© IEEE

Test Automation Challenges for Application Landscape Frameworks

Nils Wild
Research Group Software Construction
RWTH Aachen University
Aachen, Germany
wild@swc.rwth-aachen.de

Horst Lichter
Research Group Software Construction
RWTH Aachen University
Aachen, Germany
lichter@swc.rwth-aachen.de

Peter Kehren
IVU Traffic Technologies AG
Aachen, Germany
pke@ivu.de

***Abstract*—Test Automation is essential to increase the effectiveness, efficiency and coverage of software testing activities with reasonable effort which is required to keep up in a fast-paced industry with a growing demand for high quality software products. Nevertheless adopting test automation is a difficult task for development organisations. Automating the testing process for application landscape frameworks is especially hard due to the complexity caused by the variety of customer application landscape configurations, used tools and platforms. This complexity often is hard or impossible to reduce due to given domain constraints, e.g. limited resources of embedded systems and different platforms that have to be supported. In addition, existing hardware and software systems in the customer's application landscape require rather flexible solutions and configuration options for different customers. In this paper we present some major challenges adopting test automation for application landscape frameworks, as research shows that this is especially difficult for such highly configurable systems. Furthermore initial ideas on how to cope with these challenges are outlined.**

***Index Terms*—software testing, test automation, application landscape frameworks, highly configurable systems, software quality assurance**

I. Introduction

Software is an integral part of our life and changed how we deal with everyday tasks. Customers got used to frequent updates and releases of new features and eventually expect them [16]. Over the years, development life-cycles had to be shortened to cope with rapid changing requirements and meet these expectations and the need for higher quality software products. Agile development approaches and DevOps practices are the now dominant practices for organizations to deal with those challenges [11]. To ensure the quality of the frequently needed releases, automated testing approaches are mandatory to limit the time effort spend on each release [6]. Testing involves planning, designing, implementing, executing and evaluating test cases. For each of these activities methods and techniques were developed and evaluated in research. Nevertheless it's difficult for companies to adopt these methods and techniques. In this paper we present some major challenges adopting test automation for application landscape frameworks, as research shows that this is especially difficult for such highly configurable systems [9] [13].

The paper is structured as follows: Section II outlines important characteristics of application landscape frameworks. Section III describes the challenges that have to be solved to establish an automated testing process for such frameworks. Furthermore, initial ideas on how to deal with these challenges are presented. Finally, conclusions are given in Section V.

II. Application Landscape Frameworks

Previous studies show that testing strategies are highly dependent on the kind of developed software, the applied development processes and the context and culture of a the development organization [17] [13].

In this paper we focus on *application landscape frameworks*. These frameworks provide services for a specific domain that are used to develop customer specific application landscapes. An *application landscape* is a coherent set of all important information systems, services, building blocks, applications, components and interfaces with regards to business continuity. [12]

As every customer has its own requirements, an application landscape framework needs to be configurable. This variability is often realized by different kinds of configuration mechanisms. Sometimes an application landscape framework is implemented as a software product line, sometimes as a suite of components offering the needed services as a platform.

As the services itself are usually complex and large, they are typically developed by different and rather independent teams, to increase development velocity and decrease coupling. But this comes at a price, as additional effort is needed for coordination between the development teams and when integrating the single services into a new customer specific application landscape. This becomes even more tortuous if different tools and technologies have to be used by the teams due to development constraints - e.g. hardware resources or regulations - and thus can not be homogenized. The complex development setting of an application landscape framework and its customer specific instantiations is sketched in Fig. 1.

In any case, a framework based application landscape development leads to a lot of possible configurations and arrangements and additional complexity for testing the framework itself as well as the created customer specific application landscapes. Hence, tools and techniques are needed to ensure

2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto, Portugal, pp 330-333
DOI 10.1109/ICSTW50294.2020.00059

© IEEE

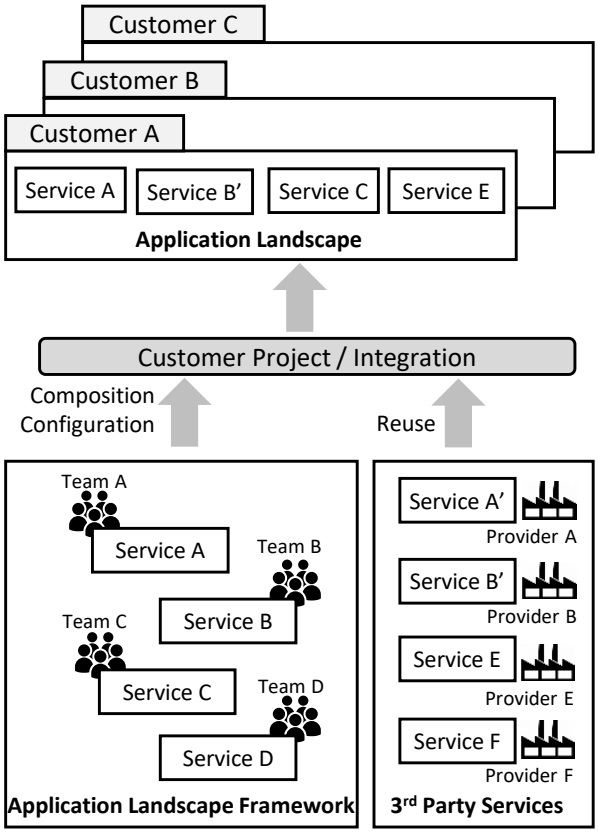

Fig. 1. Framework-based Application Landscapes

the functionality and quality of each of the offered services and their integration in different customer application landscapes. As manual testing is not feasible due to shortened release cycles an automated continuous integration (CI) process is needed. Unfortunately adopting CI is not trivial as it requires changes of the organization, tools, architecture and overall development and testing process [2].

In the following we present challenges, that we identified at a company developing and maintaining an application landscape framework for the domain of public transportation. The provided services span most of the daily tasks of public transportation companies from schedule planning and dispatching to in-vehicle ticket selling and communication between drivers and control centers as well as real-time customer information. Each of the offered services consists of multiple software and possibly hardware components. This diverse environment leads to a huge variety of used tools, languages and platforms throughout the development process and across different teams.

III. Test Automation Challenges

Application landscape frameworks provide development organizations with different challenges regarding quality assurance in general an test automation in particular. Five challenges based on experiences gained with the public transportation application landscape framework will be discussed in further detail.

***C1**: How to manage and maintain the configurations used for customer specific application landscapes?*

Usually, application landscapes are developed in a dedicated customer project using the services of the framework (see Fig. 1). To fulfil the customer's requirements, offered services need to be appropriately composed and configured. Moreover, sometimes offered services are not needed at all or similar services have to be reused from other providers that the customer is operating in its current application landscape. In addition, support for different platforms like Windows, Android or Linux is needed, depending on the hard- and software that is used by the customer.

In order to derive effective test selection criteria the information on what system configurations exist has to be preserved. Acquiring and maintaining that knowledge is hard and requires a lot of effort. Thus the knowledge degrades over time and a complete configuration model is usually nonexistent. To be able to maintain that knowledge with little effort an abstraction layer to set these parameters within the different sources to configure the services could help to provision the customer application landscape and provide a centralized source for all configurations.

Orchestration and infrastructure as code tools provide developers with tools to do so but do not help to reconstruct these configuration models. Furthermore, as certain services within the customer's application landscape can be replaced by third party services, different service landscapes have to be considered and integrated into the configuration model. Graphical user interfaces to model these landscapes and their configuration models could improve their acceptance and usage by customer project engineers.

***C2**: How to improve variation coverage based on the managed configurations?*

Testing such highly configurable application landscapes requires additional effort as executing the same test case under different configurations will lead to different results [14].

To address this problem, methods and tools for software product line testing can be considered, but they rely on central product line and parameterization models [3] [8]. As excessive testing of all possible configurations is not viable a central model of created configurations could be used to test these configurations or derive parameter subsets by applying combinatorial testing methods to efficiently increase variation coverage [10] [8] [3].

Furthermore tools and techniques are needed to test the integration between services of the application landscape framework and ones provides by third parties. Although mocks and service virtualization can help in testing such interactions, most tools to create these still rely on capture and replay mechanisms or explicit definition of the mocks' behavior [15]. Former is not ideal for testing purposes and usually has to be

2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto, Portugal, pp 330-333
DOI 10.1109/ICSTW50294.2020.00059

© IEEE

tailored which adds some disadvantages of the later. On the other hand, manual test data is hard to create as it requires a lot of expert knowledge, which might not be available due to incomplete or even ambiguous documentation, especially for third party services.

***C3**: How to create appropriate mocks with little or incomplete knowledge about the mocked services?*

Two types of information are needed in order to mock a service. What message format is returned by the service and how is the output related to the input? In case of stateless services it is sufficient to model the relationship between input and output data as the input data contains all the information needed to determine the output [4]. On the other hand stateful services rely on their current state. Requests to these services can change their state and thus the output of following requests. This makes it even harder to model these services as experts with that knowledge might not be available, especially for third-party integrations. Recent research tries to solve these problems and mitigate the disadvantages of capture and replay mechanisms as well as manual test data input by learning message formats, relations of input and output as well as behavior specifications instead of fixed values [18] [7] [4].

***C4**: How to constraint teams in order to reduce the effort to distribute necessary test information across teams while preserving their independence?*

Even though the development activities of each team are independent - to decrease coupling and increase development velocity - some coordination for inter-service communication and testing activities is needed. To ensure transparency across teams, the used tools and cultures have to be integrated. One of the main issues experienced to integrate different services developed by different teams is the lack of information about the quality and current status for specific services and corresponding features. To keep the effort for integrating that information low, company wide requirements for organizational and technical aspects can be defined. One possible solution would be to require each team to use specific tools and/or provide their changes at a specific point in time. This solution is not ideal as it limits the freedom of the teams to choose the right tools for their needs, introduces organizational coupling and slows down the overall development process and might not be possible in case of aforementioned domain constraints.

Instead of requiring the teams to use one specific tool one could think about less constrained requirements. Such loosely constrained requirement could be, that each team has to choose tools that support widely adopted formats like xUnit[1] for test execution results. Such constraint largely preserve the independence of teams to choose the right tools, as there are still a bunch of tools to choose from, while reducing the effort for a transparent and standardized test reporting across the company on the other hand. We think a board of team representatives to coordinate these efforts helps to ensure that such constraints do not hinder teams to develop their services and are widely accepted. Criteria and best practices to define appropriate constraints by means of flexibility on the one hand and effort reduction on the other hand is needed, to help companies to find the right balance between these contradicting properties.

***C5**: What data about automated test cases should be gathered and how can it be used to improve the overall quality assurance process? How can we gather that data with as little additional effort as possible?*

To answer these questions we need to think about internal processes within the overall quality assurance process, that have a potential to be improved. An example for such improvement potential is the organization and the effort that is invested into fixing detected defects or faulty test cases. As mentioned before multiple framework services have to be configured appropriately to provide the functionality a customer is requesting. It follows that each functionality of a customer's application landscape is usually dependent on functionalities of one or multiple services. If at least one functionality of one of these services contains a defect the functionality of the customer's application landscape contains that defect as well. This wouldn't be the case for perfectly isolated test cases for these functionalities. As the integration of services can be tested independently of the functionality of the integrating services, the results of the integration test would be independent of those testing the service functionality. Unfortunately, test cases usually are not perfectly isolated and free of dependencies. Having the knowledge about these dependencies could help in improving the ability of handling failed tests or scheduling their execution.

Given the following possible four test result combinations of test cases on the integration and service level, certain treatment instructions to handle failed test cases can be derived to save unnecessary effort to debug and identify defects at the integration level that are just results of defects at the service level.

a) At least one service level test case failed, all related integration test cases succeeded. If this is the case customer's application landscape functionalities related to the service functionality have to be considered faulty even if the integration level tests succeeded. The according service functionality has to be fixed.

b) At least one service level test case failed, at least one related integration test case failed. In this case the corresponding customer's application landscape functionalities have to be considered as faulty as well. In case of perfectly isolated test cases, both the service functionality as well as their integration has to be fixed, as the test results are independent of each other. In case of a dependency the failed integration test case will get a lower prioritization as the integration between services might be fine but there is a chance that the fault is just a result of the faulty service functionality.

c) All service level test cases succeed, at least one related integration test case failed. Even in this case all depending

[1]https://xunit.net/docs/format-xml-v2

2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto, Portugal, pp 330-333
DOI 10.1109/ICSTW50294.2020.00059

© IEEE

functionalities have to be considered as faulty. But in this case the resources can be focused on fixing the integration of the involved services as the chances are high that it is an integration issue. Nevertheless we have to keep in mind that this might still be a faulty service functionality that is not covered by a service level test case.

d) All service level tests succeed, all related integration test cases are successful. This is the ideal case. Given the test suite is sufficient to ensure the quality of the customer's application landscape the tested functionalities can be considered as complete and correct.

Yet again, the knowledge about such relations of test cases degrades over time and it is impractical to invest the effort to regain that knowledge in a big bang approach, when the knowledge was already lost. Instead it has to be acquired automatically and incrementally within the quality assurance process, with little overhead. Coverage and tracing techniques might be utilized to gain that knowledge. To further automate the testing process such knowledge can be used to create backlog items with appropriate priorities. This would improve transparency of test results and trigger explicit actions to fix defects and faulty test cases. As too many of such backlog items would lead to more effort or will not be accepted by the developers, such method has to be as precise as possible with regards to required manual activities. Besides creating these backlog item, encouraging developers to annotate these items with a defect taxonomy could have additional benefits, as research shows that such a taxonomy "provides systematic backup for the design of tests, supports decisions for the allocation of testing resources and is a suitable basis for measuring the product and test quality" [5]. This could also be used to flag flaky tests that could be detected automatically by tools like deflaker [1]. The information gained through annotating these issues could then be used to further improve he precision of the backlog item creation and prioritization. In our opinion making these and other properties of test cases visible can substantially improve the automated testing and quality assurance process.

IV. Conclusions

Test automation is not just about tools and techniques but also has to consider processes and organizational challenges. Successful companies usually don't have one single product but add new services and eventually develop a domain specific application landscape framework. The growing number of services comes with a growing number of teams and probably tools and thus organizational challenges. Even though the development of different services should be decoupled as much as possible to increase development speed, a certain level of transparency across teams is needed when the quality of dedicated customer specific application landscapes has to be ensured, because multiple services work together to provide the required functionality. Many of the benefits of test automation comes from the analysis of the system under test and planning of testing activities. The question is what data should be preserved for analysis in order to improve the testing process.

In this paper we presented five major challenges for test automation of application landscape frameworks and proposed initial ideas to cope with them. All in all the challenges in adopting test automation aren't just about obtaining the needed skills but also about applying these methods and techniques as well as gathering and analyzing the right data about the systems that have to be tested. Based on these findings and insights we will move on in our future research to propose solutions for these challenges step by step.

References

[1] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov. Deflaker: Automatically detecting flaky tests. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 433–444, May 2018.

[2] A. Debbiche, M. Dienér, and R. Berntsson Svensson. Challenges when adopting continuous integration: A case study. In A. Jedlitschka, P. Kuvaja, M. Kuhrmann, T. Männistö, J. Münch, and M. Raatikainen, editors, *Product-Focused Software Process Improvement*, pages 17–32, Cham, 2014. Springer International Publishing.

[3] E. Engström and P. Runeson. Software product line testing – a systematic mapping study. *Information and Software Technology*, 53(1):2 – 13, 2011.

[4] H. F. Enişer and A. Sen. Virtualization of stateful services via machine learning. *Software Quality Journal*, Oct 2019.

[5] M. Felderer and A. Beer. Using defect taxonomies to improve the maturity of the system test process: Results from an industrial case study. In D. Winkler, S. Biffl, and J. Bergsmann, editors, *Software Quality. Increasing Value in Software and Systems Development*, pages 125–146, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[6] V. Garousi and T. Varma. A replicated survey of software testing practices in the canadian province of alberta: What has changed from 2004 to 2009? *Journal of Systems and Software*, 83(11):2251 – 2262, 2010. Interplay between Usability Evaluation and Software Development.

[7] M. A. Hossain, S. Versteeg, J. Han, M. A. Kabir, J. Jiang, and J. Schneider. Mining accurate message formats for service apis. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 266–276, March 2018.

[8] H. Lackner and B.-H. Schlingloff. Chapter four - advances in testing software product lines. volume 107 of *Advances in Computers*, pages 157 – 217. Elsevier, 2017.

[9] B. Lima and J. Faria. A survey on testing distributed and heterogeneous systems: The state of the practice. pages 88–107, 07 2017.

[10] R. E. Lopez-Herrejon, S. Fischer, R. Ramler, and A. Egyed. A first systematic mapping study on combinatorial interaction testing for software product lines. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–10, April 2015.

[11] A. Mann, A. Brown, M. Stahnke, and N. Kersten. State of devops report. Technical report, Puppet, Circle CI, Splunk, 2019.

[12] M. Paauwe. Application landscape definition - dragon1. [online] Available at: https://www.dragon1.com/terms/application-landscape-definition [Accessed 10 Jan. 2020].

[13] D. Parsons, T. Susnjak, and M. Lange. Influences on regression testing strategies in agile software development environments. *Software Quality Journal*, 22(4):717–739, Dec 2014.

[14] X. Qu. Testing of configurable systems. *Advances in Computers*, 89:141–162, 12 2013.

[15] R. G. Renu Rajani. Service virtualization as an enabler for devops. Technical report.

[16] S. Research. State of the connected customer 3rd edition. Technical report, Salesforce, 2019.

[17] J. Rooksby, M. Rouncefield, and I. Sommerville. Testing in the wild: The social and organisational dimensions of real world practice. *Computer Supported Cooperative Work (CSCW)*, 18(5):559, Sep 2009.

[18] S. Versteeg, M. Du, J. Schneider, J. C. Grundy, J. Han, and M. Goyal. Opaque service virtualisation: A practical tool for emulating endpoint systems. *CoRR*, abs/1605.06670, 2016.

2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto, Portugal, pp 330-333
DOI 10.1109/ICSTW50294.2020.00059