GUI Testing in Production: Challenges and Opportunities

Giovanni Denaro, Luca Guglielmo, Leonardo Mariani, Oliviero Riganelli
University of Milano-Bicocca
[Giovanni.Denaro|Luca.Guglielmo|Leonardo.Mariani|Oliviero.Riganelli]@UniMiB.It

ABSTRACT

Automatic system testing of commercial software applications is extremely challenging and requires facing many issues, including the integration of test generators with the software process of the organization that is in charge of verification and validation activities. In this paper, we discuss the challenges that we faced in the early stages of a technology transfer project aimed to introduce a test generator in a software house that develops Web-based Enterprise Resource Planning (ERP) solutions, and the insights that we gained.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

ACM Reference format:
Giovanni Denaro, Luca Guglielmo, Leonardo Mariani, Oliviero Riganelli. 2019. GUI Testing in Production: Challenges and Opportunities. In *Proceedings of Companion of the 3rd International Conference on Art, Science, and Engineering of Programming, Genova, Italy, April 1–4, 2019 (Programming '19),* 3 pages.
https://doi.org/10.1145/3328433.3328452

1 INTRODUCTION

System test case generation techniques can automatically generate test cases that stimulate an application under test using its GUI. Existing approaches address a range of platforms and GUI technologies, including desktop [3, 6, 11, 15, 17, 18, 23, 25], Web [4, 13, 16, 19, 20] and mobile applications [1, 2, 12, 24], and exploit different strategies to generate test suites that satisfy criteria such as covering sequences of events [18], sequences of interacting events [23], data-flow relations among event handlers [3], and code coverage [11]. However, these techniques have been mostly experienced with open source software, and there is limited knowledge about both their effectiveness with commercial systems and the issues that can arise when integrating them with the development process of a software company.

In this paper, we report our early experience with the automatic system testing technique *ABT* [14, 15], applied to a Web-based ERP solution developed by a mid-size Italian software house. ABT is a state-of-the-art model-based technique that exploits Q-Learning to automatically generate system test cases. The reported experience let us discuss some ideas that can be exploited to improve both *ABT* and automatic model-based test generation techniques in general.

The specific Web-based ERP solution considered in this project is an application that handles the commercial process of a multinational company, managing data entities like orders, invoices, shipping activities, and maintenance requests. The development was conducted by a team of six software engineers. The testing activities are carried out by a team of three professionals who develop and maintain the test plan and manually perform system testing.

The major challenges that we faced in our experience concerned three main aspects: *scalability*, *lack of oracles* and *test reporting*. We discuss these challenges in Section 2. Interestingly, we found that the domain-specific nature of these challenges allows for the definition of domain-specific solutions that may improve the effectiveness of current model-based testing techniques, including ABT, up to the level necessary to address commercial ERP applications. These solutions concern with *GUI partitioning*, *ad-hoc operations*, *business rules*, and *test-plan coverage* techniques. We discuss them in Section 3. We summarize our final remarks in Section 4.

2 THE CHALLENGES OF TESTING BUSINESS ORIENTED APPLICATIONS

This section discusses the three main challenges that we identified based on our experience with a commercial Web-based ERP solution developed and maintained by a partner company: scalability, lack of oracles, and text reporting.

Scalability. The execution space of all interaction sequences induced by the GUI of business oriented applications is indeed both large and strictly structured.

The *size of the execution space is large* due to the many choices that are available to the user at each step of the execution. In the commercial application that we are considering, at every step of the execution the user can interact with 95 menu and button items available on the top menu bar, in addition to the widgets displayed in the center of the window. Since the top menu bar is continuously available for interaction, the space of all the possible interactions grows exponentially with the length of the interaction sequence, making a thorough exploration of the execution space impossible in practice. With 95 executable actions at every step, even limiting the test space to simple test sequences of five steps will result in more than 7 billion test cases, and this number further increases of several orders of magnitude if we consider the additional actions that become available after selecting the menu items. Covering such an execution space is infeasible, and any test generation strategy must be properly guided to produce relevant test cases.

The *strictly structured organization of the GUI* exacerbates the problem of generating relevant test cases. In fact, the way applications are used in practice follows strict patterns that depend on the goal of the user. For instance, if a user wants to modify an order, she/he will click on the order menu, will select an order, and finally

Programming '19, April 1–4, 2019, Genova, Italy

ACM ISBN 978-1-4503-6257-3/19/04…$15.00
https://doi.org/10.1145/3328433.3328452

will edit it. On the other hand, users are less likely to move from one area of the application to another one without finalizing any operation, such as opening the list of the orders, selecting an order, then moving to invoices without making changes on the orders, and then again moving to customers without changes on the invoices. This means that the execution space contains many meaningless interaction sequences, and that test case generation techniques can hardly distinguish them from the useful interaction patterns.

To give a concrete feeling of the impact of the above considerations, our experience with an industrial business oriented application indicates that the length of the minimal interaction sequence of the operations that lead to data modifications can range between 8 and 38 steps. When the probability of picking up a specific action is close to 1 over 100, it is evident how most of these interaction sequences are similar to singularities in the execution space, and it is almost impossible to exercise them with randomized GUI exploration strategies. For instance, by executing ABT in overnight test generation sessions (about 10 hours), we easily succeeded to exercise the functional behaviors that depend on short interaction sequences, but generally missed important portions of the functional logic of the application under test. For example, ABT easily explored the selection of the graphical menus in several different orders, but never tested interaction sequences that lead to the modification of relevant domain objects.

Lack of Oracles. In most industrial domains, there exist many relevant classes of failures that cannot be captured as explicit runtime exceptions and hangs, and testing tools that ignore these classes of problems are relatively useful. For instance, the large majority of the test requirements in the test plan of the considered business oriented application, written by professional test engineers, include steps that check the correctness of the execution by inspecting the GUI state, the database state, or both. Common examples include: ensuring that graphical menus, buttons and text fields appear with correct labels and can or cannot be modified in given screens and application states; ensuring that data change operations result in correct updates of given database tables; and verifying that, based on the user inputs, given screens visualize the correct subsets of data items among those existing in the database. Failures in satisfying these or similar types of test requirements would not cause any runtime exception or hang, and would thus be missed by most automatic testing techniques, including ABT. Improving the effectiveness of *the oracle* is a necessary condition to address an industrial context.

Test Reporting. To be effective, a test generator must demonstrate the ability to alleviate the testing effort for validating and assessing the applications under test. Many techniques [5, 7–10, 21, 22], including ABT, provide their results as (i) executable test scripts to replay the generated test cases, (ii) test reports on the execution contexts of uncaught exceptions and hangs detected while generating the tests, (e.g., a string representation of an exception and its stack trace), and (iii) code coverage indicators that quantify the achieved test adequacy. However, based on the feedback that we collected from our industrial partners, these results are not sufficient to produce a significant reduction of the testing effort.

A fundamental requirement of industrial testers is to explicitly identify the functional behaviors exercised with the generated test cases. In our project this was indicated as the key piece of information to optimize the test effort. The test scripts and coverage reports that are typically produced are useful, but they convey little information about the functional behaviors that were and were not tested. Mapping the generated test cases to corresponding functional behaviors requires testers to manually replay step-by-step each single test script, which almost nullifies the benefits of using a test generator. Similarly, deducing the uncovered functional behaviors by looking for the inputs that lead to uncovered code is known to be very demanding. As a matter of facts, testers can hardly use the current ABT reports to effectively plan the additional testing effort to improve the functional adequacy of their tests with respect to functional behaviors that were missed or only partially tested by ABT. Augmenting ABT with reports that include *detailed information about the tested and the untested functional behaviors* is a key necessity to integrate ABT in an industrial testing process.

3 OPPORTUNITIES FOR IMPROVEMENT

Arguably the above discussed challenges generalize to many industrial applications beyond the application domain considered in this paper. Though general solutions seldom exist, industrial business oriented applications can be addressed with domain specific solutions that stem from the recurring patterns that relate the GUI structure to the business logic, and thus to the test requirements of these applications. In the following we discuss GUI partitioning and ad-hoc operations, to address the scalability challenge; business rules, to address the lack of oracles challenge; and test-plan coverage, to address the test reporting challenge.

GUI Partitioning. A first observation is that, in most business oriented applications, since each graphical menu gathers the operations to manipulate a given type of data entity among the ones that are handled by the underlying information system, the graphical menus partition the functional logic of the application in sub-areas. For example, in the industrial application that we studied, the functionalities that manipulate invoices can be exercised (only) with interaction sequences that start with the selection of the graphical menu *Invoices*. This particular type of GUI structure, which is designed to improve usability, can be exploited to mitigate the combinatorial explosion of the interaction sequences that the test generator may produce. The test generator can be guided to focus on interaction sequences that both start from distinct graphical menus and do not include actions that jump across different graphical menus without completing any operation. In this way, the test generator may achieve a dramatic reduction of both the size and the complexity of the execution spaces that must be sampled, improving the significance of the generated tests as well.

Ad-hoc Operations. Operations that depend on long interaction sequences, such as the operations to create and modify data entities, represent a very challenging class of operations. Long interaction sequences are often caused by the presence of forms with many input widgets that have to be filled-in. Input forms with twenty, thirty or more input fields are common cases in industrial business oriented applications. The bottom line is that a test generator may gain a lot in effectiveness if input forms are handled in an ad-hoc fashion, for instance with smart operations that extensively fill-in forms before submitting them, instead of relying on random

choices that may partially fill-in forms, with little chance of successfully submitting a request that satisfies the many constraints on the validity of the input data.

Business Rules. Our interaction with engineers revealed that there usually exists a well identified and relatively small set of output data that capture the most relevant classes of *semantic* failures. In our project, many test requirements were concerned with checking the correctness of either attributes of widgets that frequently appear in the GUI (e.g., graphical menus, buttons, text fields and data grids) or database changes produced as a result of an operation. For example in the considered application, the engineers designed a rule that requires to crosscheck that, after selecting the menu *Invoices*, the GUI shows a panel titled *Invoices* with all the required buttons, and that, after entering an invoice form with valid data, the database table *INVOICES* contains a new record with the same data. Properly codified rules like this one can allow us to augment the test case generation process with a non-trivial failure detection ability. We thus envision the possibility of defining a domain-specific language that allows engineers to specify business rules that can be exploited by the test case generation techniques.

Test-Plan Coverage. Although code-level metrics can be useful, engineers need to have a proper understanding of the requirements that have been tested, especially for system-level testing activities. This information is typically encoded in test plans. For example, the test plans of our partner are spreadsheets organized in sections, where each section is a sheet that indicates the set of test objectives that must be satisfied for a specific entity. For instance, a section may indicate the set of behaviors that should be tested when validating the functionalities about the handling of invoices, and a single test objective is a statement such as "when the *new invoice* button is pressed, a form with five tabs and only empty fields must be shown to the user". Each test objective is a row in a sheet that reports the set of operations to be performed (e.g., "press the *new invoice* button") and the checking activities that must be performed on the application to determine the correctness of the response of the system (e.g., "check that a form with five tabs and all empty fields is shown") in different columns.

The output of test case generation tools can be much more appreciated if the result of their activity can be expressed as the coverage of items in the test plan. When the test plans are stored in a semi-structured format, like the one we described, there is indeed the opportunity to achieve this goal. Test engineers could exploit a coverage report based on the test plan to quickly identify the scenarios that have been already validated automatically, and the ones that require manual intervention. Finally, any reported failure could be easily interpreted in terms of the test scenario that was executed.

4 CONCLUSIONS

Automatically testing commercial applications raises several challenges that require sophisticated and dedicated solutions to be addressed. In this paper, we discussed multiple opportunities that can be exploited to effectively address these challenges and make automatic system testing better suited for commercial applications.

We are currently actively working on extending ABT based on the ideas described in this paper.

Acknowledgements. This work has been partially supported by the H2020 ERC CoG project Learn (n. 646867), the ERC PoC project AST (n. 824939) and the PRIN research project GAUSS (Contract 2015KWREMX).

REFERENCES

[1] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using gui ripping for automated testing of android applications. In *Proceedings of the International Conference on Automated Software Engineering*, ASE '12, 2012.
[2] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. MobiGUITAR ïŁ¡ a tool for automated model-based testing of mobile apps. *IEEE Software*, 32(5):53–59, 2014.
[3] S. Arlt, A. Podelski, C. Bertolini, M. Schaf, I. Banerjee, and A. M. Memon. Lightweight static analysis for gui testing. In *Proceedings of the International Symposium on Software Reliability Engineering*, ISSRE '12. IEEE Computer Society, 2012.
[4] S. Artzi, J. Dolby, S. Jensen, A. Moller, and F. Tip. A framework for automated testing of javascript web applications. In *proceedings of the International Conference on Software Engineering*, 2011.
[5] M. Baluda, G. Denaro, and M. Pezzè. Bidirectional symbolic analysis for effective branch testing. *IEEE Transactions on Software Engineering*, 42(5):403–426, 2015.
[6] G. Becce, L. Mariani, O. Riganelli, and M. Santoro. Extracting widget descriptions from guis. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, LNCS. Springer, 2012.
[7] P. Braione, G. Denaro, A. Mattavelli, and M. Pezzè. Combining symbolic execution and search-based testing for programs with complex heap inputs. In *Proceedings of the 26th International Symposium on Software Testing and Analysis*, ISSTA '17.
[8] P. Braione, G. Denaro, A. Mattavelli, M. Vivanti, and A. Muhammad. Software testing with code-based test generators: Data and lessons learned from a case study with an industrial software component. *Software Quality Journal*, 22, 2014.
[9] P. Braione, G. Denaro, and M. Pezzè. JBSE: A symbolic executor for java programs with complex heap inputs. In *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '16, pages 1018–1022, 2016.
[10] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013.
[11] F. Gross, G. Fraser, and A. Zeller. Search-based system testing: high coverage, no false alarms. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2012.
[12] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (FSE)*, 2013.
[13] A. Marchetto, P. Tonella, and F. Ricca. State-based testing of Ajax web applications. In *proceedings of the International Conference on Software Testing, Verification, and Validation*. IEEE, 2008.
[14] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro. Autoblacktest: a tool for automatic black-box testing. In *ICSE '11: Proceedings of the 33rd International Conference on Software Engineering*, pages 1013–1015, 2011.
[15] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro. Automatic testing of gui-based applications. *Software Testing, Verification and Reliability*, 24(5):341–366, 2014.
[16] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro. Link: Exploiting the web of data to generate test inputs. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '14, pages 373–384. ACM, 2014.
[17] L. Mariani, M. Pezzè, and D. Zuddas. Augusto: Exploiting popular functionalities for the generation of semantic gui tests with oracles. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2018.
[18] A. M. Memon and Q. Xie. Studying the fault-detection effectiveness of gui test cases for rapidly evolving software. *IEEE Transactions on Software Engineering*, 31(10):884–896, 2005.
[19] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)*, 6(1):3:1–3:30, 2012.
[20] A. Mesbah, A. van Deursen, and D. Roest. Invariant-based automatic testing of modern web applications. *IEEE Transactions on Software Engineering (TSE)*, 38(1):35–53, Jan. 2012.
[21] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of the International Conference on Software Engineering*, ICSE '07, pages 75–84. ACM, 2007.
[22] N. Tillmann and J. de Halleux. Pex: White box test generation for .NET. In *Proceedings of the International Conference on Tests and Proofs*, TAP '08, 2008.
[23] Y. Xun, M. Cohen, and M. A.M. Gui interaction testing: Incorporating event context. *IEEE Transactions on Software Engineering*, 37(4):559–574, 2011.
[24] W. Yang, M. R. Prasad, and T. Xie. A grey-box approach for automated gui-model generation of mobile applications. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2013.
[25] X. Yuan and A. M. Memon. Generating event sequence-based test cases using GUI run-time state feedback. *IEEE Transactions on Software Engineering (TSE)*, 36(1):81–95, January/February 2010.