On the Evolution of Keyword-Driven Test Suites

Renaud Rwemalika*, Marinos Kintis*, Mike Papadakis*,
Yves Le Traon* and Pierre Lorrach†
*SnT, University of Luxembourg, Luxembourg
†BGL BNP Paribas, Luxembourg
*{firstname.surname}@uni.lu
†pierre.lorrach@bgl.lu

***Abstract*—Many companies rely on software testing to verify that their software products meet their requirements. However, test quality and, in particular, the quality of end-to-end testing is relatively hard to achieve. The problem becomes challenging when software evolves, as end-to-end test suites need to adapt and conform to the evolved software. Unfortunately, end-to-end tests are particularly fragile as any change in the application interface, e.g., application flow, location or name of graphical user interface elements, necessitates a change in the tests. This paper presents an industrial case study on the evolution of Keyword-Driven test suites, also known as Keyword-Driven Testing (KDT). Our aim is to demonstrate the problem of test maintenance, identify the benefits of Keyword-Driven Testing and overall improve the understanding of test code evolution (at the acceptance testing level). This information will support the development of automatic techniques, such as test refactoring and repair, and will motivate future research. To this end, we identify, collect and analyze test code changes across the evolution of industrial KDT test suites for a period of eight months. We show that the problem of test maintenance is largely due to test fragility (most commonly-performed changes are due to locator and synchronization issues) and test clones (over 30% of keywords are duplicated). We also show that the better test design of KDT test suites has the potential for drastically reducing (approximately 70%) the number of test code changes required to support software evolution. To further validate our results, we interview testers from BGL BNP Paribas and report their perceptions on the advantages and challenges of keyword-driven testing.**

***Index Terms*—keyword-driven testing, acceptance testing, end-to-end testing, test code evolution, test clones**

I. Introduction

The increased adoption of Agile and DevOps methodologies necessitates quick and reliable test feedback on every code change. In this context, testing has an important role; to ensure code integrity and protect from defects.

Acceptance testing is used by many companies to ensure that the System Under Test (SUT) meets its requirements [1]. These tests are generated manually and, typically, designed as a set of usage scenarios describing the manual steps to be performed [2]. In the context of agile methodologies, with rapid development and change pace, the cost of manual acceptance testing is prohibitive. A solution to this problem is test automation [3].

There are several ways to automate acceptance tests, e.g., Model Based Testing [4] and Capture/Replay [5]. Among them, Keyword-Driven Testing (KDT) is a scripting technique using keywords where each keyword describes a set of actions that are required to perform a specific step. Using keywords, testers can model concepts from the SUT with different levels of abstraction (e.g. "Login" and "Click button"); targeting various domains: Web (Selenium), Android (Apium) and Desktop (Sikuli); allowing different formalisms such as data driven tests or the gherkins syntax.

Our study attempts to answer a fundamental question about KDT: "*What are the practical benefits and challenges of adopting KDT?*". An answer to this question will have a direct impact on practitioners who want to make an informed decision about adopting a test automation technique and to researchers who want to understand the KDT evolution and automate test refactoring and repair.

Practitioners need experience reports on test automation techniques in order to choose and perform the most appropriate one. Interestingly, as the "World Quality Report (2017-18)" [6] shows, practitioners struggle selecting appropriate test automation methods. To ameliorate this, our work presents benefits and challenges of KDT as resulted from the analysis of eight-months of test evolution data from our industrial partner, as well as, findings from the interviews conducted with the corresponding practitioners.

Researchers aim at automating testing. Thus, they need information about the challenges and benefits of KDT testing. Our study sheds light on this manner by identifying and quantifying the practical gains and losses of this practice. We also look into the way KDT test suites are maintained by identifying and categorising the nature of changes performed during the KDT evolution. This information is essential to better understand KDT maintenance and forms the basis of automated test code refactoring and repair techniques.

Literature involves several studies related to the evolution of test code, but they study other testing levels, e.g., the unit level [7]–[10] or focus on different technologies [9]. Experience reports related to end-to-end test evolution are scarce. In this work, we present such a report in the context of KDT paradigm with our partner, BGL BNP Paribas.

Our analysis reveals that test fragility (the sensitivity to SUT evolution) and test clones (keywords with similar test functionality) are the most important problems of KDT, while at the same time KDT offers major opportunities for test code reuse.

We observe that test fragility causes a constant test adaptation (even in response to simple GUI changes) and that test clones are prominent. We show that over 90% of the project keywords are changed and over 30% of the the keywords are clones. We also find that among identical clones, 50% of them co-evolve. These findings indicate the need for automated test repair and refactoring techniques.

On the positive side, our study provides evidence that following the good design practices of KDT (such as the separation of concern and the reusability of keywords) has the potential to reduce the required number of maintenance changes. Our analysis shows that this reduction is approximately 70% demonstrating major benefits of KDT.

Our results also help improving the understanding on the fine-grained changes performed during the evolution of KDT. We provide a taxonomy of test code changes and reveal the presence of test clones caused by the difficulty of selecting appropriate keywords. We believe that the main drawback of KDT lies in the absence of appropriate tooling, allowing to deal with test code growth, navigation and comprehension. Specifically, we show that practitioners agree with the promises of KDT, on the advantages of the separation of concern and the reusability of keywords.

To the best of our knowledge, this is the first study that analyses the evolution of KDT test suites in real, industrial settings. Our study was conducted in partnership with the Quality Assurance team (QA) from the IT department of BGL BNP Paribas. We analysed eight-months of test evolution data which correspond to the first year of transition towards automated acceptance testing using KDT. Overall, we studied 145 KDT test cases, 129 test code commits and 2,578 changes.

This study makes the following contributions::

1) A taxonomy based on a fine grained detection algorithm able to measure evolution of a keyword-driven test suite and a taxonomy of test code changes
2) An analysis of eight-months of KDT test code evolution data in real, industrial settings. To the best of our knowledge, this is the *first research study* that presents results on the evolution of industrial, KDT test suites.
3) Perception from real-world practitioners on the advantages and challenges of adopting KDT and maintaining the corresponding test suites.

II. Background

This section presents KDT, along with illustrative examples and a description of its supporting framework, Robot Framework [11], and the industrial context of our study.

A. Keyword-Driven Testing – KDT

KDT [12] aims at separating test design from technical implementation. Its goal is to limit the exposure to unnecessary details and avoiding duplication. KDT advocates that this separation of concerns allows tests to be written easier, to create more maintainable tests and enables experts from different fields and backgrounds, work together at different levels of abstraction.

```
1  *** Settings ***
2  Library           Selenium2Library
3
4  *** Test Cases ***
5  Valid Login
6      Open browser to login page
7      User "demo" logs in with password "mode"
8      Welcome page should be open
9
10 *** Keywords ***
11 Open Browser To Login Page
12     Open Browser    ${LOGIN URL}    ${BROWSER}
13     Maximize Browser Window
14     Set Selenium Speed    ${DELAY}
15     Login Page Should Be Open
16
17 Login Page Should Be Open
18     Title Should Be    Login Page
19
20 Go To Login Page
21     Go To    ${LOGIN URL}
22     Login Page Should Be Open
23
24 User "${username}" logs in with password "${password}"
25     ...
26
27 Welcome Page Should Be Open
28     ...
29
30 *** Variables ***
31     ${SERVER}         localhost:7272
32     ${BROWSER}        Firefox
33     ${DELAY}          0
34     ${LOGIN URL}      http://${SERVER}/
35     ...
```

Fig. 1. An example of a KDT test.

Figure 1 shows an example of a KDT test. This test, named "Valid Login" (line 5, adopted from the official documentation of Robot Framework), is responsible for validating the correct behaviour of the login form in an imaginary SUT. Lines 6–8 present the "steps" of the tests and, in KDT parlance, they are calls to *keywords*. In turn, these keywords are defined in the respective definition blocks between lines 10 and 28. Each keyword is itself decomposed in a series of steps. Keywords can have *arguments*. For instance, keyword "Open browser" (line 12) takes two arguments, "${LOGIN URL}" and "${BROWSER}". The use of arguments to call keywords allows to further extend the reuse of keywords.

As can be seen from the figure, most part of this fully automated test is written in plain English. This enables the unobstructed collaboration in the creation of the tests between different experts. For instance, a business analyst can write the high-level part of the test (lines 4–8) and an automation expert can implement the remaining part of the test (lines 10-35), adding the technical details to automate the steps.

KDT tests can be represented using a tree structure. Figure 2 shows this structure for the test of Figure 1. The root of the tree (purple rectangle) is the *Test Case* that is executed by calling all the keywords contained. The intermediary nodes (white rectangles) are called *User Keywords* since they are

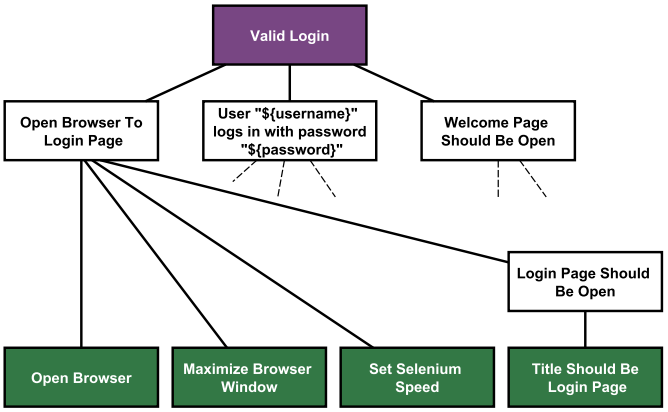


Fig. 2. Tree representation of the "Valid Login" KDT test.

created by the tester. Finally, the leaf nodes (green rectangles) are *Library Keywords*. *Library Keywords* are implemented by the system or an external library and responsible for either controlling the control flow of the tests or interacting with the SUT.

We group keywords into seven categories based on their functionality and present them in Table I. We define a *SYNC* keyword category for keywords dealing with the synchronization between tests and SUT; e.g., a keyword that waits 10 seconds for a GUI element of the SUT to become available. In the rest of the paper we use the term keyword to refer to *User Keywords* unless stated otherwise.

B. Robot Framework

One the tools used for the application of KDT is Robot Framework [11]. Robot Framework is a popular framework used world-wide by major companies, including Nokia, KONE, ABB. This is also the tool adopted by our industrial partner and, thus, used in this work. Robot Framework is an open source tool originally developed by Nokia Networks and is mainly used for acceptance testing. The "Valid Login" KDT test of Figure 1 was written using this framework.

One of the main advantages of Robot Framework is its high modularity. Indeed, Robot Framework is platform-agnostic and thanks to its driver plugin architecture, the core framework does not require any knowledge of the SUT. For instance, in Figure 1, lines 1–2 show that the script is using the external library for Selenium to interact with the SUT. Another advantage of the framework lies in its simple syntax, which makes it easily accessible to testers, regardless of their background.

C. Industrial Context

In this work, we aim at investigating the evolution of KDT test suites at the acceptance testing level based on the industrial practice. To this end, we work together with BGL BNP Paribas that has recently (1 year ago) adopted KDT and uses it in its daily software development work for acceptance testing.

One of the reasons that our partner adopted KDT is that test cases at this testing level were created by different domain experts (business analysts and automation experts) and the adoption of a common language between the experts was

TABLE I
KEYWORD CATEGORIES

Label	Explanation
ACTION	Keyword performing an action on the SUT capable of modifying its state.
ASSERTION	Keyword verifying that a predicate is true at a specific point of test execution
CONTROLFOW	Keyword allowing to modify the control flow of the test execution.
GETTER	Keyword allowing to extract an element from the SUT.
LOGGING	Keyword dumping logs during execution.
SYNC	Keyword relating to the synchronization between the SUT and the tests.
USER	Keyword created by a user.

imperative. All the tests used in our study have been created by a team of 3 testers and 2 business analysts working at BGL BNP Paribas using Robot Framework.

III. Research Questions

In this study, we attempt to answer two main questions about KDT test suites at the acceptance testing level: "*What are the benefits and challenges of adopting KDT?*" and "*What kind of changes are performed during the evolution of a KDT test suite?*". Answers to these questions will enable practitioners to make more informed decisions about KDT and will improve our understanding of KDT test suite evolution. Thus, we pose the following research questions:

RQ1: *What types of test code changes are performed during KDT test suite evolution?*

Analyzing the changes performed by the testers during KDT test suite evolution forms the basis of any automated test refactoring and test repair technique. Although research presents such information in the case of unit testing [7], no previous study has discussed such fine-grained changes in the context of KDT at the acceptance level, to the best of our knowledge.

RQ2: *How complex are the KDT test suites and how does this complexity affect their evolution?*

As mentioned in Section II, one of the advantages of KDT is that it allows the separation of the technical implementation details of test code and its corresponding intention. This fact can lead to test suites having several "levels of abstraction" (cf. Figure 2). To this day, it is not clear how complex the KDT test code is and how this complexity affects its evolution. Answering this question will provide us with a better understanding of the difficulties faced by practitioners when they try to apply KDT and can guide future research directions in ameliorating these problems.

RQ3: *Does code duplication exist in KDT test codebases? What is its impact on the evolution of the test code?*

Similar code fragments are known to exist in source code and test code alike [13]–[16]. In RQ3, we investigate whether KDT codebases contain duplicated test code and how these test clones affect the evolution of the test codebase. Answering this research question is important because if such test clones exist, we need to investigate appropriate techniques to detect them, analyze them and monitor their evolution.

RQ4: *What are the practitioners' perceptions of the benefits and challenges of KDT in practice?*

RQ4 pertains to documenting and analyzing the practitioners' opinion about the advantages and disadvantages of KDT. Such analysis can help other testers to adopt (or not) KDT. Additionally, this research question gives us the opportunity to ask the practitioners' opinion about our results, validating them and understanding them better.

IV. Experimental Design and Analysis

This section presents the experimental design followed to answer the research questions posed. First we formally define the concepts that will be used throughout the rest of the paper next we discuss the industrial project used in our study. Finally, we present the experimental design we followed to answer each RQ.

A. Definitions

Tree: Keywords can be represented as trees, thus, we can define a $tree\ T$ as an ordered, directed, acyclic graph with nodes $N(T)$ and edges $E(T) \subseteq N(T) \times N(T)$. The nodes of the tree denote keywords and each edge between two keywords denotes a "step": the parent keyword has the child keyword as a *step*. For instance, in Figure 2, keyword "Open Browser To Login Page" has four steps: "Open Browser", "Maximize Browser Window", "Set Selenium Speed" and "Login Page Should Be Open". As the tree is ordered, the execution of the steps will follow the order in the tree, from left to right. A node with no parent is a $root$ node that should be defined in the *Test Case* block, while a node with no children is a $leaf$ node and should be a *Library Keyword*.

Keyword Level: The *level* of keyword k, is the maximum number of edges that exist on the subpath(s) from k to a leaf keyword. In Figure 2, "Login Page Should Be Open" is a *level 1* keyword whereas "Open Browser To Login Page" is a *level 2*. *Library Keywords* at the leaves of the tree have a *level 0*.

Keyword Connectivity: Connectivity is a metric of reusability among the keywords. A keyword can belong to several test cases represented as trees: let keyword k belong to trees $T_1, T_2, ...T_n$, i.e. $k \in N(T_1) \cup N(T_2) \cup ... \cup N(T_n)$, then we calculate the connectivity of k by counting the number of nodes (keywords) in the subpath(s) from the root nodes of $T_1, T_2, ...T_n$ to k.

Keyword Churn: Keyword churn is the number of lines of code added, edited or deleted from one version to the next over a period of time.

The last 3 definitions correspond to metrics used in our study. The *keyword level* is used to group keywords having equal levels together. According to the philosophy of KDT, lower level keywords should be more linked to the technical details of the SUT whereas higher level keywords should be more abstract, expressing the functional requirements. The *connectivity* metric expresses the degree to which a keyword is reused and, as a consequence, the degree to which a change to this keyword can impact the test suite. Finally, the *churn* corresponds to the degree to which a keyword is changed during the evolution of the test suite.

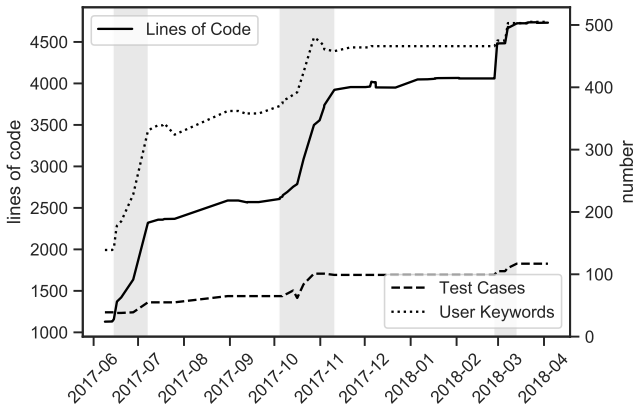


Fig. 3. Evolution of TestSuiteA

B. Industrial Project's Description

The project used in our study, hereafter referred to as *SubjectA* for confidentiality reasons, pertains to all the business activities of our partner. The front-end is a web application implemented in AngularJS, and, the back-end is composed of hundreds of services written in various programming languages. These services are managed by different teams, involving more than 100 developers. The KDT test suite used in our study, referred to as *TestSuiteA*, is developed by 3 testers working at the Quality Assurance (QA) team of our partner and 2 business analysts.

Figure 3 shows the evolution of TestSuiteA across the eight-month period studied. The figure depicts the evolution of the number of *Test Cases* comprising the test suite, the number of *User Keywords* and the lines of code of the test suite. As can be seen, our analysis begins with a test suite of 39 test cases, 139 user keywords and 1129 lines of code and ends with 117 test cases, 505 keywords and 4732 lines of code.

In the time span depicted in Figure 3, we isolated three periods during which we saw an increased test creation activity (shown in grey). After discussing with the QA team, they corroborated that these periods were more focused on test creation and the remaining ones on test maintenance. Thus, we analyze separately these periods (greyed and non-greyed) and refer to them as "Creation" and "Maintenance".

C. Experimental Design

1) Answering RQ1: To answer RQ1, we extract all the changes occurring in the test suite and group them per type of change. The types identified describe an action (insert, update, delete) performed on a code unit element (*User Keyword*, *Test Case*, *Variable*, etc.).

To this end, we extracted the 129 commits from the evolution of TestSuiteA. For each pair of consecutive commits, we gather the changes using a fine grain change algorithm.

Algorithm 1 Element Matcher

Input: $E_1 \subset v_n$, $E_2 \subset v_{n+1}$
Output: final matching set: M_{final}

```
1:  M_final ← ∅
2:  E_1,unmatched ← ∅
3:  for each e_1 ∈ E_1 do
4:      if findMatchFileAndName(e_1, E_2) then
5:          M_final ← M_final ∪ (e_1, e_2)
6:          E_2 ← E_2 − e_2
7:      else
8:          E_1,unmatched ← E_1,unmatched ∪ e_1
9:      end if
10: end for each
11: for each e_1 ∈ E_1,unmatched do
12:     if findMatchFileAndContent(e_1, E_2) then
13:         M_final ← M_final ∪ (e_1, e_2)
14:         E_2 ← E_2 − e_2
15:     else if findMatchNameAndContent(e_1, E_2) then
16:         M_final ← M_final ∪ (e_1, e_2)
17:         E_2 ← E_2 − e_2
18:     else
19:         M_final ← M_final ∪ (e_1, ∅)
20:     end if
21: end for each
22: for each e_2 ∈ E_2 do
23:     M_final ← M_final ∪ (∅, e_2)
24: end for each
```

The algorithm relies on previous, state-of-the-art studies [7], [17]–[19]. In these studies, the authors built abstract syntax trees (ASTs) of Java classes and used tree edit distance algorithms to extract an optimal change path from one tree to the other, with each tree corresponding to a version of the code base.

To detect the changes, the algorithm works in two phases:

1) Finding a match between elements of $v_1 \in V$ and $v_2 \in V$ where V is the set of versions – with one version corresponding to one commit – to come up with a mapping $e_{1n} \rightarrow e_{2n}$ where $e_{mn} \in E_n$ and E_n is the set of elements from v_n.
2) Finding a minimum edit script that transforms V_1 to V_2 given the computed mapping.

Phase 1 is essential to the edit script since the more elements that can be matched, the better the minimum edit script will perform. Phase 2 produces an edit script detecting the basic edit operations *INSERT*, *UPDATE*, *DELETE* for each pair of matched elements.

Listing 1 presents the algorithm used for phase 1 to find an appropriate matching set $E_{1n} \rightarrow E_{2n}$.

- **Lines 3–10**: Search for two elements present in the same file with the same name. If no match is found from $e_1 \in E_1$, it is tagged as unmatched.
- **Lines 11–21**: The same operation is performed, relaxing the constraints. First, at line 12 the name is relaxed, to check if the element was renamed. Then at line 15 the file is relaxed to check if the element was moved to another file. If no suitable match is found for $e_1 \in E_1$, it is matched with a $null$ element and will be consider as a *DELETE* operation in phase 2.
- **Lines 22–24**: Check if there are elements from E_2 that weren't matched, in which case they will be considered as an *INSERT* operation in phase 2.

In phase 2, for each pair of matched elements, we extract the differences. In the case of *User Keyword* and *Test Case*, we use an edit distance algorithm on the sequence of steps which is a modification of the *String-to-String* algorithm presented in [20] using the Levenshtein edit distance.

2) Answering RQ2: For each keyword we extract its level and connectivity, using the tree structure of KDT presented in Section IV-A. We then cluster the keywords by each of these metrics. For each group, we analyze the number of changes performed and the keyword churn. In order to avoid skewing the churn results, we compute the churn during "Creation" and "Maintenance" separately.

Next, we attempt to provide an estimation of the number of changes saved due to the reusability offered by KDT. To answer this, for each tests, we create a sequence of steps executed during execution. Therefore, if a keyword is used twice, the steps from that keyword would appear twice in the sequence. We then compute the changes for each sequence of step execution from one version to the next. The sequences of steps obtained are similar to the ones generated by a classical Capture/Replay (CR) tool. While these results cannot be used to directly compare the maintenance cost of CR and KDT, it provides an estimation of the benefits of reusing keywords.

3) Answering RQ3: To answer this question, we extract similar keywords, also referred to as clones in the literature, and we analyze their evolution. To detect test clones in KDT test suites, we built a clone detection tool specifically designed for KDT test code. The tool is based on the fine grain change algorithm presented in the previous section. We extract the differences between each pair of keywords $k_1, k_2 \in E_n$, ignoring changes related to documentation and *update name* (cf. Table II). For each pair k_1, k_2 we check whether they belong to one of the two types of clones analyzed in our work (definitions adopted from [16]):

- **Type I keyword clones**: identical keywords except for changes in whitespace, layout and documentation. The clone detection tool tags a keyword pair as Type I clones only in the case of an empty set of differences.
- **Type II keyword clones**: keywords with a content syntactically identical except for step arguments. The clone detection tool tags a pair as Type II clones only if the set contains differences of type *update step arguments* and/or *update step return values* from Table II.

Additionally, for each keyword, we extract the set of changes happening during the period under study. From this change list, we define 3 types of keyword evolution:

- **Keyword evolving**: If the change list of a keyword k is not empty, it is defined as evolving.

TABLE II
TYPES AND TOTAL AMOUNT OF CHANGES OVER THE 8-MONTHS STUDY

change type	Creation	Maintenance
insert documentation	**430**	2
insert step	135	62
insert test case	94	12
insert user keyword	394	80
insert variable	286	77
update documentation	106	96
update for loop body	0	0
update for loop condition	0	0
update name	45	6
update step	249	107
update step arguments	105	**144**
update step expression	7	6
update step return values	0	1
update step type	5	3
update variable definition	34	45
delete documentation	0	2
delete step	25	34
delete test case	26	2
delete user keyword	70	38
delete variable	6	23
Total	2017	738

- **Keyword co-evolving**: Among the set of keywords evolving, keywords k_1, k_2 are defined as co-evolving if their changed list is identical.
- **Keyword not evolving**: Keyword k is defined as not evolving if its change list is empty.

Finally, we analyze the relationship between keyword evolution and keyword similarity by cross analysis of categories.

4) Answering RQ4: To answer RQ4, we conduct a series of interviews with the 3 testers working on TestSuiteA at BGL BNP Paribas. The interviews were based on a questionnaire of 14 open-ended questions and were conducted in a semi-formal setting. The aim of these interviews is two-fold. First, we aim at gathering qualitative data to support our observations and, second, we want to collect impressions and anecdotal evidences from the practitioners on their adoption of KDT. The interviews were analyzed based on the two main questions of the study.

V. RESULTS

A. RQ1: Types of Changes during KDT Test Suite Evolution

This research question pertains to the types of changes performed by the testers during TestSuiteA's evolution. The identified types and their total amount are presented in Table II. The first column of the table shows the type of changes as extracted by our change algorithm. The next columns present the total amount of these changes during the *Creation* and *Maintenance* periods (as defined in Section IV-B) over the 8 months of the study.

We see that during *Creation*, the main activities in terms of number of changes is "insert documentation", "insert user keyword", "insert variable" and "update step". The first three types of changes are naturally related to test creation, so this outcome is expected. A more interesting finding is that a lot of effort is devoted in documenting the keywords created. After discussing with the QA team, this effort is justified by the fact that this documentation will prove useful in the case of KDT test failures. "Update step" refers to modifications of steps of existing keywords. The specific kinds of modifications will be further investigated later in the section (see also Figure 4).

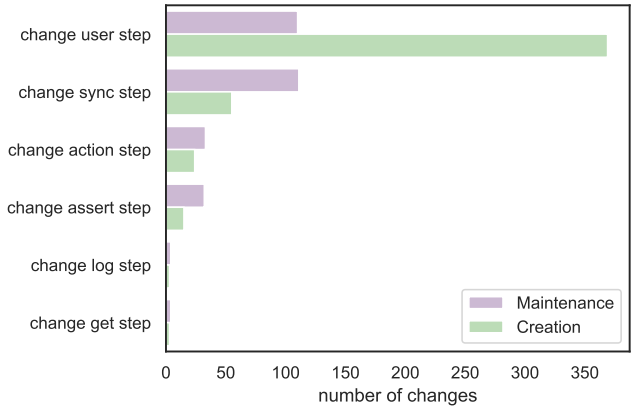


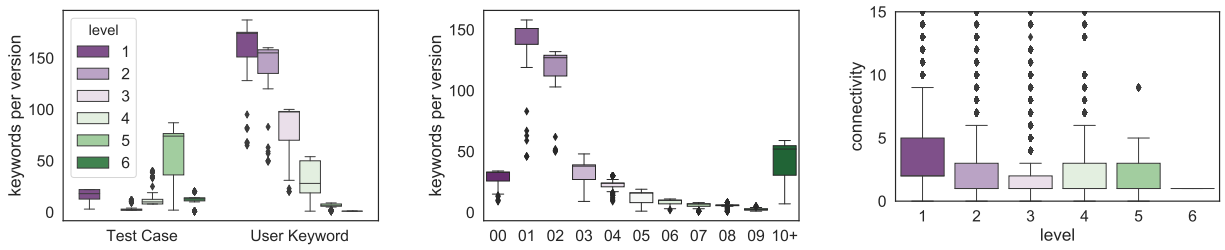
Fig. 4. Total number of step changes per type

During *Maintenance*, the main types of changes performed are the "update step arguments", the "update step", "update documentation" and "insert user keyword". After manually analyzing the changes to the arguments, we found two prevalent categories of commonly-changed arguments: arguments referring to *synchronization* between the SUT and the KDT tests, e.g. wait 3 seconds and arguments referring to *locators*, i.e., ways of locating elements in the GUI interface of the SUT. The arguments of the first category are typically used in the *SYNC* category of Table I, and the latter at keywords of the *ACTION* and *ASSERTION* categories. Our results suggest that keywords belonging to these categories experience a high number of changes. Practioners corroborate those results and motivate those results in RQ4.

> Most changes during KDT test evolution refer to *synchronization* or element *location* changes between the SUT and the test suite and to *assertions* (keyword categories: *SYNC*, *ACTION* & *ASSERTION*).

Apart from "update step arguments", "update step" constitutes one of the most common change for both *Creation* and *Maintenance* periods. To further investigate the nature of these changes, Figure 4 plots the number of changes (x-axis) against the category of the enclosing keywords (y-axis), as presented in Table I with different colors for the periods studied.

As can be seen from the figure, *change user steps* is by far the greatest activity during creation, we see that changes in *synchronization steps* are equally important during maintenance. The interview conducted in RQ4 motivate that finding and explains it by the fact that many keywords are refactored during creation of new tests to become more generic so they can be reused. Another trend is that except for the *user steps*, all other categories evolve more during maintenance. This is due to the same effect as mentioned earlier where changes



(a) Keyword Level (b) Keyword Connectivity (c) Connectivity per level category

Fig. 5. Understanding KDT Test Suite Complexity

in the application cause tests to break. *user steps* are less affected by that effect since they are more abstract and thus less sensitive to trivial application evolution.

B. RQ2: KDT Test Suite Complexity and Evolution

The results for RQ2 are split into two parts: first, results about the complexity of KDT test suites are reported; and, second, the way this complexity affects its evolution is presented.

1) KDT test suite complexity: To understand KDT test suite complexity, we calculate the *keyword level* and *connectivity* metrics, defined in Section IV-A. The first metric refers to the different "abstraction levels" (moving from pure technical to requirements expression) of the test suite and the second one, to the reusability among the keywords. Figures 5(a) and 5(b) present the corresponding results.

Figure 5(a) depicts our results of the *keyword level* for *Test Cases* and *User Keywords*, with the y-axis referring to the number of keywords per version. Recall that *Test Cases* are the complete instantiation of a test - root node in the tree representation - and *User Keywords* are user defined abstraction of the steps - intermediate nodes in the tree representation - (see also Section II-A). As can be seen from the figure, most *Tests Cases* are relatively complex, with a level of 5, whereas most *User Keywords* are simple (levels 1 to 2). This indicates that most user defined actions remain simple, in accordance with the philosophy of KDT.

> KDT *Test Cases* are complex having several levels of abstraction, whereas most *User Keywords* are simple.

Regarding the keyword reusability, Figure 5(b) plots the number of keywords per version (y-axis) with the *keyword connectivity* (x-axis). As can be seen, there is a high degree of reusability among *User Keywords*. More precisely, only 20.34% of the lines of code are used only once. Overall, the reused keywords amount to 51.56% of the total lines of code of TestSuiteA. As we will see next, this reusability is key to the decreased cost of the KDT test suite maintenance.

Another interesting finding is the presence of dead test code, i.e., keywords not used anywhere in TestSuiteA; these keywords have a connectivity of 0 in Figure 5(b). In total, 5.58% of the keywords were not used, which amounts to 4.58% of the test codebase. When we presented our findings to the QA team, they were surprised and confirmed the existence of dead code, explaining that there is no tooling to support such analysis. Our tool solves this issue and it is planned to be integrated into the team's test code development processes.

To investigate whether keywords of a particular level tend to be more reused than others, Figure 5(c) plots the keyword connectivity among the different keyword levels. By examining the figure, it becomes clear that keywords levels exhibit relatively high connectivity, indicating that the reusability of keywords is not restricted to a particular level with the exception of level 1 showing a slightly higher connectivity.

> There is a high degree of reusability among the keywords with 60% of keywords being reused, summing to 51.56% lines of test code savings.

2) KDT complexity and evolution: The second part of RQ2 refers to the evolution of KDT test suites and how their complexity affects it. To better understand the amount of changes performed during test code evolution, Figure 6 presents the test code churn (y-axis) over the eight-month period analyzed (x-axis), with a similar setup to Figure 3.

The purple line in the figure denotes the average churn across TestSuiteA's evolution and the light purple, its variance represented here by the standard deviation. From the figure, it can be observed that during *Creation*, the churn is 8.13%, on average, whereas in the *Maintenance* period, its value is 3.61%. Overall, keywords are changed with a churn rate of 5.11%. This number suggests that keywords are not entirely rewritten, but localized modifications are performed.

To investigate further how the complexity of the KDT test code affects its evolution, Figure 7 plots the number of changes, for the whole period studied, against the keyword connectivity and level and Figure 8 plots the churn against the same metrics.

After examining Figure 7(a), it becomes clear that keyword reused one to three times are mostly changed. Keywords with higher connectivity do not change that often. Moreover, the figure shows that changes are performed on dead code (connectivity 0). This confirms that testers are unware of the fact that these keywords are never executed, generating easy to avoid maintenance. Regarding the results for changes and level, depicted in Figure 7(b), we can observe that the changes to *Test Cases* do not follow a specific trend, whereas for the

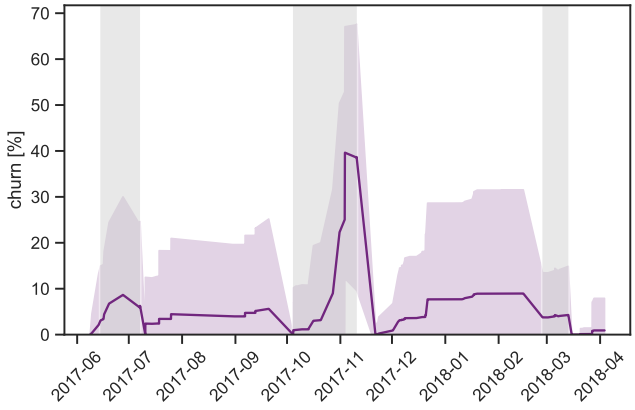
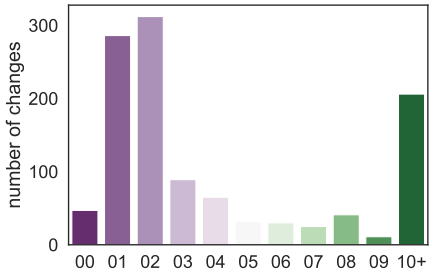
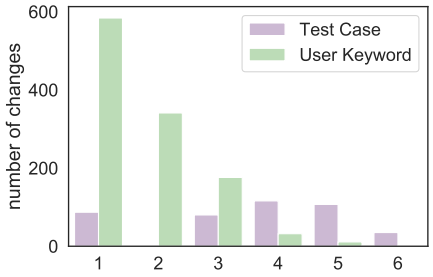


Fig. 6. KDT test code evolution: Churn over time



(a) Connectivity



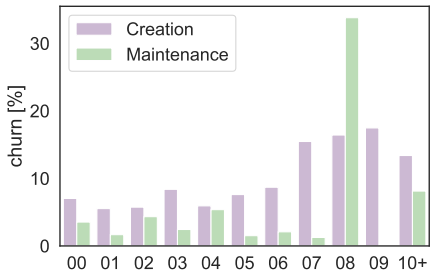
(b) Level

Fig. 7. Changes distribution according to level and connectivity

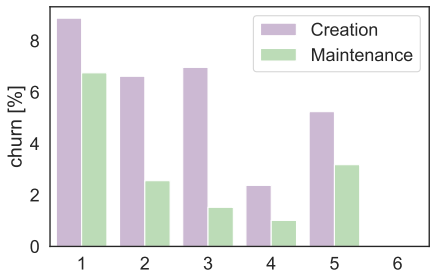
changes to the *Users Keywords*, the lower level the keyword is, the more it is suceptible to be changed.

Regarding our findings on the relation between churn rate and connectivity, depicted in Figure 8(a) for the *Creation* and *Maintenance periods*, we can conclude that, during *Creation*, keywords that are reused often, i.e. higher connectivity, exhibit approximately 50%-60% increased churn rate, whereas, during *Maintenance* the opposite holds. Finally, regarding the results presented in Figure 8(b) about churn and keyword level, we can see that, during *Creation*, keywords with lower levels exhibit high churn values, whereas in *Maintenance* this only holds for keywords of level 1. These results suggest that low level, highly reused keywords (basic action on the SUT), are evolving at a higher rate.

As we saw earlier, in TestSuiteA's evolution, keyword



(a) Connectivity



(b) Level

Fig. 8. Churn distribution according to level and connectivity

changed with a churn rate of 5.11% but we also saw in the previous section that keywords are reused often. This raises the questions: How many changes have been saved due to the reusability of the keywords? To answer this question, we compare the the number of changes applied to TestSuiteA to the same suite without the keyword abstraction as explained in Section IV-C2. We find that using KDT reduces the number of changes applied on TestSuiteA by 70.77% during "Creation", by 72.69% during "Maintenance" with an overall reduction during the entire period of 71.31%.

> The reuse of keywords reduces the maintenance cost by more than 70%. Changes are mostly done to keywords that are reused one, two or three times. Low level keywords are the ones that are changed most often. Keywords are evolving with a churn rate of 5%.

C. RQ3: KDT, Test Clones and Evolution

In RQ3, we explore whether KDT test suites contain test clones and how these clones affect TestSuiteA's evolution. Table III presents the corresponding results. The table presents the total number of keywords that appear during the evolution of TestSuiteA (for all 129 versions) for each type of clone detected (first column – Type I keyword clones, Type II and non-clones("Others")) and each type of evolution (second column). The types of evolution studied are the divided into three categories: keywords that are evolving strictly in the same way as others ("Co-evolution"), keywords that are evolving independently from others ("Evolution") and keywords that do not evolve ("No change").

TABLE III
KDT TEST CLONES AND EVOLUTION

Keywords	Types of Evolution: Co-evolution	Evolution	No change	Total
Type I	3526	3599	412	7537 (13.7%)
Type II	171	8462	491	9124 (16.5%)
Others	1888	33433	3184	38505 (69.8%)
Total	5585	45494	4087	55166 (100%)
Percent	10.1%	82.5%	7.4%	-

We can observe several interesting findings from the table. First, we see that Type I and Type II clones comprise 30.2% of the total amount keywords, indicating that almost one third of the test code written is duplicated. This finding highlights the fact that practitioners applying KDT will benefit from tools and techniques that can assist them in managing test clones.

Secondly, our results suggest that approximately 50% of the Type I test clones evolve in the exact same way, indicating that the practitioners apply the same changes multiple times, wasting valuable effort. This is a high figure, especially when compared to the co-evolution of non-cloned keywords which is 4.9%. Taking these results into consideration and the fact that almost 10% of the keywords are evolving in the same way, it becomes obvious that automated refactoring techniques can reduce the maintenance effort of KDT test suite evolution.

Finally, another interesting result exhibited in Table III concerns the overall evolution. We observe that only 7.4% of the keywords are not evolving. This shows that during the TestSuiteA's evolution more than 90% of the keywords are modified.

> Test clones exist in KDT test code. Type I and II clones amount to 30% of the test codebase. 50% of Type I clones evolve the same way, suggesting plenty of opportunities for test refactoring. More than 90% of the keywords evolve during their lifetime.

D. RQ4: Benefits and challenges of KDT: The Practitioners' perspective

This research question pertains to the benefits and challenges of KDT as perceived by the practitioners. In the following, we present the main findings of our interviews grouped by the two main questions of our study:

1) What are the benefits and challenges of adopting KDT?: All interviewees agreed on two main benefits of KDT: the low learning curve and its simple syntax. Thanks to its syntax that is close to the natural language, new users can easily start being productive. This syntax is also well-suited for communication purposes with teams that may have different backgrounds and expertize. The layered structure of keywords (i.e., the different keyword levels) plays an important role in facilitating this by hiding the technical details at the lower levels of the test suite and exhibiting the more business-oriented at the higher levels.

The main challenges encountered by the practitioners reside in their interaction with the SUT. Even a small evolution of the SUT can easily break the tests. Additionally, the testers discuss that finding the elements of the SUT that will be used in the tests is challenging, especially in applications where testability was not the primary concern.

2) What kind of changes are performed on the test suite and why?: The testers report two main reasons for the changes: SUT evolution and keyword adaptation.

Regarding the SUT evolution, the testers reported that as the SUT evolves, its components evolve as well which will cause the tests to adapt. The testers focus on two types of changes that are in according to our findings regarding RQ1 (cf. Section V-A): *locators*, i.e., finding which GUI elements of the SUT should be used in the tests and *synchronization* issues between the tests and the SUT.

Regarding keyword adaptation, the testers said that they create keywords in a "best effort" approach to cover the current needs. As new features of tests are developed, keywords are modified to become more generic. This fact explains the results illustrated in Figure 4 where we observed many changes in *user steps* during *Creation*.

> Practitioners find KDT easy to adopt, with a simple syntax that facilitates communication. The main reasons for test code evolution are due to *locators*, *synchronization* issues and *keyword refactoring*.

VI. THREATS TO VALIDITY

Threats to the external validity result from the generalization of our results outside the context of the study. Conducting the study with one industrial partner, the conclusions we draw may not be able to generalize to other companies using KDT. However, SubjectA is built using popular technologies, i.e., web frameworks and Java, which are wide-spread across the industry. Secondly and most important, this study is the first one, to the best of our knowledge, that analyzes the evolution of KDT test suites based on real-world data. Of course, this does not preclude the need for other studies to investigate further our results. Finally, another potential threat originates from the fact that we interviewed only 3 testers for RQ4.

Threats to the internal validity are due to the design of the study, potentially impacting our conclusions. The simple syntax of the test code allows for a robust model to be constructed. Our change algorithm presents some limitations: although phase 2 is based on the state of the art, it cannot detect *Move* operations, resulting instead in two operations *Delete* followed by an *Insert*. This limitation might have influenced our results during the accounting of the number of changes. However, the rather low number of the *delete step* operations (cf. Table II) indicates that this effect is marginal. Regarding the clone detection algorithm, as shown in [14], the rate of false-positives is known to be low for Type I and Type II clones.

Threat to construct validity result from the non suitability of the metrics used to evaluate the results. The main threat lies in the division of our work in two periods: "Creation"

and "Maintenance". While empirical data motivated this separation, they lack of theoretical grounding. Further work on the test execution is needed to better motivate this decision.

VII. Related Work

In the literature, we find a great amount of work tackling the problem of test code evolution. For instance [10] and [8] analyze the co-evolution between test code and production code. Levin et al. [8] analyze 61 open source projects to establish a relationship between test maintenance and production code maintenance. They create a model to see what type of changes in the code base cause maintenance of the test suite.

Pinto et al. [7] analyze the evolution of test suites and extract actions performed on the test suite in order to see how the test suite is evolving. They conclude that test repair occurs in practice and that it is not due to only assertion fixes and suggest further research of automatic repair tools. Although this work is similar to ours, the focus of the study is the unit level while we focus on the acceptance level.

To the best of our knowledge, Skoglund and Runeson [21] were the first to conduct an empirical study on the evolution of system level tests. The authors conduct an exploratory analysis to investigate potential test suite maintenance issues. They explore three strategies to minimize the number of changes in the test code resulting from a production code change. Their study found that in one strategy, more changes were needed to maintain the test code while with the other no changes were needed to the unit test code. However, since their dataset is synthetic, they only draw qualitative conclusions.

Grechanik et al. [22] perform a cost-benefit analysis of tool-based GUI-based application (GAP) functional test suites versus manual maintenance. They describe a case study with 45 professional programmers and test engineers. They show that the automated GUI testing approach (QTP) reports more broken test script statements due to changes in GUI with fewer false positives than the manual approach. However, they recommend against the tool-based repair approach for experienced test engineers because of the high cost of each license and the low added value.

Shewchuk et al. [23] create a functional test suite using the tool IRFT for the open source project JEdit. In their work, they measure the effort to create and maintain the test suite and compared its size and number of changes against the production code. They conclude that the method to create test using IRFT is effective, with respect of the effort needed to develop and maintain functional test suites as well as the fault detection capabilities of those suites. One limitation of this study is that it uses synthetic test suites.

Alegroth et al. [9], [24] analyze the costs and factors associated with the maintenance of Visual GUI testing (VGT) based on an empirical study. They identified 13 factors influencing maintenance. Their work shows that the cost of creation is much higher than the cost of maintenance. They also show that the cost of maintenance can be reduced by frequent evolutions instead of few big changes. Lastly, the authors build a cost model comparing manual testing to automated testing. They conclude that the return on investment of using VGT is positive.

In their work, Labuschagne et al. [25] explore the cost and benefits of automated regression testing in practice. To do so, they select 61 projects and analyze their test execution reports. They show that in some cases tests break because of invalid assumption and that maintenance cost could be reduced via the use of better development processes.

Another similar study is the one of Lavoie et al. [16] who analyzed potential code duplication in TTNC-3 test scripts in industrial telecommunication software. Their findings suggest that 24% of the code fragments in the test suites are clones. We find analogous evidence of the presence of clones in KDT test code.

Although much work has been conducted on test evolution, work on acceptance testing code evolution is still scarce. In this study, we try to fill this gap and provide quantitative and qualitative data on the cost and benefits of creating and maintaining KDT in practice.

VIII. Conclusion and future work

Understanding the changes performed by testers during test code evolution is key to automated test refactoring and repair techniques and will provide valuable information on the challenges they face. Towards this direction, this paper presents an extensive study on the evolution of industrial Keyword-Driven (KDT) test suites across an eight-month period where we identify and categorise the corresponding test code changes. Our results suggest that KDT test design is complex with several levels of abstraction and that this design favours reusability; more than 60% of the keywords are reused which has the potential of reducing the changes needed during evolution up to 70%.

Additionally, we find that keywords change with a relatively low rate (approximately 5%) indicating that after a keyword's creation only fine-grained, localised changes are performed by the testers. Our results suggest that the most common changes to KDT tests are caused by *synchronization* or element *location* changes between the SUT and the test suite and to the *assertions* of the tests. Our findings indicate that during evolution 90% of the keywords evolve and that test clones exist in KDT test suites; approximately 30% of the keywords are duplicated. Finally, we report on the practitioners' perception on the challenges and benefits of adopting KDT.

This work forms the first step towards improving test quality and supporting test maintenance. Our results show that KDT techniques require tooling to support keyword selection, test refactoring and test repair.

Acknowledgement

The authors would like to thank Raphaël Formica, Mohamed Reqba, Isabelle Hoffert-Clausse and Christophe Chattou. This work is partially funded by Alphonse Weicker Foundation and by the Luxembourg National Research Fund[1]

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

REFERENCES

[1] P. Ammann and J. Offutt, *Introduction to Software Testing*, 2nd ed. New York, NY, USA: Cambridge University Press, 2016.

[2] 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.

[3] S. Thummalapenta, S. Sinha, N. Singhania, and S. Chandra, "Automating test automation," in *2012 34th International Conference on Software Engineering (ICSE)*, June 2012, pp. 881–891.

[4] S. Sivanandan and Yogeesha C. B, "Agile development cycle: Approach to design an effective Model Based Testing with Behaviour driven automation framework," in *20th Annual International Conference on Advanced Computing and Communications (ADCOM)*. IEEE, sep 2014, pp. 22–25.

[5] 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.

[6] Capgemini Sogeti and Micro Focus, "World Quality Report 2017-18," p. 74, 2017, (last accessed on October 2018). [Online]. Available: https://www.sogeti.com/globalassets/global/downloads/testing/wqr-2017-2018/wqr_2017_v9_secure.pdf

[7] L. S. Pinto, S. Sinha, and A. Orso, "Understanding Myths and Realities of Test-suite Evolution," *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, vol. 1, pp. 33:1–33:11, 2012.

[8] S. Levin and A. Yehudai, "The co-evolution of test maintenance and code maintenance through the lens of fine-grained semantic changes," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sept 2017, pp. 35–46.

[9] E. Alégroth, R. Feldt, and P. Kolström, "Maintenance of automated test suites in industry: An empirical study on Visual GUI Testing," *Information and Software Technology*, vol. 73, pp. 66–80, 2016.

[10] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," *Empirical Software Engineering*, vol. 16, no. 3, pp. 325–364, jun 2011.

[11] R. RobotFramework. (2018) Introduction. [Online]. Available: http://robotframework.org/

[12] 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.

[13] B. Baker, "On finding duplication and near-duplication in large software systems," in *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*, Jul 1995, pp. 86–95.

[14] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, may 2009.

[15] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Inf. Softw. Tech.*, vol. 55, no. 7, pp. 1165 – 1199, 2013.

[16] T. Lavoie, M. Mérineau, 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, jul 2017.

[17] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," in *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '96. New York, NY, USA: ACM, 1996, pp. 493–504.

[18] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14*. New York, New York, USA: ACM Press, 2014, pp. 313–324.

[19] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling:tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, Nov 2007.

[20] E. Ukkonen, "Algorithms for approximate string matching," *Information and Control*, vol. 64, no. 1-3, pp. 100–118, jan 1985.

[21] M. Skoglund and P. Runeson, "A case study on regression test suite maintenance in system evolution," in *IEEE International Conference on Software Maintenance, ICSM*, 2004, pp. 438–442.

[22] M. Grechanik, Q. Xie, and C. Fu, "Experimental assessment of manual versus tool-based maintenance of GUI-directed test scripts," in *2009 IEEE International Conference on Software Maintenance*. IEEE, sep 2009, pp. 9–18.

[23] Y. Shewchuk and V. Garousi, "Experience with Maintenance of a Functional GUI Test Suite using IBM Rational Functional Tester," in *Proceedings of the 22nd International Conference on Software Engineering & Knowledge Engineering (SEKE'2010)*, Redwood City, San Francisco Bay, CA, USA, 2010, pp. 489–494.

[24] E. Alegroth, R. Feldt, and H. H. Olsson, "Transitioning Manual System Test Suites to Automated Testing: An Industrial Case Study," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, mar 2013, pp. 56–65.

[25] A. Labuschagne, L. Inozemtseva, and R. Holmes, "Measuring the cost of regression testing in practice: a study of Java projects using continuous integration," *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*, pp. 821–830, 2017.