AutoBlackTest: a tool for automatic black-box testing

Conference Paper *in* Proceedings - International Conference on Software Engineering · January 2011

DOI: 10.1145/1985793.1985979 · Source: DBLP

4 authors:

Leonardo Mariani
Università degli Studi di Milano-Bicocca
122 PUBLICATIONS **1,525** CITATIONS
SEE PROFILE

Mauro Pezzè
University of Lugano
188 PUBLICATIONS **3,960** CITATIONS
SEE PROFILE

Oliviero Riganelli
Università degli Studi di Milano-Bicocca
44 PUBLICATIONS **177** CITATIONS
SEE PROFILE

Mauro Santoro
Università degli Studi di Milano-Bicocca
13 PUBLICATIONS **140** CITATIONS
SEE PROFILE

AutoBlackTest: A Tool for Automatic Black-Box Testing

Leonardo Mariani[§] Mauro Pezzè[†§] Oliviero Riganelli[§] Mauro Santoro[§]

[§]Department of Informatics, Systems and Communications
University of Milano Bicocca - Milano, Italy
{mariani,pezze,riganelli,santoro}@disco.unimib.it

[†]Faculty of Informatics
University of Lugano - Lugano, Switzerland
mauro.pezze@usi.ch

ABSTRACT

In this paper we present `AutoBlackTest`, a tool for the automatic generation of test cases for interactive applications. `AutoBlackTest` interacts with the application though its GUI, and uses reinforcement learning techniques to understand the interaction modalities and to generate relevant testing scenarios. Early results show that the tool has the potential of automatically discovering bugs and generating useful system and regression test suites.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Reliability

Keywords

Black-Box Testing, Test Automation, Q-Learning

1. INTRODUCTION

System testing is the important activity of executing a set of test cases on the whole application to check its conformance with the specifications. Most research and development effort focused on the problem of automating unit testing [3], leaving system testing poorly automated with the exceptions of capture and replay tools that replay system test cases [4], and GUI testing tools that check for absence of interference between GUI events [5]. Other system testing activities and in particular the generation of functional test cases are largely manual and time-consuming.

In this paper we present `AutoBlackTest`, a tool for the automatic generation of test cases for interactive applications. `AutoBlackTest` interacts with an application through its GUI, learns the most relevant ways to interact with the application itself, identifies system crashes and other domain independent problems, and generates regression test suites. `AutoBlackTest` use reinforcement learning techniques to understand the modalities of interacting with an application through its GUI, other techniques exploit learning approaches in the testing domain to tackle different classes of problems [6, 1, 9].

ICSE'11, May 21–28, 2011, Waikiki, Honolulu, HI, USA
ACM 978-1-4503-0445-0/11/05

While classic GUI testing tools aim to cover the space of the combinations of events generated with the GUI widgets, thus focusing on a sort of structural coverage, `AutoBlackTest` automatically identifies test cases that sample the ways the software under test is executed.

This paper summarizes the technique implemented by `AutoBlackTest`, describes the tool and reports an early experience on the application of `AutoBlackTest` to `Twitthere` [7], a client for `Twitter`. During the experiments the test cases automatically generated with `AutoBlackTest` found 2 unknown faults of `Twitthere` and executed 71% of the statements of the application. This experience suggests the suitability of `AutoBlackTest` to effectively test widely-used, real-life and small-size applications, such as applications for smart telephones, like iPhone and Android Apps, and small desktop clients for the quick and effective interaction with specific features of remote services, for instance clients for adding tweets on Twitter or uploading photos to Facebook.

2. AUTOMATIC SYSTEM TESTING

`AutoBlackTest` transforms the problem of automatically generating system test cases for an application into the problem of exploring an unknown environment (the application) by means of an agent. `AutoBlackTest` uses reinforcement learning to guide the agent in this exploration task [8]. The agent aims not only to explore the various features of the application under test, but also to identify the most significant features and their combinations.

In particular, `AutoBlackTest` uses Q-learning. Q-learning incrementally builds a model that represents how the application can be used. The model consists of a set of states, which represent the states of the application, and a set of state transitions labeled with the names of the actions that trigger the transitions. According to Q-learning, when the agent executes an action, it assigns the action a reward that indicates its immediate utility in that state according to the objective of the agent. The reward accumulated by each action is used to compute its Q-value that represents an estimation of the reward expected in the future if the action is executed in that state. The first time an agent executes an action in a given state, the Q-value of the ⟨action, state⟩ pair is the value of the reward returned by the execution of the action. While the agent continues interacting with the application by either executing new actions or replying known actions, Q-values are iteratively updated according to the experience gained during the learning process [8]. Executing an action with a high Q-value in the current state does not necessarily return an immediate high reward, but

the future actions will very likely return a high cumulative reward. This feature of Q-learning is extremely useful in guiding the agent towards re-executing and deeply exploring the most relevant scenarios. The agent builds the Q-learning model by alternating exploration and exploitation activities. During exploration, the agent chooses the action to execute randomly, while during exploitation the agent executes the action with the highest Q-value. Each agent execution is called episode. The user defines both the maximum length of an episode, which indicates the maximum number of actions that are executed before interrupting the episode, and the number of episodes that must be executed before interrupting the learning phase. The model obtained at the end of the learning phase represents the portion of the execution space that has been explored. The model distinguishes high-value from low-value paths, that are paths with high and low Q-values.

To effectively adapt Q-learning to testing, we need to define how the *state* of the application is perceived by the agent, what are the *actions* that can be executed by the agent, how the *reward* is assigned, and how the agent *alternates* exploitation and exploration activities. In the following, we describe how we defined these elements, and how we synthesized system test cases from the model produced by the learning phase.

2.1 Learning

State extraction. In our case, the state of the environment is the state of the application. Since the concrete state of the application is huge (it includes the value of variables, objects, registries, etc.) and hard to abstract without using a detailed specification of the implementation, we heuristically approximate the state of the application with the state information that is directly visible to the user: the state of the GUI. In particular, we abstract the state of the GUI by defining a set of operators that extract the relevant data from the widgets in the current view. For instance, we extract the number of items in a listbox but we do not include the values of these items in the state; similarly we extract the length of the strings in the textareas, rather than using the values in the textareas.

This notion of state let us build models that can be feasibly analyzed. Moreover, we can estimate how much an action impacts on the execution of the system according to the effects that the execution of the action has on the state. We can find examples of functions that execute the application without impacting on the state, and viceversa. However, we found that our approximation is a good approximation of many scenarios that occur frequently in the practice.

Agent actions. The catalog of actions that are available to the agent is composed of simple and complex actions. Simple actions are atomic actions executed on a single widget. For instance, left clicking a button or typing some text in a textarea. When a simple action has parameters, such as the row of a listbox that should be clicked or the text that should be written in a textarea, we pick up a value from a predefined list of possible choices identified according to the boundary testing criterion, for instance, clicking the first or the last item in a listbox, writing a text of length 0, 1 or the maximum allowed by the textarea.

A complex action is a workflow of simple actions derived according to heuristics that represent situations that commonly occur in practice. For instance, if the current view includes multiple widgets that can be filled in, like textareas or checkbox, and at least one button, `AutoBlackTest` activates a complex action that interacts with the set of widgets that can be filled in (different variants of this complex action may fill in all or a subset of these widgets) and then clicks on one of the available buttons. This complex action eases the compilation of form-like views. Similarly, further complex actions are available for other situations like the presence of frames that split a view into multiple sub-views.

Reward function. The reward function gives a feedback to the agent about how well it executed the application. We estimate the quality of the execution by means of the extent of the changes on the state. Intuitively, actions like writing some text in a textarea do not cause significant uses of the application, and do not change much the GUI, and hence the state. On the contrary, actions like submitting a request may produce a new view radically different from the previous one, thus both significantly changing the GUI and producing a relevant use of the application. The incremental refinement of the Q-values according to the reward will increase the importance of the actions with low impact (low reward) that are necessary to execute actions with high impact (high reward), guiding the agent toward the execution of different variants of the most relevant scenarios.

In more details, given two GUI states s_1 and s_2, the reward function associates to an action a that changes the state of the application from s_1 to s_2 a numeric utility value, computed by counting the number of widgets that are present in s_1 and changed in s_2. A widget that disappeared contributes to the computation of the reward with a value of 1. A widget that is present in both states contribute to the reward with the fraction of properties that are returned by its state abstraction function, and that changed their value. Thus, a widget that changed all its property values contributes with a value of 1; if no attributes changed, the widget does not contribute to the reward; otherwise, the widget contributes with a value between 0 and 1. The overall sum of the contributions is normalized with respect to the number of widgets in s_1.

`AutoBlackTest` also computes coverage data while interacting with an application. Coverage data are useful to understand the capability of the technique to identify new uses of the system, and can be used to refine the reward.

Agent behavior. In `AutoBlackTest`, we need to combine a good degree of exploration to lead the agent toward the discovery of new uses of the application, with an incentive toward exploitation to increase the chance of exploring several variants of relevant (highly-rewarding) scenarios. Thus, we assign the same probability to exploration and exploitation activities. Since complex actions have a higher probability to be effective than simple ones, we increase the chance of preferring complex over simple actions. In particular, when the agent executes a random action and has a choice between complex and simple actions, it choses a complex action with a probability of 70% and a simple action with a probability of 30%.

2.2 Generating regression test suites

The learning phase explores the execution space of the application and produces a model that represents different ways of using the system. In principle, we can use the mode to synthesize a set of regression test cases by applying coverage criteria for graphs. However, the large size of the model

would result in a huge set of test cases expensive to be executed. Since the Q-values associated with ⟨action, state⟩ pairs represent the relevance of the action according to our heuristic, we synthesize a test suite that covers all the transitions with Q-values above a given threshold. The threshold can be adaptively defined by the tester depending on the desired size of the test suite. The generated test suites likely include the majority of the relevant uses of the application and can be used for efficient regression testing.

3. AUTOBLACKTEST

The `AutoBlackTest` prototype implementation integrates `IBM Rational Functional Tester` [4] and `TeachingBox` [2].

`IBM Rational Functional Tester (RFT)` is a capture and replay tool for automatic regression testing. `AutoBlackTest` uses `RFT` to extract the list of widgets present in a given GUI, to access the state of the widgets and to interact with the widgets. `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. Thus, `AutoBlackTest` can be readily applied to applications developed with many different GUI frameworks.

`AutoBlackTest` uses `TeachingBox` to implement the Q-learning algorithm. `TeachingBox` implements the core of the Q-learning algorithm, which can be adapted to the specific needs by implementing the state abstraction functions, the actions and the reward function.

`AutoBlackTest` also provides the capability to generate a test suite that can be automatically re-executed with `RFT` from the model obtained as a result of the learning phase.

4. PRELIMINARY VALIDATION

We validated `AutoBlackTest` on the `Twitthere` application [7]. `Twitthere` is an application for rapidly posting, modifying, deleting and reading tweets on `Twitter` and represents well the `AutoBlackTest` target application domain of small-size applications.

We executed `AutoBlackTest` for 200 episodes of length 20 (it took about 24 hours). During the learning phase `AutoBlackTest` discovered 2 faults.

The first fault is a wrong way to handle missing replies from the `Twitter` service, which happens from time to time in the daily hours when the service is extremely busy. In this case, the application hangs and cannot be used anymore.

The second fault is a wrong implementation of the delete tweet command. The application deletes the tweet from the local list, but does not delete the tweet from the server. In fact, the same tweet reoccurs if the refresh button is clicked. We identifies this fault easily by inspecting the `Twitthere` console after executing `AutoBlackText`. `Twitthere` traces a `java.lang.IllegalStateException` exception every time a tweet is deleted, and thus it traces wrong delete actions.

The learning phase produced a model with 462 states and 1176 transitions. `AutoBlackTest` covered 71% of the statements, thus showing a good capability of executing the application. The regression test suite that `AutoBlackTest` synthesized from the model could be successfully executed and covered many scenarios relevant from the usage profile viewpoint, including numerous combinations of multiple operations such as posting multiple tweets, posting and deleting a tweet, etc.

5. CONCLUSIONS

In this paper, we introduced `AutoBlackTest`, a tool for the automatic generation of test cases for interactive applications. `AutoBlackTest` automatically detects different ways of using the application under test, catches domain-independent failures, identifies the most relevant uses among the ones that have been included in the model and synthesizes a regression test suite that covers these uses.

We are currently working to extend the tool into three main directions. We are working on a second generation prototype tool that improves the implementation of parameterized actions by defining smarter strategies to generate parameter values. We are studying different ways of integrating the code coverage data into the reward function to drive the agent toward the execution of actions that execute large portions of the program source code. Finally, we are extending the prototype tool to work with a specification of the system, for instance a set of use cases, to focus the analysis on specific scenarios rather than the entire application, and thus scale to large applications.

A recorded demo is available at `http://www.youtube.com/user/DISCoUniMiB`.

6. ACKNOWLEDGMENTS

The authors want to thank Andrea Mattavelli, Mattia Vivanti and Julia Weekes for their help with the preparation of the demo.

7. REFERENCES

[1] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. *SIGSOFT Software Engineering Notes*, 29:195–205, 2004.

[2] W. Ertel, M. Schneider, R. Cubek, and M. Tokic. The teaching-box: A universal robot learning framework. In *Proceedings of the International Conference on Advanced Robotics*, 2009.

[3] J. D. Halleux and N. Tillmann. Parameterized unit testing with Pex. In *Proceedings of the International Conference on Tests and Proofs*, 2008.

[4] IBM. IBM rational functional tester. http://www-01.ibm.com/software/awdtools/tester/functional/, visited in 2010.

[5] A. Memon, M. E. Pollack, and M. L. Soffa. Hierarchical gui test case generation using automated planning. *IEEE Transactions on Software Engineering*, 27(2):144–155, 2001.

[6] D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. *Journal of Automata, Languages and Combinatorics*, 7(2):225–246, 2002.

[7] L. Rodenburg. Twitthere. http://twitthere.wordpress.com, visited in 2010.

[8] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[9] M. Veanes, P. Roy, and C. Campbell. Online testing with reinforcement learning. In *Proceedings of the International Workshop on Formal Approaches to Software Testing and Runtime Verification*, 2006.