Can we automate away the main challenges of end-to-end testing?

Renaud Rwemalika, Marinos Kintis, Mike Papadakis, Yves Le Traon
Interdisciplinary Centre for Security, Reliability and Trust (SnT)
University of Luxembourg
Luxembourg, Luxembourg
{renaud.rwemalika, marinos.kintis, michail.papadakis, yves.letraon}@uni.lu

***Abstract*—Agile methodologies enable companies to drastically increase software release pace and reduce time-to-market. In a rapidly changing environment, testing becomes a cornerstone of the software development process, guarding the system code base from the insertion of faults. To cater for this, many companies are migrating manual end-to-end tests to automated ones. This migration introduces several challenges to the practitioners. These challenges relate to difficulties in the creation of the automated tests, their maintenance and the evolution of the test code base. In this position paper, we discuss our preliminary results on such challenges and present two potential solutions to these problems, focusing on keyword-driven end-to-end tests. Our solutions leverage existing software artifacts, namely the test suite and an automatically-created model of the system under test, to support the evolution of keyword-driven test suites.**

***Index Terms*—end-to-end testing, keyword-driven testing, automatic test generation, automatic test repair**

I. Problem and motivation

In continuous integration and agile methodologies, rapid feedback is required for everyone involved and for any code change no matter whether it is a small or a full product release. A crucial part of this feedback comes from automated tests and checks that are responsible for code integrity and reliability. Agile methodologies enabled companies to drastically increase software release pace and reduce time-to-market. Unfortunately, these benefits are accompanied by a considerable increase in testing costs: more tests need to be written and executed more frequently.

End-to-end testing aims at ensuring that the system under test (SUT) is performing as designed from start to finish. Traditionally, these tests are generated manually and designed as a set of usage scenarios describing the manual steps to be performed [1]. However, performing manual end-to-end testing is tedious, time-consuming and error-prone. Furthermore, manual tests hinder the continuous integration (CI) pipeline by creating a blocking point.

To mitigate these problems, companies focus on automating the test execution and test reporting in order to avoid the CI pipeline bottleneck. However, the main test activities, i.e., test case design, test scripting and test maintenance, are challenging and are mainly performed manually [2].

Enterprise-grade applications typically involve thousands of tests [3], which are derived from the specifications and are usually written in a natural language. Converting manual tests to automated tests is a huge amount of work. Consequently, as projects lack resources, *only a fraction of manual tests end up being automated*. Evidently, this compromises the quality of testing.

Even when tests are automated, another problem arises: *tests tend to break easily when the SUT changes*, resulting in a need for test maintenance [4]. However, large systems can involve hundreds of tests, each one composed of multiple steps, making this task cumbersome, especially in the case of not modularized tests [5]. Indeed, literature suggests that simple modifications to the SUT can result in a 30 to 70 percent changes to the tests [6]–[8].

Researchers have introduced many methods that automatically create tests using functional requirements [3], [9]–[15]. However, these methods make strong assumptions that are not applicable in many industrial contexts, i.e., they require having a model of the SUT or impose a specific formalization of the requirements. Building a model for non-trivial applications can be time-consuming and error-prone while using the formalization of the requirements imposed by a tool might not be compatible with the development workflow and company practices. Furthermore, understanding the requirements and translating them in an automated process might be a challenge in itself [13].

For quality assurance (QA) teams, a common industry practice is to use light-weight techniques for automating end-to-end testing such as Keyword-Driven Testing [16] instead of model-based testing and formal methods [17]. Indeed, QA teams often do not have the knowledge to work with complex modeling tools or write complex test scripts in full-fledged programming languages. Thus, in this position paper we focus on the challenges of creating and maintaining keyword-driven test suites and propose solutions for automated test creation and repair with the aim of reducing the overall cost of keyword-driven test suite evolution.

Based on our experience working almost one year with our industrial partners on this topic, we have identified the following challenges that practitioners are facing:

1) *Translating requirements to automated tests can be challenging.* Usually practitioners do not work with formal requirements but rather with informal ones that are

written in natural language. The level of abstraction in the requirements affects the automation effort required. Analogous problems can arise when converting manual tests to automated ones. **Practitioners would benefit from automated techniques that support end-to-end test automation.**

2) *Automated tests tend to break often.* End-to-end tests are fragile to the SUT changes. Thus, the evolution of the SUT plays an important role in the maintenance cost of these tests. If test maintenance becomes extremely high, the risk of abandoning automated end-to-end testing becomes equally high, as well. **Practitioners need automated techniques to support the evolution of the tests.**
3) *It is not clear when tests should be automated.* Practitioners face the dilemma of when to automate: automating early in the SUT development lifecycle can increase the test maintenance cost; automating late can hinder the agile workflow. **Practitioners need automated techniques to repair the tests designed early in the SUT development lifecycle.**
4) *The complexity of the test code base can hinder the creation and maintenance of the tests.* As with any software artifact, a bigger test code base results in increased difficulty in maintaining it, potentially leading to test code duplication [18], test smells [19] or even dead test code. **Practitioners require effective automated techniques that can identify problems in the test code base.**

Considering the above-mentioned challenges, it becomes clear that there is a need to aid practitioners in creating and maintaining keyword-driven tests. Towards this end, we propose two solutions that leverage the structure of Keyword-Driven Testing to extract semantic information from the tests and support their evolution.

II. Keyword-Driven Testing

Keyword-Driven Testing aims at separating test design from technical implementation, hence limiting exposure to unnecessary details. Keyword-Driven Testing advocates that this separation of concerns allows tests to be written more easily, to create more maintainable tests and enables experts from different fields and backgrounds to work together at different levels of abstraction using named procedures called keywords.

Each keyword is composed of a set of keywords, enabling an end-to-end test to be expressed in a hierarchical structure. Keywords at the top of the hierarchy are typically describing the business requirements (behavioral keywords) while keywords at the bottom of the hierarchy define the technical implementation (technical keywords) to interact with the SUT.

Figure 1 shows an example of a test called "Transfer money to other bank" expressed as a rooted ordered tree. The root of the tree (purple rectangle) is the test that is executed by calling all the children nodes in a depth first manner. The intermediary nodes (beige rectangles) are called User Keywords since they are created by the tester. Finally, the leaf nodes (dark green rectangles) are Library Keywords. These keywords are executed by a driver. Library keywords are responsible for either defining the control flow of the tests or interacting with the SUT.

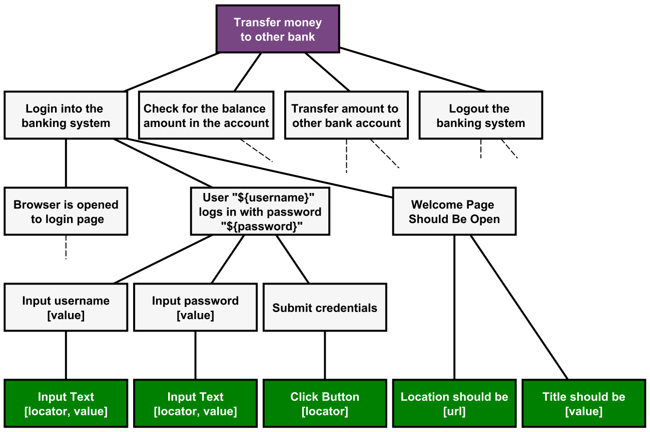
Transfer money to other bank
Login into the banking system
Check for the balance amount in the account
Transfer amount to other bank account
Logout the banking system
Browser is opened to login page
User "${username}" logs in with password "${password}"
Welcome Page Should Be Open
Input username [value]
Input password [value]
Submit credentials
Input Text [locator, value]
Input Text [locator, value]
Click Button [locator]
Location should be [url]
Title should be [value]

Fig. 1. Tree representation of a keyword-driven test.

A. Preliminary results on Keyword-Driven Testing

The first results of our ongoing analysis show that by separating concerns in keywords, and by allowing keyword reuse, Keyword-Driven Testing can reduce the number of changes performed during the test suite evolution by up to 70% (in comparison with techniques such as capture/replay). However, the cost of creation and maintenance of keyword-driven tests remains high. Indeed, when a new feature is created, new keywords need to be created and existing ones might need refactoring.

Furthermore, slight modifications to the GUI of the SUT will break low-level, technical keywords. Our experiments show that technical keywords responsible for *synchronizing* the tests (waiting for an element to appear on the screen, timeouts, etc.) and *locators* (finding and accessing GUI elements) are subject to most of the changes during maintenance activities. These results suggest that while requirements remain the same, technical implementations cause the tests to break and be maintained.

III. Research Solutions

A. Towards Automated Test Repair

As discussed in the previous section, during test suite evolution technical keywords are more prone to change than more abstract ones. This fact indicates that a large proportion of test suite maintenance is generated by software changes that do not affect the behavior of the SUT (in terms of its requirements), but only its technical details. This is not restricted to keyword-driven tests: many techniques such as capture/replay and programmable scripts are susceptible to such changes. In this position paper we intend to tackle the automatic repair of such changes and, as a consequence, reduce the maintenance cost of keyword-driven test suite evolution.

Our approach is partially influenced by the work of Gao et al. [20]. In their work, they use the event-flow graph [21] of the SUT obtained by GUI ripping [22] before and after the SUT changes, referred to as versions V1 and V2 of the SUT respectively. Next, they map the end-to-end tests to V1 and rip the application to find a potential new path in order for the tests to comply to V2 of the event-flow graph. This technique presents two main drawbacks.

First, the creation of the event-flow graph is done in a depth-first manner. This process can be time-consuming, and in case of transactions or text inputs, ripping the application might become impossible or incomplete. In our approach, we intend to leverage the hierarchical structure of Keyword-Driven Testing and limit the GUI exploration space, thus, ripping only a subset of the application, using only children from an ancestor node. Indeed, each keyword represents a unit, therefore, the investigation of a new path can be bound to some specific keywords, unlike in traditional programmable tests. Second, the approach proposed does not take into account oracles and requires user input to decide whether or not the repaired tests meet the requirements. We intend to use the information contained in the structure of keywords to restrict the domain of exploration to find test candidates, hence eliminating repairs that would violate test oracles.

In a nutshell our approach is composed of the following steps:

1) Language model: By construction, Keyword-Driven Testing offers a mapping between more abstract concepts (requirements) and their technical representation (actions on the SUT). For instance, in Figure 1, we can see that “Browser is opened to login page” is a synonym of “Open Browser To Login Page” (both keywords perform the same set of actions on the SUT). Similarly, we can see that “Welcome Page Should Be Open” performs two assertion on the system represented by the leaf nodes of the subtree (“Location should be” and “Title should be”). Given a large corpus, we can leverage this property and build a language model that contains semantic information for keywords. This information include the keyword’s name (e.g., “Browser is opened to login page”), its children and its leaf nodes.

2) Base event-flow graph: During execution of the test suite, each state of the application and actions on the SUT are saved in an event-flow graph. While the graph is not a complete model of the SUT, it holds as a baseline on how the SUT is supposed to react during normal test execution.

3) GUI ripping: During the GUI ripping phase, paths are computed using guided GUI ripping. Ripping the GUI allows to find alternative valid paths by extending the base event-flow graph. As explained earlier, the exploration domain is restricted, allowing changes only in the children keywords of an ancestor of the keyword needs maintenance, e.g., failed during the execution of the end-to-end test. The challenge is to define which ancestor to select. Indeed, choosing an ancestor too far might give the algorithm to much latitude and violate the constraints, while choosing an ancestor too close might restrict the exploration domain too much, therefore, excluding the valid repair path. We propose to start with a highly localized domain and extend it if no potential candidate is found. Empirical results will give us a better indication of the level to which the search domain can be extended.

4) Validate candidate paths: Using parts-of-speech tagging and natural language processing (NLP) against the corpus of keywords and elements of the SUT, the algorithm can target valid candidates for repair. The goal of this step is to restrict the candidate pool by creating a constraint model based on the semantic of the keywords’ hierarchy.

5) Suggest best repair candidate: Find and suggest the path with the minimal changes from V1 as the best repair candidate, or flag the test as not repairable if no alternative paths are found in V2 of the SUT. For non-repairable tests, we will investigate whether a semi-automated solution is beneficial, i.e., we will turn to the testers and use his/her guidance to guide the search. Such a solution has been employed to several other approaches with success, e.g., [20], [23], [24].

Note that while steps 1 and 2 are always performed during test suite execution, steps 3 to 5 are only executed once a step fails.

B. Towards Automated Test Creation

In the previous section, we presented how we intend to automatically repair keyword-driven tests using a language model generated from the test suite and the event-flow graph of the SUT. As we discussed, we restricted the exploration phase of the SUT in order to reduce the probability of deviating from the original tests (the tests of version V1 of the SUT). In this section, we explore the possibility of creating tests given the requirements, *i.e.* behavioral keywords. For instance, if we take Figure 1, given that we have the keywords “Login into the banking system”, “Check for the balance amount in the account”, “Transfer amount to other bank account” and “Logout the banking system”, can we generate the rest of the tree? To answer this question, we propose the following approach:

1) Matching similar keywords: The assumption behind this step is that similar keywords (similar name) should behave in a similar way (similar tree structure). The main challenge of this step is extracting the differences between similar keywords and map them to changes in children nodes. Using the language model from Section III-A should allow us to build the similarity metric and give indications on the meaning of the differences.

2) Exploring the SUT for new behaviors: New keywords might not have any similar keywords in the corpus, while other might have only a partial resolution of their structure. For such keywords, we explore the SUT using GUI elements to find meaning for a specific keyword. For instance, for the keyword “User $username logs in with password $password”, a form named “Login” with two input field having labels “user” and “password” and a validation button is a good candidate. The keyword would then be composed of three sub keywords: input username, input password and validate form. The assumption under which this technique works is that the test suite uses similar naming conventions as the SUT.

3) Keyword validation: Finally, proceeding in a similar fashion as in Section III-A, each test is executed against the SUT and the GUI is ripped in case the sequence of actions is not valid. Unlike the repair phase, the only ground truth known is the step provided by the user as input. The process cannot change input keywords but doesn't have restriction in the changes applied to keywords generated in phase 1 and 2 as long as they respect the language model. In the case a keyword is similar to the keyword being generated, the new keyword should behave in abstract similar fashion. For instance, two assertions can be derived for a keyword called "Login Page Should be Open" based on the example from Figure 1 using the knowledge derived from the "Welcome Page Should be Open" keyword.

C. Limitations

While our approach intends to tackle the limitations of existing techniques (*e.g.* a better respect of the oracles using the language model to limit the set of repair candidates for automatic test repair and the relaxation of requirements formatting for test generation), it still has some limitations.

Since the language model is solely based on user keywords, in order to have a robust model, we need a large corpus of keywords, so that we can learn from it. Furthermore, automating test creation and repair for completely new features might be harder to perform. This point is partially addressed by allowing for an extended exploration space of the SUT (using information contained in its GUI). Empirical results should provide us better indications on the performance of the language model based on the size of the corpus and the heterogeneity of the features tested.

Another limitation could arise during the ripping phase. While exploring new paths, some actions might be irreversible (*e.g.* create an account, remove an item, etc.) and thus change the available paths. Indeed, end-to-end tests often involve some cleaning steps or teardown at the end of their execution to clean the environment. Under such conditions, GUI ripping might be inefficient or even impossible.

IV. Related work

A. Automated test repair

Choudhary et al. [25] propose a tool called WATER to automatically fix tests in web applications using differential testing. Their technique is limited to four types of changes.

Chang et al. [24] propose a tool called CHATEM to automatically repair android tests. CHATEM extends the previous work of the authors, based on the ATOM tool [23] and is built upon Robot Framework, a Keyword-Driven Testing framework, and Apium. This particular technique takes as input two event sequence models (similar to representation of the SUT as event-flow graph but with a slightly different formalization) one for the previous and the current version of the SUT and computes the changes between them and updates the tests accordingly. In our approach, we do not require the user to provide a model of the SUT. Instead, we use the information available in the test suite and GUI ripping to repair broken tests. However, unlike ATOM, our approach requires the test suite to be executed.

Gao et al. [20] propose a technique to repair low level scripts using a tool called SITAR. In their approach, the authors first rip the application to generate an event-flow graph of the SUT and map the test script to the graph. In the event a match is not found, the script needs to be repaired. In our approach, we want to repair tests and overcome the limitation of having to rip the entire SUT to do so. Our technique attempts to provide a more guided GUI ripping based on the knowledge provided by the keyword-driven test suite.

B. Test generation

We are not the first to try to convert requirements written in natural language into test cases. In the literature, we can find a large body of work [3], [9]–[15]. However, most of these techniques require a (semi-)formal representation of the requirements [9], [10], [12], [13], [15] and/or manual generation of models for the SUT [12] and/or manual annotation of the requirements [11]. In contrast, we try to use available software artifacts (test suite, SUT, execution results, etc.) to automatically build the knowledge base required for test generation.

Other similar studies include the one of Blasi et al. [26] who propose a tool called jDoctor to automatically generate Java Unit Tests based on JavaDoc. The tool combines pattern-matching with natural language processing and adds a notion of semantic similarity allowing to use terms that are syntactically different from the methods in the application to be used in their grammar. The grammar is used to map sentences from the JavaDoc to Java methods to generate unit tests. We plan on using a similar approach to create our language model and map keywords to GUI elements and API entry points.

V. Conclusion

End-to-end testing is used by companies to evaluate whether the system under test conforms to its requirements. End-to-end tests are typically performed manually which hinders the adoption of agile methodologies and increases the time required to find failures. The automation of such tests is desirable but unfortunately very expensive. Even when such tests are fully automated the cost of their maintenance can be prohibitive. In this position paper, we discuss several challenges faced by practitioners during end-to-end test suite evolution and present two potential solutions to the problems of test creation and maintenance, focusing on keyword-driven, end-to-end test suites. We argue that by leveraging the hierarchical structure of such test suites, we can create a language model providing semantic information for each keyword. We intend to use this understanding of the test suite to tackle the aforementioned problems. The evaluation of the solution will be based on real industrial data, so that we can investigate the true benefits and drawbacks of the techniques.

References

[1] V. Garousi and M. V. Mäntylä, "When and what to automate in software testing? A multi-vocal literature review," *Information and Software Technology*, vol. 76, pp. 92–117, aug 2016.

[2] V. Garousi and F. Elberzhager, "Test Automation: Not Just for Test Execution," *IEEE Software*, vol. 34, no. 2, pp. 90–96, mar 2017.

[3] S. Thummalapenta, S. Sinha, N. Singhania, and S. Chandra, "Automating test automation," in *2012 34th International Conference on Software Engineering (ICSE)*, no. June. IEEE, jun 2012, pp. 881–891.

[4] L. S. Pinto, S. Sinha, and A. Orso, "Understanding myths and realities of test-suite evolution," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12*, vol. 1. New York, New York, USA: ACM Press, 2012, p. 1.

[5] R. Yandrapally, G. Sridhara, and S. Sinha, "Automated Modularization of GUI Test Cases," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, may 2015, pp. 44–54.

[6] A. M. Memon and M. L. Soffa, "Regression testing of GUIs," in *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering - ESEC/FSE '03*, vol. 28, no. 5. New York, New York, USA: ACM Press, 2003, p. 118.

[7] M. Grechanik, Q. Xie, and C. Fu, "Maintaining and evolving GUI-directed test scripts," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 408–418.

[8] H. Pirzadeh, S. Shanian, and F. Davari, "A Novel Framework for Creating User Interface Level Tests Resistant to Refactoring of Web Applications," in *2014 9th International Conference on the Quality of Information and Communications Technology*. IEEE, sep 2014, pp. 268–273.

[9] C. Nebut, F. Fleurey, Y. Le Traon, and J.-M. Jezequel, "Automatic test generation: a use case driven approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 140–155, mar 2006.

[10] C. M. Kirchsteiger, J. Grinschgl, C. Trummer, C. Steger, R. Weiss, and M. Pistauer, "Automatic Test Generation From Semi-formal Specifications for Functional Verification of System-on-Chip Designs," in *2008 2nd Annual IEEE Systems Conference*. IEEE, apr 2008, pp. 1–8.

[11] C.-Y. Hsieh, C.-H. Tsai, and Y. C. Cheng, "Test-Duo: A framework for generating and executing automated acceptance tests from use cases," in *2013 8th International Workshop on Automation of Software Test (AST)*. IEEE, may 2013, pp. 89–92.

[12] C. Wang, F. Pastore, A. Goknil, L. Briand, and Z. Iqbal, "Automatic generation of system test cases from use case specifications," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis - ISSTA 2015*. New York, New York, USA: ACM Press, 2015, pp. 385–396.

[13] S. H. Jensen, S. Thummalapenta, S. Sinha, and S. Chandra, "Test Generation from Business Rules," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, apr 2015, pp. 1–10.

[14] T. Yue, S. Ali, and M. Zhang, "RTCM: a natural language based, automated, and practical test case generation framework," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis - ISSTA 2015*. New York, New York, USA: ACM Press, 2015, pp. 397–408.

[15] I. L. Araújo, I. S. Santos, J. B. F. Filho, R. M. C. Andrade, and P. S. Neto, "Generating test cases and procedures from use cases in dynamic software product lines," in *Proceedings of the Symposium on Applied Computing - SAC '17*. New York, New York, USA: ACM Press, 2017, pp. 1296–1301.

[16] Jingfan Tang, Xiaohua Cao, and A. Ma, "Towards adaptive framework of keyword driven automation testing," in *2008 IEEE International Conference on Automation and Logistics*, no. September. IEEE, sep 2008, pp. 1631–1636.

[17] I. Banerjee, B. Nguyen, V. Garousi, and A. Memon, "Graphical user interface (GUI) testing: Systematic mapping and repository," *Information and Software Technology*, vol. 55, no. 10, pp. 1679–1694, oct 2013.

[18] T. Lavoie, M. Mrineau, E. Merlo, and P. Potvin, "A case study of ttcn-3 test scripts clone analysis in an industrial telecommunication setting," *Information and Software Technology*, vol. 87, pp. 32 – 45, 2017.

[19] V. Garousi and B. Kk, "Smells in software test code: A survey of knowledge in industry and academia," *Journal of Systems and Software*, vol. 138, pp. 52 – 81, 2018.

[20] Z. Gao, Z. Chen, Y. Zou, and A. M. Memon, "SITAR: GUI Test Script Repair," *IEEE Transactions on Software Engineering*, vol. 42, no. 2, pp. 170–186, feb 2016.

[21] A. M. Memon, "An event-flow model of GUI-based applications for testing," *Software Testing, Verification and Reliability*, vol. 17, no. 3, pp. 137–157, sep 2007.

[22] A. Memon, I. Banerjee, and A. Nagarajan, "GUI ripping: reverse engineering of graphical user interfaces for testing," in *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings.*, vol. 2003-Janua. IEEE, 2003, pp. 260–269.

[23] X. Li, N. Chang, Y. Wang, H. Huang, Y. Pei, L. Wang, and X. Li, "ATOM: Automatic Maintenance of GUI Test Scripts for Evolving Mobile Applications," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, mar 2017, pp. 161–171.

[24] N. Chang, L. Wang, Y. Pei, S. K. Mondal, and X. Li, "Change-Based Test Script Maintenance for Android Apps," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, jul 2018, pp. 215–225.

[25] S. R. Choudhary, D. Zhao, H. Versee, and A. Orso, "WATER," in *Proceedings of the First International Workshop on End-to-End Test Script Engineering - ETSE '11*. New York, New York, USA: ACM Press, 2011, pp. 24–29.

[26] A. Blasi, A. Goffi, K. Kuznetsov, A. Gorla, M. D. Ernst, M. Pezzè, and S. D. Castellanos, "Translating code comments to procedure specifications," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis - ISSTA 2018*. New York, New York, USA: ACM Press, 2018, pp. 242–253.