Ukwikora: Continuous Inspection for Keyword-Driven Testing

Renaud Rwemalika
University of Luxembourg
Luxembourg
renaud.rwemalika@uni.lu

Marinos Kintis
University of Luxembourg
Luxembourg
marinos.kintis@uni.lu

Mike Papadakis
University of Luxembourg
Luxembourg
michail.papadakis@uni.lu

Yves Le Traon
University of Luxembourg
Luxembourg
yves.letraon@uni.lu

Pierre Lorrach
BGL BNP Paribas
Luxembourg
pierre.lorrach@bgl.lu

ABSTRACT

Automation of acceptance test suites becomes necessary in the context of agile software development practices, which require rapid feedback on the quality of code changes. To this end, companies try to automate their acceptance tests as much as possible. Unfortunately, the growth of the automated test suites, by several automation testers, gives rise to potential test smells, i.e., poorly designed test code, being introduced in the test code base, which in turn may increase the cost of maintaining the code and creating new one. In this paper, we investigate this problem in the context of our industrial partner, BGL BNP Paribas, and introduce Ukwikora, an automated tool that statically analyzes acceptance test suites, enabling the continuous inspection of the test code base. Ukwikora targets code written in the Robot Framework syntax, a popular framework for writing Keyword-Driven tests. Ukwikora has been successfully deployed at BGL BNP Paribas, detecting issues otherwise unknown to the automation testers, such as the presence of duplicated test code, dead test code and dependency issues among the tests. The success of our case study reinforces the need for additional research and tooling for acceptance test suites.

CCS CONCEPTS

• **Software and its engineering → Acceptance testing**.

KEYWORDS

Continuous Inspection, Test Smell, Keyword-Driven Testing, Clones

ACM Reference Format:
Renaud Rwemalika, Marinos Kintis, Mike Papadakis, Yves Le Traon, and Pierre Lorrach. 2019. Ukwikora: Continuous Inspection for Keyword-Driven Testing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3293882.3339003

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISSTA '19, July 15–19, 2019, Beijing, China
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6224-5/19/07…$15.00
https://doi.org/10.1145/3293882.3339003

1 INTRODUCTION

As companies move towards more agile software development practices, the need for rapid and reliable feedback on the quality of every code change becomes of unparalleled importance. To achieve such rapid feedback, companies invest in the creation of automated test suites, across all testing levels, that guard the code base against the insertion of software defects.

Automating acceptance tests (or end-to-end tests), i.e. testing the system as a whole, usually from its graphical or user-facing interface, ensuring that the system meets its requirements, remains a difficult and error prone task. Automated acceptance test suites faces two main challenges: the first one is the creation of the tests themselves, and, secondly, maintaining the test suite as the application under test evolves [1]. Different techniques aim at tackling these issues. Capture/replay approaches, for instance, make the generation of tests easy, at the expense of their maintainability. On the other hand, writing automation scripts comes with a more expensive test creation process but decreased maintenance cost [4].

While the maintenance cost of automation scripts is lower, many factors might hinder the benefits of such approaches, with the most prevalent one being the introduction of test smells, i.e. poorly designed test code, in the test code base. Indeed, previous works[2, 7] suggest that the presence of test smells has a strong negative impact on test code comprehension and maintenance. Unfortunately, tooling allowing automation testers to detect such smells in their acceptance test suites remains scarce.

This paper presents Ukwikora[1], a tool that aims at filling this gap. It inspects test code developed using Robot Framework [5], an automation framework for acceptance testing, utilizing the Keyword-Driven testing (KDT) approach. The reason for focusing on this methodology is that our industrial partner, BGL BNP Paribas, has adopted KDT in its effort to migrate from manual to automated acceptance testing. Thus, our tool focuses on supporting automation testers in creating and maintaining such tests.

More precisely, Ukwikora aims at tackling the lack of static analysis tooling for KDT test suites that automation testers at BGL BNP Paribas, and at other companies, face. In our previous work [6], we analyzed the evolution of such test suites and identified several challenges that impact test maintenance. Specifically, as the KDT test suites evolve, they become a complex software artifact, suffering from the same problems as typical code bases do, e.g., the presence of code duplication and dead code. Ukwikora addresses

[1] Availble at https://github.com/kabinja/ukwikora-inspector

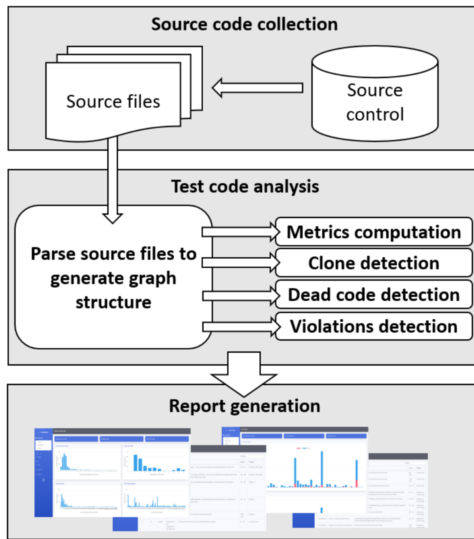

Figure 1: Main components of Ukwikora

such challenges by continuously analyzing the test code for test smells and reporting the results back to the testers.

2 TOOL DESCRIPTION

Ukwikora takes as input the Robot Framework test suites written in the Robot Framework syntax, see *Source code collection* component in Figure 1. Leveraging on the hierarchical structure of the keywords, each test can be represented as an ordered, directed, acyclic *tree* T with nodes $N(T)$ and edges $E(T) \subseteq N(T) \times N(T)$. The nodes $N(T)$ represent keywords and each edge $E(T)$ between two keywords denotes a "step": the parent keyword has the child keyword as a *step*. Since keywords can be used multiple times, by more than one tests, the test suite considered as a whole can be represented as a directed acyclic graph (DAG), G, where each node is defined as one of:

- **Test Cases** Entry point into the graph, called by the framework to be executed. Each test case considered individually is the root of a *tree* T.
- **User Keywords** Internal nodes of the graph, they are created by the user, hence their name, to provide modularity and explainability to the test. Each User Keyword is composed of *steps*, which are calls to a User Keyword or to Library Keyword.
- **Library Keywords** Represent the exit point of the graph to perform concrete actions on the SUT, or the leaves of the tree representation of a single test. Provided by the framework or external libraries, they perform concrete actions, such as interactions with the SUT, logging or assertions.

We use the generated DAG as an intermediate representation of the test code to perform our analysis, see *Test code analysis* component in Figure 1. The tool supports multi projects analysis building a single graph for all the project while annotating each node with the project it belongs to. This representation allows the use of graph theory in order to generate metrics, detect violations,

Table 1: Clone types

Label	Explanation
Type I	Identical keywords except for changes in whitespace, layout and documentation. The clone detection algorithm tags a keyword pair as Type I clones only in the case of an empty set of differences.
Type II	Keywords with a content syntactically identical except for step arguments and return values. The clone detection algorithm tags a pair as Type II clones only if the set contains differences of type *update step arguments* and/or *update step return values*
Type III	Superset of Type II clones, ignoring differences of type *update step*, *add step* and *remove step* if the step belongs to the category *logging*.
Type IV	Keyword performing the same sequence of actions on the SUT, regardless of the internal configuration of the tree. The clone detection algorithm tags a keyword pair as Type IV clones if the sequences of leaf nodes is strictly identical.

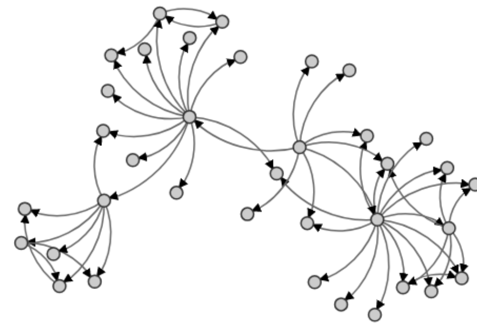

Figure 2: Projects Dependency Graph extracted from Ukwikora

duplicated test code and dead test code. Note that while the tool targets the specific syntax of Robot Framework in this first version, extracting the parsing module would allow to support different syntax for KDT since all analysis are performed on the intermediate representation of the language.

The results of the analysis are presented in a static website composed of a series of dashboards providing information both at the inter- and intra-project level, see *Report generation* component in Figure 1. The content of the dashboards are defined with the help of our partners at BGL BNP Paribas, who are already using the tool in production.

In the remaining of this section, we present the different type of analysis performed on the intermediate representation of the test code base.

2.1 Projects Dependency Graph

To improve modularity and re-usability testers build generic libraries containing low level functionalities that can be reused by different projects. The *project dependency* module of the tool provides a way to visualize dependencies between projects. The need of this visualization originated from what we consider as weakness of the Robot Framework language, namely, the way transitive dependencies are managed. Unlike popular languages like Java, C++ or Python, when a file is loaded by another one, all its dependencies are loaded and made visible as well.

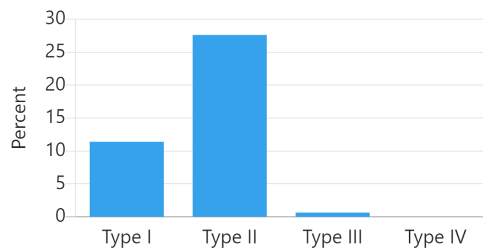

Figure 3: Percentage of duplicated lines extracted from the Summary Page of Ukwikora

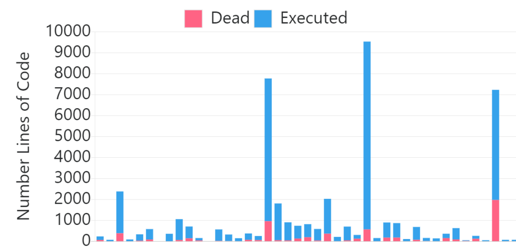

Figure 4: Number of lines of dead test code per project extracted from the Summary Page of Ukwikora

Output. The dependency graph page of the tool presents a complete project dependency graph, allowing the tester to assess the complexity of the test suites as well as detecting issues such as cyclic dependencies. Furthermore, as shown in Figure 2, the graph can become complex and difficult to analyze, therefore in each project page, we can observe a graph of its dependencies as well as projects depending on it.

2.2 Project Statistics

One of the criteria often linked to maintenance effort is the complexity of a project. The more complex a project and its components are, the harder it is to make it evolve and maintain the test code base. Detecting in time the growth of complexity of a project might prevent it from becoming to hard to manage. To this end, we defined a series of four metrics focusing on the complexity of the keywords composing a project.

2.2.1 Keywords Size. The *size* of keyword k (Test Case or User Keyword), is the number of nodes that exist on the subpath(s) from k to all the leaf keywords (Library Keyword).

2.2.2 Keywords Level. The *level* of keyword k (Test Case or User Keyword), is the maximum number of edges that exist on the subpath(s) from k to a leaf keyword (Library Keyword). According to the philosophy of KDT, the higher the value is, the more abstract the keyword. High level keywords define business processes while low level keywords define concrete way of interacting with the SUT.

2.2.3 Keywords Connectivity. The *connectivity* of a keyword is a metric of re-usability among the keywords. Let keyword k belong to a *graph* G, then we calculate the connectivity of k by counting the number of nodes (keywords) in the subpath(s) from the entry points of G to k.

2.2.4 Test Cases Sequence Length. Let a test case TC be the root of a $tree\ T$. The *sequence length* of a test case TC is the number of concrete actions perform by the test, defined as the number of leaf nodes of the $tree\ T$.

2.3 Duplicate Test Code Detection

Our previous work[6] detected a large amount of similar test code, called clones. We observed that clones composed up to 30 percent of the total amount of keywords, indicating that almost one third of the test code written is duplicated. Clones create several difficulties with inconsistencies during test code evolution generating bugs. Furthermore, we observed that about 50 percent of the strictly similar test code (clone Type I) evolve identically, indicating an added cost in the maintenance cost completely avoidable by refactoring.

The clone detection algorithm is based on the fine grain changed algorithm defined in [6] which extract the change set between two keywords. The clone detection, ignoring changes impacting the documentation and the name of the keyword, detects the changes between the two keyword and assign a clone categories defined in Table 1 for each pair of keyword. If a pair does not satisfy any of the definitions, then the keywords are not clones.

Output. In the summary, see Figure 3, and in each of the project pages, statistics about the amount duplicated test code are displayed in order for the tester to evaluate the level of duplication of the test code base. Furthermore, a test code duplication page offers a table containing all the clones, clustered by group of similar keywords. Each row contains information about the location of the keywords duplicated, the type of clones and whether the clones are inter- or intra-application.

2.4 Dead Test Code Detection

During refactoring and evolution activities, the call graph might generate keywords not called by any other. In this work, we define dead test code any User Keyword or Variable never called in a *step*, in other words not connected to a Test Case, and therefore never executed. While building the graph representation of the test suite, the tool keeps track of all the callers and the callees for each keyword are variable. Therefore, detecting detecting dead test code only consist of flagging User Keywords and Variables for which no callers were detected.

Output. For each project, the percentage of dead test code is presented in term of lines of code, see Figure 4. Furthermore, a dead test code page contains a table where each row contains information about the location of the unused User Keywords and Variables in order for the tester to be able to locate them and clean the test code.

2.5 Bad Practices and Error Detection

In its current version the tool allow to detect two types of bad practices and two error types. However, conducting an analysis of the typical change patterns and the execution results, we intend to extend the set, specifically to address issues such as weak *locators* and weak *synchronization* points causing false positive and flackiness.

2.5.1 *Duplicated Keyword Error.*
If two keywords have the same name in the same scope, it causes the program to crash during runtime. However, since keyword resolution is a dynamic process in Robot Framework, the error might be revealed after a lengthy execution time, reducing the velocity of the team to detect and address such bugs. During the analysis, while resolving names, the tool checks for duplicate names.

2.5.2 *Missing Definition.*
Identifies *steps* for which no keyword is associated or a missing variable, therefore generating an edge in the graph with no target node. Upon analysis of the results, we observe that we have false positive for keywords generated dynamically and therefore invisible during static analysis. This violation constitute the only one generating false positive, however, relying to heavily on dynamic loading might hinder the readability of the test suite and therefore, the results are kept and displayed.

2.5.3 *Transitive Dependency Warning.*
We call transitive dependency warning a call to a keyword not defined in the same file or in a direct dependency but in an indirect dependency as explained in Section 2.1.

2.5.4 *Duplicated Variable Warning.*
During the resolution of variable names, if two variables have the same name and are defined in *Variable* blocks, the frameworks loads the first variable it encounters and discards all subsequent definitions. This behavior might lead to unexpected behavior, assigning wrong values to variables.

Output. All these errors and warning are grouped in a single table, giving the level of the violation (warning or error) as well as the location and its reason. Each row of the table contains information about the location of the violation, the cause and its severity.

3 CASE STUDY

In this section, we present the results generated by Ukwikora on the KDT code base at BGL BNP Paribas. The tool is integrated in their production pipeline where the tool runs nightly (via Jenkins), generating a report with its results. First, the tool checks out all the test projects defined in its configuration file, analyzes them and generates a report for the automation testers. The report is then deployed on a server accessible by the members of the team. The KDT test code is composed of 43 projects accounting for 44,521 lines of test code, 452 test cases and 4,448 keywords.

Figure 3 presents the percentage of clones across all projects. With values of 11.4% for Type I clone and 27.6% for Type II clones, we see that there is a high potential for refactoring. This result corroborates previous studies, e.g. [3], once again highlighting the need for better tooling for automation testers. However, we see that Type III and Type IV clones remain low with values of 0.63% and 0.0% respectively. Interviewing the team and analyzing the results, we see that a large portion of the duplicated code is due to the lack of knowledge of the existence of the functionality in the code base, especially in the case of cross-project code duplication.

Figure 4 shows the number of test code lines never executed (dead code) for each of the 43 projects. The values go as high as 27.2% of dead code for a project of more than 7000 lines of test code. The main reason for this, as explained by the testers after analyzing the results, is due to refactoring activities. Indeed, in the absence of static analysis tools, during refactoring deprecated keywords and variables are often kept in fear of breaking production test code.

Ukwikora has been successfully used by both project managers and automation testers. The former appreciate the clear KPIs provided by the report and the latter the information about the locations and causes of the issues reported allowing them to take action if needed. Ukwikora provides a central point of reference and communication between different roles at BGL BNP Paribas which is one of the main goals of every continuous inspection tool.

To conclude our evaluation, we asked the users of Ukwikora which were the main weaknesses or missing features of the tool. According to their answers, the main weaknesses reside in the static nature of the results. Indeed, while the tool provides a snapshot of the state of the code base, the lack of historical data makes it hard to control the evolution of the KPIs.

4 CONCLUSION

This paper presents Ukwikora, a static analysis tool for continuous inspection of Keyword-Driven test suites written in the Robot Framework syntax. It analyzes the test code and provides information about its health and potential test smells introduced. The tool has been successfully deployed at BGL BNP Paribas and has been used in production for a month, providing valuable feedback to project managers and automation experts.

In the future, we plan on addressing the users' requests by adding historical data allowing to complement our analyses. Furthermore, we plan to improve test smell detection. For this purpose, we are conducting a systematic analysis on change patterns in the test code in order to isolate bad practices leading to systematic fixes.

ACKNOWLEDGEMENT

This work is partially funded by Alphonse Weicker Foundation and by the Luxembourg National Research Fund[2].

REFERENCES

[1] Emil Alégroth, Robert Feldt, and Pirjo Kolström. 2016. Maintenance of Automated Test Suites in Industry: An Empirical study on Visual GUI Testing. *Information and Software Technology* 73 (feb 2016), 66–80. https://doi.org/10.1016/j.infsof.2016.01.012 arXiv:1602.01226

[2] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. 2015. Are test smells really harmful? An empirical study. *Empirical Software Engineering* 20, 4 (aug 2015), 1052–1094. https://doi.org/10.1007/s10664-014-9313-0

[3] Thierry Lavoie, Mathieu Mérineau, Ettore Merlo, and Pascal Potvin. 2017. A case study of TTCN-3 test scripts clone analysis in an industrial telecommunication setting. *Information and Software Technology* 87 (jul 2017), 32–45. https://doi.org/10.1016/j.infsof.2017.01.008

[4] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. 2013. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 272–281. https://doi.org/10.1109/WCRE.2013.6671302

[5] Robot RobotFramework. 2019. Introduction. http://robotframework.org/

[6] Renaud Rwemalika, Marinos Kintis, Mike Papadakis, Yves Le Traon, and Pierre Lorrach. 2019. On the Evolution of Keyword-Driven Test Suites. In *12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE Computer Society, Xi'an, 335–345. https://doi.org/10.1109/ICST.2019.00040

[7] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2016. An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*. ACM Press, New York, New York, USA, 4–15. https://doi.org/10.1145/2970276.2970340

[2]references C17/IS/11686509/CODEMATES and AFR PHD 11278802