar AI, Almenar F, Martínez M, Rueda U, Vos TE (2016) Q-learning strategies for action selection in the TESTAR automated testing tool. In: Proceedings of META 2016 6th international conference on metaheuristics and nature inspired computing, pp 174–180
12. Esparcia-Alcázar AI, Almenar F, Rueda U, Vos TEJ (2017) Evolving rules for action selection in automated testing via genetic programming—a first approach. In: Squillero G, Sim K (eds) Applications of evolutionary computation: 20th European conference, EvoApplications 2017, Amsterdam, The Netherlands, April 19–21, 2017, Proceedings, part II. Springer, pp 82–95. https://doi.org/10.1007/978-3-319-55792-2_6
13. Esparcia-Alcázar AI, Moravec J (2013) Fitness approximation for bot evolution in genetic programming. *Soft Comput* 17(8):1479–1487. <https://doi.org/10.1007/s00500-012-0965-7>
14. He W, Zhao R, Zhu Q (2015) Integrating evolutionary testing with reinforcement learning for automated test generation of object-oriented software. *Chin J Electron* 24(1):38–45
15. Koza JR (1992) Genetic programming: on the programming of computers by means of natural selection. MIT Press, Cambridge
16. Lehman J, Stanley KO (2011) Novelty search and the problem with objectives. In: Riolo R, Vladislavleva E, Moore JH (eds) Genetic programming theory and practice IX, genetic and evolutionary computation. Springer, New York, pp 37–56
17. Memon AM, Soffa ML, Pollack ME (2001) Coverage criteria for GUI testing. In: Proceedings of ESEC/FSE 2001, pp 256–267
18. Rueda U, Vos TEJ, Almenar F, Martínez MO, Esparcia-Alcázar AI (2015) TESTAR: from academic prototype towards an industry-ready tool for automated testing at the user interface level. In: Canos JH, Gonzalez Harbour M (eds) Actas de las XX Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2015), pp 236–245
19. Seesing A, Gross HG (2006) A genetic programming approach to automated test generation for object-oriented software. *Int Trans Syst Sci Appl* 1(2):127–134
20. Vos TE, Kruse PM, Condori-Fernández N, Bauersfeld S, Wegener J (2015) TESTAR: tool support for test automation at the user interface level. *Int J Inf Syst Model Des* 6(3):46–83. <https://doi.org/10.4018/IJISMD.2015070103>
21. Wappler S, Wegener J (2006) Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In: Proceedings of the 8th annual conference on genetic and evolutionary computation, GECCO’06. ACM, New York, NY, USA, pp 1925–1932. URL <https://doi.org/10.1145/1143997.1144317>
22. Watkins C (1989) Learning from delayed rewards. Ph.D. Thesis. Cambridge University

Publisher’s Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.