Sequence Mining for Automatic Generation of Software Tests from GUI event traces

Alberto Oliveira[1], Ricardo Freitas[1], Alípio Jorge[1], Vítor Amorim[2], Nuno Moniz[1], Ana C. R. Paiva[3], and Paulo J. Azevedo[4]

[1] LIAAD-INESC TEC, FCUP-University of Porto, Porto, Portugal
alberto.p.oliveira@inesctec.pt
alipio.jorge@inesctec.pt
[2] RandTech Computing,R&D, Porto, Portugal
vitor.amorim@rtcom.pt
[3] INESC TEC, FEUP-University of Porto, Porto, Portugal
[4] INESC TEC, University of Minho, Braga, Portugal

Abstract. In today's software industry, systems are constantly changing. To maintain their quality and to prevent failures at controlled costs is a challenge. One way to foster quality is through thorough and systematic testing. Therefore, the definition of adequate tests is crucial for saving time, cost and effort. This paper presents a framework that generates software test cases automatically based on user interaction data. We propose a data-driven software test generation solution that combines the use of frequent sequence mining and Markov chain modeling. We assess the quality of the generated test cases by empirically evaluating their coverage with respect to observed user interactions and code. We also measure the plausibility of the distribution of the events in the generated test sets using the Kullback-Leibler divergence.

Keywords: Software Testing · Frequent Pattern Mining · Markov Chains · Data Mining

1 Introduction

Software development is a complex and continuous process that requires frequent changes in the code [1]. Each change can introduce errors that affect the ability of the software maker to timely deliver a quality product [2]. Errors in software can cause distrust in software users but can also lead to substantial economic losses [3] and even the sacrifice of human lives [4]. Taking into account that software development is becoming increasingly more agile [5], with systems undergoing constant changes, the moments for introducing errors are multiplying.

The software industry typically relies on test cases that are executed before each release [6]. Although the automation of test checking is a common practice [7], the set of tests is bounded to the ones previously defined and planned. Moreover, the design of test cases is mostly based on human expertise [8]. However, manually devising software tests demands much time, costs and effort of human software testers [8]. Correctly selecting the tests and evaluating their outputs is crucial in order to efficiently improve the quality of software [8].

In this paper, we propose an adaptable framework for learning software test generators from user interaction data. It has been developed in the context of a software company that produces the web-based application Anywhere+, a platform for managing insurance products. Nevertheless, our proposed approach can be used with any web GUI-based software. In our pipeline, the first step is to store the user interaction logs. A browser plugin captures this data as users work normally. From this data, we discover sequential micro-patterns using sequence mining. The third step is to chain the discovered patterns into a global Markov chain model. Finally, this model is used to generate test cases based on these patterns automatically. The approach is tested on real data in terms of coverage and plausibility of the generated patterns. As a result, we have obtained stable growth rates in terms of coverage – adding more generated tests increases our coverage metric value, even reaching full coverage for one of the cases – and very low values of the Kullback-Leibler's divergence between the distribution of actions in user sessions and in artificially generated tests.

This paper is organized as follows. We first discuss related work. Then we give an overview of the software's deployment pipeline. We describe how data is collected, how frequent sequences are found and how we use Markov chains to produce our test generator. We wrap up with evaluation and conclusions.

2 Related Work

Given the importance of the software development process and the tremendous possibilities that AI can bring to it [9], this is a fertile ground for AI research. Many works can be found in the last two decades with contributions to different phases of the process and in particular to test generation.

Isabella and Retna [10] present a general overview of test case generation for GUI based testing. This includes generation of test cases, repairing infeasible test suites and multiple GUI testing tools over various types of software, as well as its usage advantages and disadvantages. Conroy et al. [11] proposed a generic method for generating tests for testing web services from their reference legacy GUI applications. This work mainly relies on the concept that GUI elements are programming objects whose values can be set and retrieved and whose methods are associated with actions that users perform on these elements, which is very similar to our solution's plugin purpose.

The closest work to ours in spirit and method is the one accomplished by Zhou et al. [12]. It first builds a Markov usage model based on improved state transition matrix (STM), which is a table-based modeling language. It then generates a software reliability test method, including test case generation and test adequacy determination using the previously created Markov usage model. An improved Kullback discriminant was chosen as the judgment criteria of convergence from the test chain to the usage chain in order to measure if the testing process is sufficient.

Lastly, the approach of Last et al. [8] aims to automate the input-output analysis of execution data based on a machine learning methodology. This methodology relies on the info-fuzzy network (IFN), which has a tree-like structure. The network is used to predict output values given test-cases.

3 The Software Deployment Framework

The main contribution of this paper is the software test generation process that relies on captured GUI events data. This is part of a broader software deployment framework for an insurance ERP called Anywhere+ developed by RandTech Computing (`https://rtcom.pt`). It has been extended to incorporate this Artificial Intelligence component, from data collection to test generation. This framework automates the complex software update process, including code, databases and tests, and enables safer, faster and more frequent updates. It is flexible enough to be easily adapted to other deployment flows. The framework follows a modular structure (Figure 1).

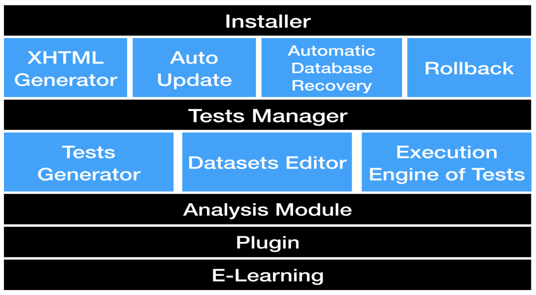

Fig. 1: Software Deployment Framework

The Plugin component collects interaction data from the browser as the application is used. This is high-level data that represents business events. In the Analysis Module this data is used to induce a model for generating tests. The Tests Generator uses the model to generate software tests. Since some actions of the tests require specific values (for example, filling in the name of a client), this is provided by a specific dataset of attribute-value pairs created using the Datasets Editor. When the software tests are automatically executed, the E-learning component captures the sequence of screenshots that can later be used for user training. The Installer is the component that deploys new versions of the software. The XHTML generator automatically transforms the XHTML files which compose the application's UI. This generator is capable of assigning graphical widgets to high-level functional categories corresponding to embedded business concepts. This is important to give semantics to the events to be logged by the Plugin. The AutoUpdate component warns users of new updates. The Automatic Database Recovery changes the structure of the database if needed, and Rollback is there to recover the previous database if anything goes wrong.

4 Data acquisition

As the application is used, GUI events are continuously recorded by the Plugin component. This provides a memory of the real sessions that will drive the building of test case scenarios.

In order to define and execute the software tests, it is fundamental that the framework can recognize the various business concepts. For that, we have defined a syntax for the XHTML generator which recognizes a set of patterns used on the UI design of the Anywhere+ application. The XHTML files which compose the Anywhere+ application are transformed by inserting the reference to the business concepts. Despite its complexity, this transformation occurs transparently and automatically, both for programmers as for users and is triggered at each build of the application.

4.1 Data Format

User activities are recorded on a text file, following a simple and optimized structure for the analysis task. The interactions' format firstly contain the *timestamp*, *session id*, *tab id* and the *business concept* separated by a comma. Secondly, there are three components separated by semicolon: action/command, target and value.

The action field is mandatory, but target and value parameters may be void. The captured user sessions are the input for the Analysis Module that builds output for the Test Generator. This is a Markov model whose states are sequences of user actions. The Test Generator pre-processes the interaction events logged and looks for frequent sequences with a given maximum length. The resulting sequences are chained into a single Markov model. This model is built by identifying all the initial states from the sequences and then, for each initial state, it explores the next states. The transition probabilities are estimated using the number of transitions from a determined current state to a next state. The Markov chain model is then output in JSON format.

5 Frequent Sequence Identification

To generate the frequent sequence patterns, we considered various frequent sequence mining algorithms. These algorithms can be categorized by their search approach as breadth-first search or depth-first search. Depth-first search algorithms need less database scans in order to obtain all frequent sequences so they are more computationally efficient with larger databases.

Fournier-Viger et al. [13] proposed the data structure CMAP (Co-occurrences Map, CMAP) capable of keeping a co-occurrences map of items extracted from a single database scan and also a new approach to the sequence pruning stage based on this data structure and on the co-occurrences' properties. The CM-SPAM algorithm is an optimized version of SPAM [14] for the frequent pattern mining task. The SPAM algorithm first constructs a vertical sequence representation of the sequences database and obtains the set of frequent items, according to the given minimum support parameter. It then searches for candidate patterns based on the set of frequent items. SPAM uses bitmaps for faster pattern joining operations. The algorithm outputs sequences and their respective frequency.

This library uses the IBMGenerator format for sequence databases, which is represented by a binary file of integers ordered by little-endian, where positive values represent events, -1 represents the separation between events and -2 represents the end of a sequence.

6 Software Test Generator

A Markov model provides us with flexible, easily understandable representations of the operational profiles of given programs or software systems [15]. The Markov property says that the probability distribution of future states of a process relies only upon the current state. Therefore, a Markov model captures the time-independent probability of being in state s_1 at time $t + 1$ knowing that the state at time t was s_0. The relative frequency of event transitions during program executions provides a probability estimate for each possible immediate state.

For the deployment framework, the Markov model is represented as a dictionary structure, $< key, values >$, where key is the current initial state (associated with the previous actions) and $values$ is a list of future actions associated to their probabilities. The states in our application are the actions performed by the user. This model is stored in JSON format.When used to reproduce sequences, Markov models can lead to infinite cycles of alternating states. To avoid that, we have adopted an end-of-sequence token which, when generated, terminates the sequence.

Given the Markov model, tests are generated in multiple ways using a proportional sampling criterion. This approach, proposed by Zhou et al. [12], divides the choice interval space, making it between [0,1] and splitting by the occurrence probabilities of each action (for example, if we have A = 0.3, B = 0.2 and C = 0.5, our interval will be split into intervals of [0,0.3],]0.3,0.5] and]0.5,1]). A random number between 0 and 1 is generated and the action is chosen according to where the generated value fits (for example if the generated value is 0.4, the chosen action will be B).

The generated tests correspond to interaction paths that could be followed by the platform's users. The test generation algorithm takes three parameters, *N*, *L* and *markov*. *N* represents the order of the Markov model, i.e., the number of actions to be taken into account for the next action of the Markov chain. *L* is the number of sequences we want to generate, and *markov* is the Markov model generated by the Analysis module.

7 Evaluation

The current evaluation of our approach focuses on code coverage and plausibility. In our experiments, we generate large numbers of tests and observe how these two dimensions evolve. We aim at assessing the quality of the generated tests, as well as determining the minimum number of tests that must be generated to ensure quality.

7.1 Metrics

The metrics used are Proportion of Actions Covered (PAC) and the Kullback-Leibler's (KL) divergence [16]. PAC is the ratio between the number of distinct actions in real sessions and the number of distinct actions in sessions built by the Analysis Module. This metric does not measure the code coverage directly since it is based on the *de facto* users' actions. If a part of the code is never involved in real sessions, it is not tested. However, the more PAC grows, the more code is tested. The Kullback-Leibler's divergence [16] is a comparison measure between two probabilities' distributions. Using it, we can compare the distribution of events in the real session with the generated ones. The expression to calculate this divergence is presented below.

$$D_{\mathrm{KL}}(P||Q) = \sum_{i} P(i) log \frac{P(i)}{Q(i)} \tag{1}$$

7.2 Results

Code Coverage We performed the test for the code coverage using a growing number of N (from 1 to 6) and a maximum value for L (600). For each value of N, we generated 0 to 600 tests and cumulatively measured the coverage of the tests. In Figure 2, we observe how PAC grows with the number of generated tests, reaching a value of 1 to $N = 1$ and nearly 1 for the remaining N. Although there is no clear relation between N and the measured of PAC, higher values of N do not seem to pay off in terms of coverage. As we will see, this is also the case with plausibility. In any case, a number of 600 test cases already offers excellent coverage. The parameter N (size of the frequent sequences) does not seem to have a clear influence. This indicates that the number of tests has to be relatively large to assure high coverage.

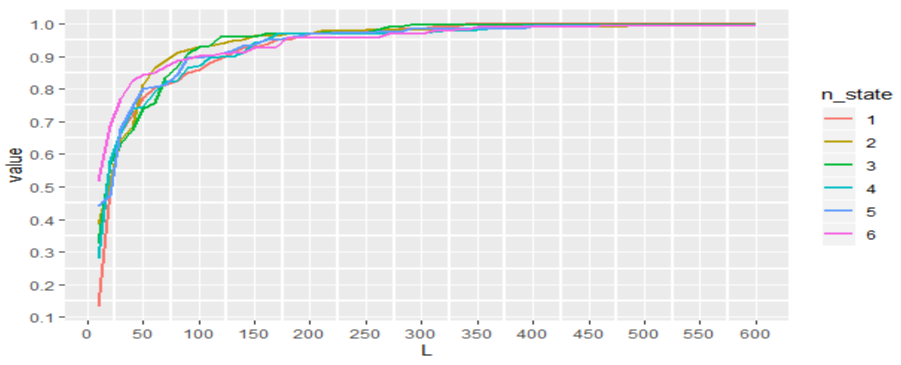

Fig. 2: Evaluation results for code coverage (PAC)

Kullback-Leibler's divergence We have executed an experiment similar to the previous one for measuring the plausibility of the generated tests. Now we measure KL of the produced distribution of events given the observed one. We obtained the results shown in Figure 3. We see that KL tends to zero for all values of N (a KL value close to 0 indicates that the generated sequences are a good representation of real sequences). With $L > 2500$ test cases, we already obtain plausible distributions for all N. This shows that we can find various safe pairs, L and N. Combining both evaluation dimensions, and taking into account that lower values of N and L are preferable for computational reasons, right combinations would be $N \in \{1, 2, 3\}, L \geq 2500$.

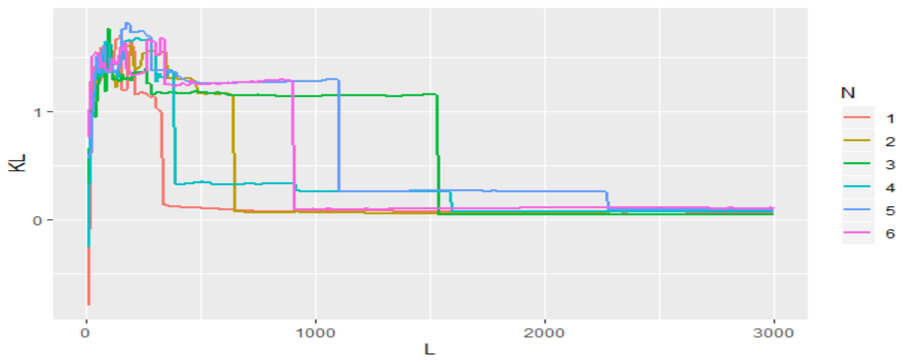

Fig. 3: Evaluation results for the Kullback-Leibler divergence

8 Conclusions

With this work, we have managed to implement a tool that automatically generates software tests based on GUI event logs. This proposed solution has a high degree of adaptability for easy adoption by other systems. However, to implement this methodology, it is necessary for some degree of permanent system users', in order to obtain useful results. For further research/improvements, we will deepen the presented empirical study and consider other dimensions. Currently, the Kullback-Leibler's divergence does not compare distributions of micro sequences but only of individual items. Despite the merits of the PAC metric, which give more importance to more frequent user actions, we should also measure the plain coverage of code. These metrics are continuously and automatically obtained throughout the software development and deployment process. It is, therefore, important to provide developers with dashboard tools for easy access to these performance indicators. In another line of evaluation, we are designing an A/B test methodology that enables the direct comparison of the performance of automatic software testing with manual test design.

Acknowledgments

This work is financed by the Northern Regional Operational Program, Portugal 2020 and the European Union, through the European Regional Development Fund (`https://www.rtcom.pt/wordpress/rute-randtech-update-and-test-environment/`). Also, this work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project UIDB/50014/2020.

References

1. Ajouli, Akram and Henchiri, Khouloud, MODEM: an UML profile for MODEling and Predicting software Maintenance before implementation, 2019, https://doi.org/10.1109/ICCISci.2019.8716421

2. M. Choetkiertikul, H. K. Dam, T. Tran and A. Ghose, *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Predicting Delays in Software Projects Using Networked Classification (T), 2015, pp. 353-364, https://doi.org/10.1109/ASE.2015.55
3. L.Grossman. (2010) Metric math mistake muffed mars meteorology mission. [Online]. Available: `https://www.wired.com/2010/11/1110mars-climate-observer-report/`
4. L. Kelion, "Fatal a400m crash linked to data-wipe mistake", BBC, 2015. [Online]. Available: `https://www.bbc.com/news/technology-33078767`
5. M. A. Babar, A. W. Brown, and I. Mistrik, *Agile Software Architecture: Aligning Agile Processes and Software Architectures*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.
6. R. Florea and V. Stray, "The skills that employers look for in software testers", *Software Quality Journal*, 2019.
7. B. Anderson. (2017, Oct.) Best automation testing tools for 2019 (top10 reviews). [Online]. Available: `https://medium.com`
8. M. Last, M. Friedman, and A. Kandel, "Using data mining for automated software testing", *International Journal of Software Engineering and Knowledge Engineering*, vol. 14, no. 4, pp. 369–393, 2004, https://doi.org/10.1142/S0218194004001737
9. D. L. Giudice, "How AI Will Change Software Development And Applications Key takeaways", 2016.
10. A. Isabella and E. Retna, "Study paper on test case generation for GUI based testing", CoRR, vol. abs/1202.4527, 2012. [Online]. Available: `http://arxiv.org/abs/1202.4527`
11. K. Conroy, M. Grechanik, M. Hellige, E. Liongosari and Q. Xie, “Automatic test generation from GUI applications for testing web services”, Oct. 2007, pp. 345–354.
12. K. Zhou, X. Wang, G. Hou, J. Wang and S. Ai, "Software reliability test based on markov usage model", JSW, vol. 7, no. 9, pp. 2061–2068, 2012, https://doi.org/10.4304/jsw.7.9.2061-2068
13. P. Fournier-Viger, A. Gomariz, M. Campos and R. Thomas, "Fast vertical mining of sequential patterns using co-occurrence information", May, 2014.
14. J. Ayres, J. Flannick, J. Gehrke and T. Yiu, "Sequential pattern miningusing a bitmap representation", July, 2002.
15. W. J. Gutjahr, "Software dependability evaluation based on markov usage models", *Perform. Eval.*, vol. 40, no. 4, pp. 199–222, 2000, https://doi.org/10.1016/S0166-5316(99)00052-8
16. S. Kullback, *Information Theory and Statistics*. New York: Wiley, 1959.