Tool Support for Refactoring Manual Tests

Elodie BERNARD*†, Julien BOTELLA‡, Fabrice AMBERT*, Bruno LEGEARD*‡, Mark UTTING§
*FEMTO-ST Institute, Univ. Bourgogne Franche-Comté, CNRS Besançon, France
† Sogeti, Lyon, France
‡ Smartesting, Besançon, France
§ USC Business School, University of the Sunshine Coast, Sippy Downs, Australia
[elodie.bernard|fabrice.ambert|bruno.legeard]@femto-st.fr
julien.botella@smartesting.com
utting@usc.edu.au

***Abstract*—Manual test suites are typically described by natural language, and over time large manual test suites become disordered and harder to use and maintain. This paper focuses on the challenge of providing tool support for refactoring such test suites to make them more usable and maintainable. We describe how we have applied various machine-learning and NLP techniques and other algorithms to the refactoring of manual test suites, plus the tool support we have built to embody these techniques and to allow test suites to be explored and visualised. We evaluate our approach on several industry test suites, and report on the time savings that were obtained.**

***Index Terms*—Test suite refactoring, Test suite minimisation, Lightweight MBT, Machine Learning for Software Testing**

I. Introduction

The increasing complexity of applications, the accelerated pace of production releases, and frequent personnel turnover, can lead to test assets becoming poorly maintained. A large test suite with a long history is likely to contain tests that are obsolete, redundant, use inconsistent terminology, and many other flaws that make them less than optimally useful. For example, 82% of respondents to the 2019 survey of the French Software Testing Qualification Board answered that they had been confronted with a problem of obsolescence of test repositories [1].

This paper focuses on manual test suites, described by natural language, and the challenging of providing good tool support for refactoring those test suites to make them more understandable, maintainable, reusable, concise, and more amenable to automation. The driving force is primarily cost: the tool support should reduce the cost of doing such refactoring compared to refactoring manually, and the resulting refactored test suites should be less costly to use and maintain than the original test suites.

We begin by discussing the motivations for our work, the contributions, and the related work. Then Section III discusses how we have applied various machine learning and natural language processing (NLP) techniques and other algorithms to the refactoring of manual test suites, Section IV describes the tool support we have built, Section V gives a series of recommended steps for effective refactoring, Section VI describes an evaluation of our approach on several industry test suites, and Section VII discusses conclusions and future work.

A. Motivation

Expanding on the general goal of cost-effectiveness mentioned above, there are three primary goals for our work:

1) **Test Suite Exploration:** is the first necessity for a tester who comes to a large and unfamiliar repository of manual test cases. We want to provide automated ways of grouping test cases, finding redundancies and similarities between test cases, so that the tester can quickly understand and start to optimize the test suite. For this paper, we assume the general case where no test execution history is available, so exploration is based just on the test case documentation - i.e. test steps - in natural language text of the manual test cases.
2) **Legacy Test Sanitization:** is the process of reversing the entropy of the test suite, in order to make it more maintainable and easier to use and understand. This typically involves changes such as: reintroducing consistent terminology and abbreviations, standardizing test step description, detecting and removing redundant tests, abstracting similar test steps by transforming them into a single parameterized step, and grouping related tests into suites.
3) **Test Model Generation:** can be a useful first step towards automating some of the manual tests, but is also a helpful way of visualizing the similarities and differences between the test cases in a test suite. We want our tool to be able to generate a concise graphical summary of any given set of tests, and then export that partial model of the tests as a starting point for automation using model-based testing, and also as documentation that gives an overview of those tests and the relationships between them.

B. Contributions

The primary contributions of this work are:

- demonstration of how machine learning and NLP techniques can be applied to the task of refactoring manual test suites;
- the creation and visualization of test models for any given suite of tests;
- novel tool support for refactoring manual tests and a recommended refactoring process;

- an industry evaluation that demonstrates that the tool support reduces the time required for refactoring a test suite, and that the resulting test suite has reduced execution time.

II. Related work

Test asset management is a regular concern for teams, whether to eliminate redundancies or to manage test obsolescence. Test suite reduction addresses the first of these concerns, and is an active research area [2], [3]. Several approaches and tools propose to reduce the size of test suites and thus the time required to run them, but whether they are based on requirement coverage [4], [5] or test similarity [6], all are dedicated to automated testing. Our work has similarities with test techniques following reduction, we implement clustering on tests [7] but on manual tests expressed in natural language. By manual tests, we mean system-level test cases or acceptance test cases composed of a set of steps, which are themselves composed of an action and an expected result. These tests are based on a variety of application test assets: from web applications to large desktop applications.

NLP techniques are, in the context of testing, generally used to generate tests from requirements. They can capture the semantics of requirements to generate keyword sequences corresponding to the requirement tests [8]–[10]. In our approach, NLP is used in two activities. Firstly, to group tests based on similarities between scenarios [11] and secondly, in the refactoring of test steps to allow the merging of similar test steps by using a distance calculation between terms [12].

III. Refactoring Techniques

This section describes the various machine-learning clustering techniques and NLP (Natural Language Processing) algorithms that we have applied to the tasks of grouping tests by their similarity and assisting in the refactoring of those test steps. For each task, we describe several approaches that we tried, and the one that we selected as being the most useful, with rationales. The memory and runtime efficiency of each approach was a major consideration, since the resulting tool must run inside a browser on a typical client machine, so that companies can refactor their test suites without their test-suite data going off-site.

A. Test clustering

The first thing a user needs, when refactoring manual tests, is to be able group tests by functional scope. This can allow them to identify where the refactoring effort should be spent. We tried several clustering techniques. They all have in common the necessity to characterize each test by a numeric vector in order to compute the distance between tests. For a given set of test cases, the resulting set of vectors forms a matrix, which is the basis for the clustering algorithms. The next subsection describes various techniques we tried to create a vector that characterizes each test case, and the following subsection describes clustering techniques.

1) Vectorization of test cases: A vector is a list of numbers that define a position in space of the object it represents relative to other objects of that kind. We evaluated seven approaches to representing a manual test by a vector: the first three are *word-based*, based on the set of words used in the description of the test, while the others are *step-based*, considering just the steps within each test.

1) our initial approach was the common bag-of-words encoding, which just counts the number of times each word appears in the test. A problem with this approach is that it creates very large vectors, because there are many distinct words. These vectors consume too much memory for our requirements.
2) we tried removing all short words (three or less characters). But because this ignores some significant words (on, off, etc.), clustering results were less accurate.
3) we tried to avoid non-significant words by using tf-idf (Term Frequency-Inverse Document Frequency), which weights each word by its usage in the full document. This gives smaller, more relevant vectors. This technique, often used for SEO (Search Engine Optimization), gives good results but needs longer computation time prior to clustering. Vector size is still high and when the number of words increases, memory usage and computation time also increase.
4) our first step-based approach was to treat each test step as a single string, and use bag-of-words on these strings. So each vector element is the number of times that exact step appears in the test case. However, this strict equality was too strong, since steps with data values are considered different, and so are not grouped together.
5) an improvement on the previous approach is to replace each step parameter value (if any) by a placeholder (i.e. 'data') prior to comparison. This significantly improves results for tests with parameterized steps, but not for other nearly-identical steps, which are important when working with legacy and often obsolete test suites.
6) we merge steps that are 'syntactically close' (*Levenshtein* [13] distance less than K) into a single vector element. As above, we also replace parameter values by a placeholder before comparing steps. The problem with the Levenshtein metric is that it considers a three character string with one single-character difference to another string, to have the same distance value as a hundred character string with one single-character difference from another string.
7) to solve the above issues with Levenshtein distance, we compute a similarity *ratio* between each pair of test steps, using the *Jaro-Winkler* distance [14], which produces a number between 0 and 1. The higher the result is, the closer the strings are. We consider two steps as identical if their *Jaro-Winkler* distance is higher than 0.8. This creates vectors with fewer elements, but still meaningful enough to be used for clustering.
8) A further improvement consists of not considering the

step number of usage during the vector creation, but only its presence. We found that this avoids creating strong vector differences if a test step is included in several iterations of a loop.

This final approach to vector creation has sufficiently small memory and runtime overhead to be practical, since it is based on test steps rather than words. The distance calculations invoke *Jaro-Winkler* $O(N^2)$ times, where N is the number of steps, but this can be reduced by pre-filtering if required.

We now describe how this vector creation approach has been applied as a basis for several clustering algorithms.

2) Clustering algorithms:

1) a first approach is to use the *k-means* algorithm to partition a number of observations (here test sequences) into k clusters. Results were good, but the problem was that the user must choose k prior to clustering. This k can be evaluated with the elbow method (https://www.scikit-yb.org/en/latest/api/cluster/elbow.html) but this requires running the algorithm multiple times before choosing k, which increases computation time significantly.
2) a second approach was the *mean-shift* algorithm (https://scikit-learn.org/stable/modules/generated/sklearn.cluster.MeanShift.html), which discovers groups in a smooth density of samples (here tests) and chooses the number of clusters automatically. This gave good results, but we noticed that users often want to extract 'sub-groups' from the groups that were originally computed, rather than having just one level of clustering.
3) to enable multiple levels of clustering, we can use a *hierarchical clustering* algorithm, such as http://www.analytictech.com/networks/hiclus.htm. This algorithm starts by assigning each test to its own cluster. Then the closest clusters are grouped into a single cluster, repeating this process until all tests are clustered into a single cluster. The result of the algorithm is a *dendrogram* [15] – see Fig. 1 for an example. To be more convenient, we found a graphical representation to enable the used to interact with this dendrogram as seen in Fig. 3.

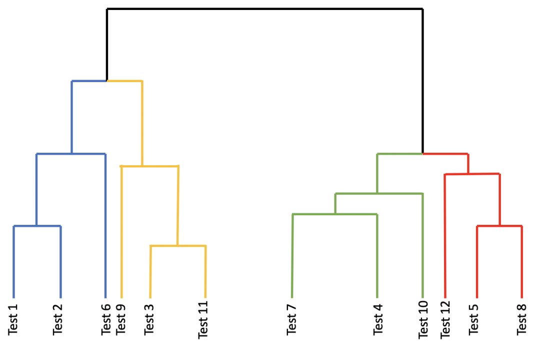


Fig. 1. An example of a simple Dendrogram diagram.

Once the tests have been grouped together, we want to enable the user to choose one of those groups and refactor some of the test steps within the group to make them more consistent and homogeneous.

B. Test step refactoring

Two mechanisms have been implemented to reduce the entropy of the test suite. One merges similar steps by making them identical, and the other introduces parameterized steps to capture steps that are identical apart from data values.

1) Merging similar steps: Two steps may represent the same test action, but have differences in their wording. This can be due to typos, or because several testers may have maintained the test suite. In order to make manual test execution easier the same test action should have the same wording each time it is used. Furthermore to automate the test suite execution, we may want to use a keyword approach, with each keyword representing a test action. The number of keywords to implement will be lower after the test suite has been refactored. The tool uses the same string distance techniques as for clustering (e.g. Jaro-Winkler distance) to identify similar steps and allows the user to merge such test steps, and all their usages in the test suite.

2) Parameterizing steps: The second way to merge test steps is to parameterize them. For example, if there are two similar steps: '**Click on the *red* button**' and **Click on the *green* button**', then these steps could both be replaced by a parameterized step: '**Click on the *COLOR* button**', where *COLOR* is a parameter that can be instantiated to a concrete data value such as ***red*** or ***green***.

To support this refactoring, once a test step and its parameter have been identified by the user, the tool uses regular expressions to find all related test steps that are candidates for adding this parameter. The chosen step and its parameter are transformed into a regular expression (i.e. 'Click on the * button'), which is used to search for potential instantiations of the new parameterized step. These can then be displayed to the user as potential parameterization candidates.

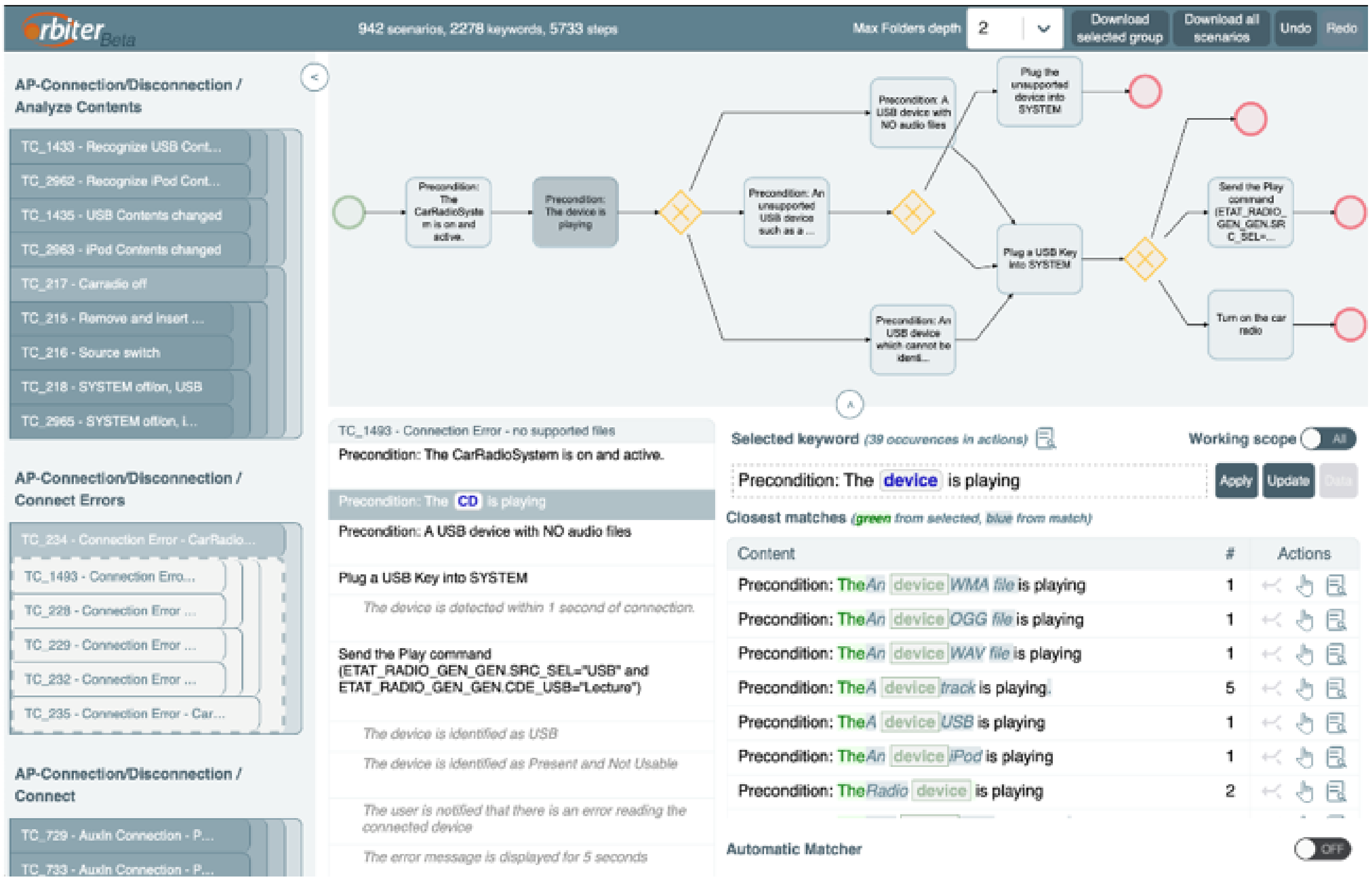
IV. Tool

The Orbiter tool includes several features dedicated to test refactoring, each with a recommended usage. Figure 2 shows the general interface of the Orbiter tool (at Oct 2019), which is available and can be used online on a trial basis [16].

In this section, we will explain each of the main features of this tool. In the next section we will describe a recommended methodology to better use these features to perform efficient test suite refactoring.

The tool is composed of four main features:

- Test suite analysis for test case clustering;
- Optimization and refactoring of test cases;
- Visualization of test cases and relations between them;
- Metrics for tracking refactoring.



orbiter Beta
Max Folders depth
2
Download selected group
Download all scenarios
Undo
Redo
Precondition: The CarRadioSystem is on and active.
Precondition: The device is playing
Precondition: A USB device with NO audio files
Plug the unsupported device into SYSTEM
Precondition: An unsupported USB device such as a ...
Plug a USB Key into SYSTEM
Precondition: An USB device which cannot be identi...
Send the Play command (ETAT_RADIO_GEN_GEN.SRC_SEL=...
Turn on the car radio
Apply
Update
Data

942 scenarios, 2278 keywords, 5733 steps

AP-Connection/Disconnection / Analyze Contents

- TC_1433 - Recognize USB Cont...
- TC_2962 - Recognize iPod Cont...
- TC_1435 - USB Contents changed
- TC_2963 - iPod Contents changed
- TC_217 - Carradio off
- TC_215 - Remove and insert ...
- TC_216 - Source switch
- TC_218 - SYSTEM off/on, USB
- TC_2965 - SYSTEM off/on, i...

AP-Connection/Disconnection / Connect Errors

- TC_234 - Connection Error - CarRadio...
- TC_1493 - Connection Erro...
- TC_228 - Connection Error ...
- TC_229 - Connection Error ...
- TC_232 - Connection Error ...
- TC_235 - Connection Error - Car...

AP-Connection/Disconnection / Connect

- TC_729 - Auxln Connection - P...
- TC_733 - Auxln Connection - P...

TC_1493 - Connection Error - no supported files

Precondition: The CarRadioSystem is on and active.

Precondition: The CD is playing

Precondition: A USB device with NO audio files

Plug a USB Key into SYSTEM

The device is detected within 1 second of connection.

Send the Play command (ETAT_RADIO_GEN_GEN.SRC_SEL="USB" and ETAT_RADIO_GEN_GEN.CDE_USB="Lecture")

The device is identified as USB

The device is identified as Present and Not Usable

The user is notified that there is an error reading the connected device

The error message is displayed for 5 seconds

Selected keyword *(39 occurences in actions)*

Working scope All

Precondition: The device is playing

Closest matches *(green from selected, blue from match)*

Content	#	Actions
Precondition: TheAn device WMA file is playing	1	
Precondition: TheAn device OGG file is playing	1	
Precondition: TheAn device WAV file is playing	1	
Precondition: TheA device track is playing.	5	
Precondition: TheA device USB is playing	1	
Precondition: TheAn device iPod is playing	1	
Precondition: TheRadio device is playing	2	

Automatic Matcher OFF

Fig. 2. Screenshot of the Orbiter Tool.

A. *Test suite analysis for test case clustering*

This first feature allows test cases to be imported from a test management tool (e.g. ALM, Squash), to cluster those test cases using hierarchical clustering algorithms, and display them in a hierarchical view using dendrograms [15]. We can thus combine the existing grouping of test cases by their folders in the repository (a tree structure) with the clustering based on the similarities between the textual description of each test case.

In addition to applying this clustering to the whole test suite in one group, it can also be applied to each folder at a given level within the test repository. This keeps each top-level repository folder separate, but allows clustering of tests within that folder. This feature can also be used to analyze redundancies that occur between different folders, by switching between different views as shown in Fig. 3. The left diagram shows test cases grouped within each sprint, to identify similarity within Sprint 1, and within Sprint 2, etc. At this level, identifying whether similarities exist between sprints is complex. But by changing the depth level to the global level (right-hand side), we can see there are similarities between tests in sprints 4 and 5, such as between the test cases *"Sprint 5/Management of Owner with Secondary residence"* and *"Sprint 4/Owner Spain residence"*.

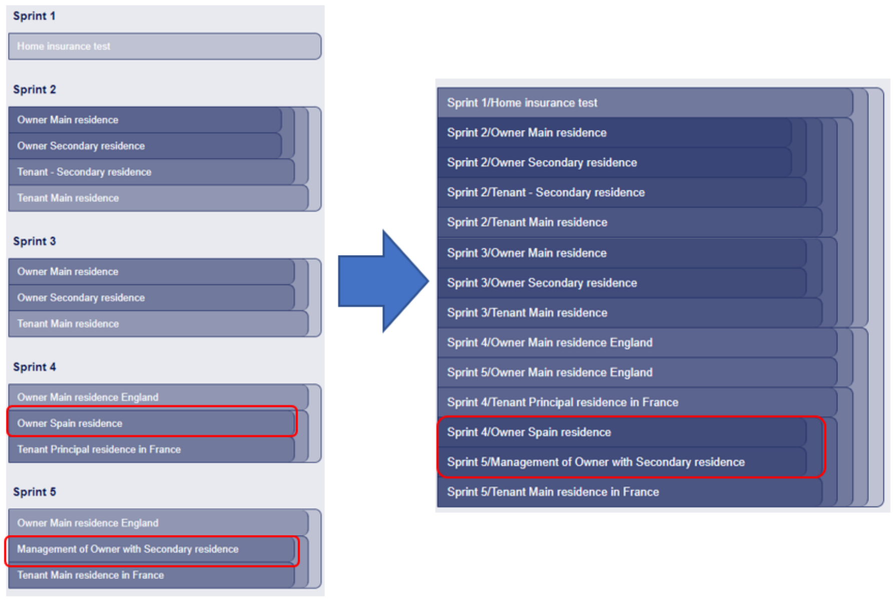
These redundancies are difficult to identify 'manually' in common test repositories, as this requires consulting all test cases one by one to identify these redundancies. Here, the dendrogram visualization makes the task much easier. Finally, the dendrograms allow the user to select a cluster of similar test cases in order to apply optimization and refactoring to those tests. This is the subject of the following section.

B. *Optimization and refactoring of test cases*

This subsection describes three refactoring features of Orbiter that can be used to clean up and improve the quality and consistency of a set of manual test cases.

1) Test correction: The first refactoring feature that Orbiter offers is the identification of similar test case steps, so that they can be refactored to use the same terminology, phrasing, etc. As described above, this refactoring can be applied at different levels, to find similar test case steps within a selected set of test cases, or globally across the entire repository.
For example, Fig. 3, shows two test cases selected, so applying this refactoring in this state means that only the steps contained in these two test cases will be analyzed for similarity.

Figure 4 illustrates the optimization and refactoring possibilities offered by the tool. The left panel shows the details of the chosen test cases plus all their test steps and expected results (the indented light gray lines). Here, the step '**Choose**


Sprint 1
- Home insurance test

Sprint 2
- Owner Main residence
- Owner Secondary residence
- Tenant - Secondary residence
- Tenant Main residence

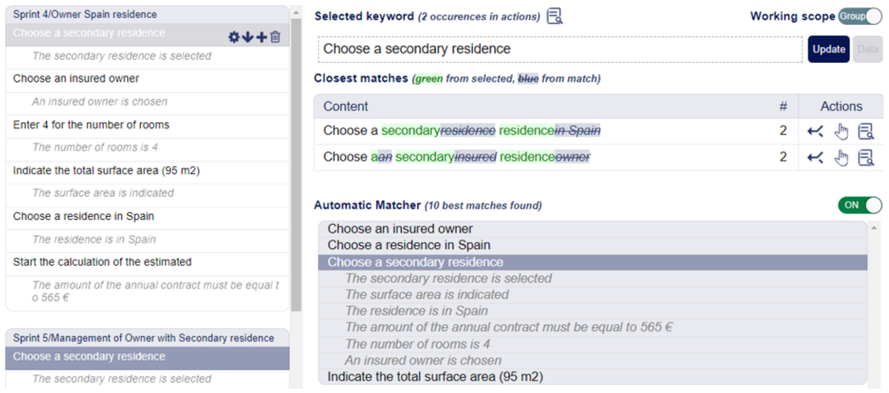
Sprint 3
- Owner Main residence
- Owner Secondary residence
- Tenant Main residence

Sprint 4
- Owner Main residence England
- Owner Spain residence
- Tenant Principal residence in France

Sprint 5
- Owner Main residence England
- Management of Owner with Secondary residence
- Tenant Main residence in France

- Sprint 1/Home insurance test
- Sprint 2/Owner Main residence
- Sprint 2/Owner Secondary residence
- Sprint 2/Tenant - Secondary residence
- Sprint 2/Tenant Main residence
- Sprint 3/Owner Main residence
- Sprint 3/Owner Secondary residence
- Sprint 3/Tenant Main residence
- Sprint 4/Owner Main residence England
- Sprint 5/Owner Main residence England
- Sprint 4/Tenant Principal residence in France
- Sprint 4/Owner Spain residence
- Sprint 5/Management of Owner with Secondary residence
- Sprint 5/Tenant Main residence in France

Fig. 3. Hierarchical clustering of test cases in Orbiter – left side shows clustering within each folder, while the right side shows global clustering of all tests.


Sprint 4/Owner Spain residence
Choose a secondary residence
The secondary residence is selected
Choose an insured owner
An insured owner is chosen
Enter 4 for the number of rooms
The number of rooms is 4
Indicate the total surface area (95 m2)
The surface area is indicated
Choose a residence in Spain
The residence is in Spain
Start the calculation of the estimated
The amount of the annual contract must be equal t o 565 €

Sprint 5/Management of Owner with Secondary residence
Choose a secondary residence
The secondary residence is selected

Selected keyword *(2 occurences in actions)*
Working scope Group
Choose a secondary residence
Update
Data

Closest matches *(green from selected, ~~blue~~ from match)*

Content	#	Actions
Choose a secondary~~residence~~ residence~~in Spain~~	2	
Choose a~~an~~ secondary~~insured~~ residence~~owner~~	2	

Automatic Matcher *(10 best matches found)* ON

Choose an insured owner
Choose a residence in Spain
Choose a secondary residence
The secondary residence is selected
The surface area is indicated
The residence is in Spain
The amount of the annual contract must be equal to 565 €
The number of rooms is 4
An insured owner is chosen
Indicate the total surface area (95 m2)

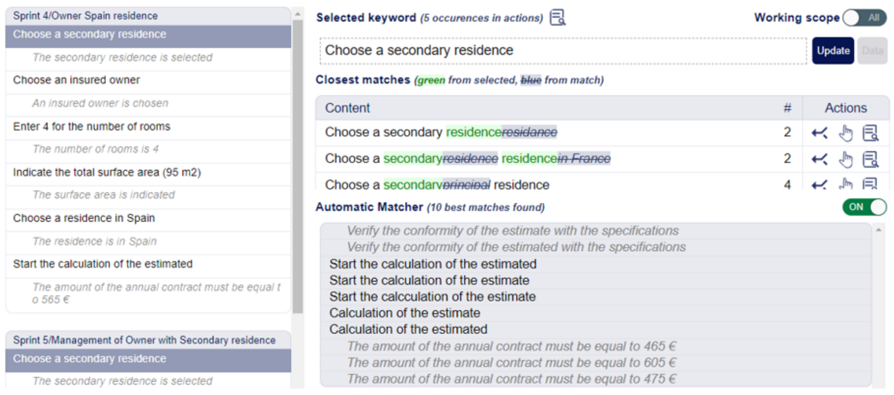
Fig. 4. Illustration of test step refactoring features in the Orbiter tool. The left-hand panel shows a selected test step, and the right-hand panel shows the closest matches (2), followed by the top-ten matching lines.

a secondary residence' is selected. At the top of right-hand panel the 'Selected keyword' area shows the selected test step and allows us to edit it. In the 'Closest matches' area below this, we find a list of test steps (two in this case) that are most similar to the selected step. The text of each matching test step shows the differences from the selected keyword by displaying missing text in blue with a line through the middle of the text, and added text in green on a green halo background. Each match also provides several action buttons, including one that replaces the matching text by the selected keyword to make them identical. This supports one-click refactoring to remove differences such as spelling mistakes and inconsistent terminology.

Finally, the 'Automatic Matcher' panel at the bottom shows the results of a background search within the current working scope to find the top ten phrases (test steps or expected results) that may be good candidates for refactoring, because they are highly similar to some other phrase in the working scope. The resulting phrases are not necessarily related to the selected keyword, but simply suggest to the user other refactoring tasks within the current scope that may be productive.

In the next screenshot, Fig. 5, the working scope has been changed to global (all tests), and we can see that more close matches to our selected keyword are found and the 'Automatic Matcher' panel shows a different set of 10 lines with higher levels of similarity to each other. The use of global scope versus local scopes will be discussed in the following section, which will present good practices for test suite analysis and refactoring.

2) *Test correction:* The second feature associated with test optimization and refactoring is test correction. Test correction offers the possibility to delete, add, swap, and edit steps. These operations can be performed at the scale of a test case or more broadly at a test group. Thus for a step it is possible to edit



Sprint 4/Owner Spain residence
Choose a secondary residence
The secondary residence is selected
Choose an insured owner
An insured owner is chosen
Enter 4 for the number of rooms
The number of rooms is 4
Indicate the total surface area (95 m2)
The surface area is indicated
Choose a residence in Spain
The residence is in Spain
Start the calculation of the estimated
The amount of the annual contract must be equal t o 565 €

Sprint 5/Management of Owner with Secondary residence
Choose a secondary residence
The secondary residence is selected

Selected keyword *(5 occurences in actions)*

Choose a secondary residence

Closest matches *(green from selected, ~~blue~~ from match)*

Content	#	Actions
Choose a secondary residence~~residance~~	2	
Choose a secondary~~residence~~ residence~~in France~~	2	
Choose a secondary~~principal~~ residence	4	

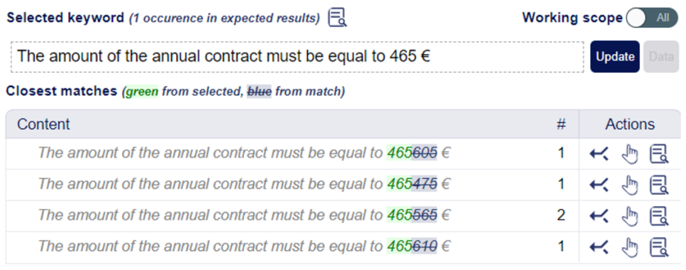
Automatic Matcher *(10 best matches found)*

Verify the conformity of the estimate with the specifications
Verify the conformity of the estimated with the specifications
Start the calculation of the estimated
Start the calculation of the estimate
Start the calcculation of the estimate
Calculation of the estimate
Calculation of the estimated
The amount of the annual contract must be equal to 465 €
The amount of the annual contract must be equal to 605 €
The amount of the annual contract must be equal to 475 €

Fig. 5. Screenshot of Orbiter tool with Working Scope changed to global (all tests), showing more similar matches.

it either at the level of a single test case or on a whole set of test cases.

3) Refactoring by parameterization: Orbiter also offers the possibility of inserting parameters into test steps in order to unify several test steps that are similar except for one small phrase. The use of parameters can give a higher level of refactoring by unifying keywords that deal with the same action, but use different values.



Selected keyword *(1 occurence in expected results)*

The amount of the annual contract must be equal to 465 €

Closest matches *(green from selected, ~~blue~~ from match)*

Content	#	Actions
The amount of the annual contract must be equal to 465~~605~~ €	1	
The amount of the annual contract must be equal to 465~~475~~ €	1	
The amount of the annual contract must be equal to 465~~565~~ €	2	
The amount of the annual contract must be equal to 465~~610~~ €	1	

Fig. 6. Before applying parameterization refactoring.

In Fig. 6, with the selected keyword '**The amount of the annual contract must be equal to 465€**', it is clear that there are 4 close matches that are extremely similar. The only difference is the amount of the annual membership fee. The definition of a parameter is easily done by selecting the value selected to be parameterized, here 465 €, then by defining a name for the parameter, here *"amount"*.

The tool also provides automatic propagation functionality by suggesting steps that can match with the application of the parameter, thus facilitating the homogenization of steps. This homogenization can be done individually by updating the selected keyword, or by applying the parameterization to the closest matches proposed. The details of the manipulations will be done through the presentation of the refactoring process.

C. Visualization of test sequences and relations between them

For a selected test case or group of test cases, it is possible to obtain a visualization of these scenarios in the form of a business process diagram showing the sequences of all the steps in all those tests (see Figures 8 and 8 for examples of these diagrams). These diagrams are animated in the sense that we can hover over a specific test case and see its path through the business process diagram highlighted in bold. In addition, as steps are updated the diagram is also updated on the fly, so the user can see test step nodes being merged in the diagram. This use of workflow diagrams as a support for refactoring is described in the description section of the tool's usage process.

D. Metrics for tracking refactoring

In order to determine the progress of refactoring and optimization of the test repository, the tool provides metrics such as: the number of test cases, the number of steps (actions and expected results) within those tests, and also the number of 'keywords' (any test step that is used more than once). The use of these metrics is described further in Section VI.

V. Recommended Refactoring Process

This section describes the refactoring process we used during the evaluation. It is organized into five main phases:

1) choosing a scope to refactor;
2) choosing which test steps to refactor;
3) refactoring those test steps to homogenize them;
4) refactoring whole test cases
5) applying parameterization refactoring.

A. Identifying a scope to refactor

One of the first questions that can arise when starting to refactor an existing test repository is deciding where to start. Even if a first subset of test cases is identified, it can still be complex to know where to begin. We have identified two complementary starting points, where the choice between them depends on how well the existing test cases are structured.

In the case where the existing test cases are very poorly structured, with many similar test steps written differently, so requiring significant refactoring, it is advisable to start by using the 'Automatic Matcher' on a large or global scope in order to improve the test steps. This will allow an initial homogenization and simplification of the global test repository.

This operation can be stopped when the Automatic Matcher no longer offers any relevant results.

This starting point is optional, because it depends on the maturity of the test repository, but it is advisable to carry it out before starting further refactoring, as it saves a significant amount of time for further optimizations. Indeed, careless usage of the Automatic Matcher on small local scopes can result in redundant operations. An example is to work on a specific scope, choose to homogenize steps in large numbers (within this restricted scope), but then have to repeat these same operations later because another scope contains similar test steps to the scope previously studied.

Another advantage of using the Automatic Matcher with a global scope early in the refactoring process is that it highlights similar steps across the whole repository, and therefore provides the tester with a better knowledge of the test repository.

The second starting point, which is independent of the Automatic Matcher, consists of looking directly at the dendrogram tree structure in order to determine the scope of test cases to target for refactoring. Even if Orbiter allows you to study a repository as a whole, starting with a subset can be a smart solution, particularly for huge repositories. It is the dendrogram that will determine where to start refactoring.

As explained in the tool description, Orbiter allows you to group the test case by similarity to different degrees: on the tree structure on different levels (1, 2, 3, etc.) or at level 0, i.e. globally without taking the tree structure into consideration. In all situations, the test cases are grouped by similarity and it is possible to study the proposed dendrogram independently of the level studied. The dendrogram itself is composed of a set of depths ranging from the lowest to the highest level. What we call the lowest level is the groups to the left of the dendrogram in Fig. 7 (shown in green and labelled as ‘lowest level’): they are the ones that are most similar to each other.

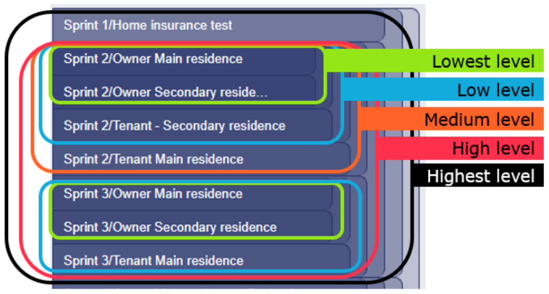

Sprint 1/Home insurance test
Sprint 2/Owner Main residence
Sprint 2/Owner Secondary reside...
Sprint 2/Tenant - Secondary residence
Sprint 2/Tenant Main residence
Sprint 3/Owner Main residence
Sprint 3/Owner Secondary residence
Sprint 3/Tenant Main residence
Lowest level
Low level
Medium level
High level
Highest level

Fig. 7. A dendrogram, showing low-level and high-level groups of tests.

The further the groups move to the right, the fewer similarities there are (the highest level is represented in black on Fig. 7 and labelled ‘highest level’). The lowest levels have the most similarity, so it is efficient to first target these groups and then move on to the higher levels with less similarity. In addition, refactoring test cases starting from the lowest levels sometimes leads to the reevaluation of test cases on higher levels. A slightly higher level test case may become a lowest level test case. As the opposite is not very common, it is even more strategic to start at the low level than at the high level in order to limit refactoring work.

Once the scope of study has been chosen, refactoring and optimization of the tests can begin.

B. Choosing test steps to be refactored

Once a set of test cases have been selected, their test steps can be inspected to see if refactoring is required. Note that we are focused here on *test steps* to be refactored and not test cases, because we first work on all the steps before consulting test cases in detail. This avoids inspecting test cases that are not currently reviewed and may contain inconsistencies. The correction of the steps will provide clarity to the test cases and thus facilitate their inspection later.

The choice of steps to be performed can be made in different ways that we will describe. As before, we recommend using the Automatic Matcher on the chosen scope first, as it quickly eliminates redundancies and simplifies the scope to be studied. Then, different options appear to determine which step to begin with. We recommend using test visualization via the business process diagrams, because it is a significant support for refactoring operations as well as for the acquisition of skills on the test repository studied. Indeed, a crucial aspect of refactoring operations, which is a real challenge, is to build up good knowledge of the test repository. In the context of refactoring operations, familiarity with the test repository is not always present, and yet it is essential to promote the most effective refactoring possible.

This is where workflow is important, as it allows a rapid increase in competence through visual representations. Figure 8 shows the diagram for four test cases, and illustrates how we can quickly describe the related business behaviour: in the context of a subscription, two choices are possible initially – either choose a main residence or a secondary residence. Then for each of these choices we can specify if the insured is owner or tenant, define the number of rooms, and then finally calculate an estimate.

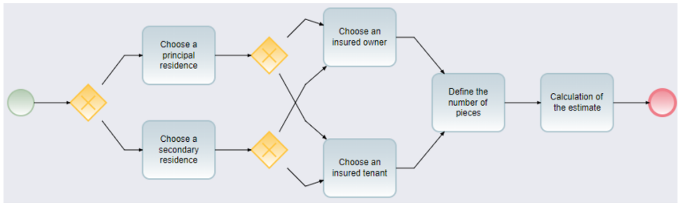

Choose a principal residence
Choose a secondary residence
Choose an insured owner
Choose an insured tenant
Define the number of pieces
Calculation of the estimate

Fig. 8. Business process diagram for visualizing a set of four test cases.

As a result, although knowledge of the business domain and the system under test is a real strength for refactoring test cases, it is less necessary with the help of visual representations.

In addition to helping to improve skills and understanding of the business, the workflow can be the input to the choice of steps to be reworked. The study of the workflow by identifying patterns leads to refactoring by quickly targeting redundancy points. The most useful pattern to identify is tasks on the same vertical line. Fig. 9 shows a diagram where we have highlighted a column of test steps by a red rectangle. Several redundancies can be identified between these four test steps: '**Choose a principal residence**' and '**Choose a principal residance**' are two identical tasks, but one is misspelled. This type of error is easily identified and can therefore be quickly corrected. On the other hand, for the following line '**Choose an insured owner**' or '**Choose an insured tenant**' no refactoring is necessary, because these are two distinct actions. The elimination of redundancies in the workflow leads to the quick clarification of the visual representation as described above.

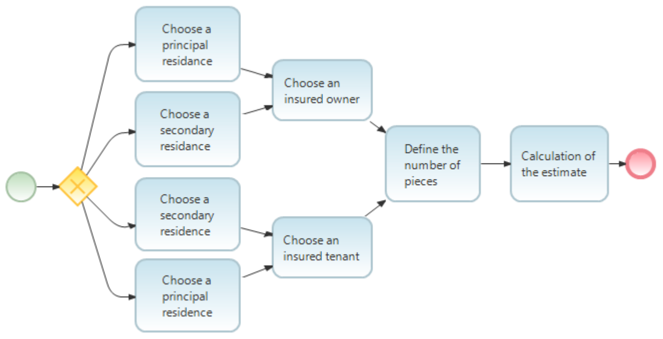

Choose a principal residance
Choose a secondary residance
Choose a secondary residence
Choose a principal residence
Choose an insured owner
Choose an insured tenant
Define the number of pieces
Calculation of the estimate

Fig. 9. Business process analysis

After a step has been selected for refactoring, there are several refactoring approaches that can be applied, as discussed in the following subsections.

C. Homogenize test steps

The selected test step is displayed as the 'Selected keyword' in the tool and one can apply a set of operations (deleting, moving, updating, parameterization, etc.) to this test step. Even if a range of options is possible, we recommend starting by unifying, optimizing and correcting test steps before moving up to the level of operations dedicated to whole test cases.

For a selected step, the tool proposes a set of steps more or less similar to the chosen step. By default, the tool performs its analysis on the chosen scope, that is, on the selected group of test cases. As a result, homogenization and correction are carried out at the local scope level. However, during this phase we recommend changing the working scope to be as global as possible, to avoid redundant work. During our evaluation experiments, we noticed that, in general, when a correction had to be made at a small local scope, it also had to be made at a larger scope. The initial phase of refactoring is longer if a larger scope is used, because it requires the analysis of a larger number of closest matches. Nevertheless, using a larger scope can save a significant amount of time for the rest of the work. It makes it possible to harmonize other perimeters in advance of later phases. On the other hand, if the objective of refactoring is not to work on other parts of the test repository, then keeping the working scope in the group is a right choice.

Orbiter displays the number of occurrences of the chosen keyword. This information is essential, because when updating the step, we must keep in mind that the modification will be performed for the number of occurrences indicated. It is therefore suggested that before a refactoring operation is made, one should investigate where the keywords are called from, in order to know the context of use of the step. Thus, once the choice of the working scope has been made, we can consult the closest matches in order to homogenize and correct, if necessary, the steps close to the chosen keyword.

Be aware, it is important not to apply any parameters at this time. Indeed, it is more appropriate to apply parameters once you have visibility on the test cases. Depending on the context and needs, the application of parameters is not necessarily appropriate, we will detail the use of parameters later.

A recommended way of choosing a good test step for applying homogenization refactoring is firstly to select one of the steps proposed in the closest matches, then identify if that steps needs refactoring. Then look at the visualization of the test cases that contain that step and closely related steps using the business process diagram view, and from this diagram choose related steps that need refactoring. Continue by alternating between these two approaches.

This homogenization phase can be considered complete when the visual representation is simplified and the proposals in closest matches do not allow new update operations to be carried out.

D. Refactoring test cases

Once the homogenization phase of test steps is completed it is time to work on whole test cases, to correct, homogenize and optimize them. The consultation of test cases has different objectives and makes it possible to perform different operations, in particular to modify steps in their context of use.

For each scenario, it is recommended to read the steps and select them, if work is to be done on them. As mentioned above, it should be kept in mind that keywords are potentially used several times and therefore updating should be considered according to the call contexts. When consulting the test cases, it is possible to perform refactoring work as mentioned above as well as reordering the steps. The reorganization of the steps can involve moving the step, deleting it and splitting it up. The choice to perform these different operations depends on different factors. The easiest action to identify is the step deletion. For a selected keyword, which appears in several occurrences and whose use is not relevant, it is interesting to delete it in order to reduce the number of steps in the repository and thus reduce the execution time of test cases. Then, it is important to identify steps that are too dense and contain too many actions. These steps are often time-consuming to execute and in case of failure, traceability with the source of the error is more complex to identify, as several actions have been performed. It is therefore suggested to split the steps and avoid a sequence of actions in the same step. This operation increases the number of steps in the repository, but unifies it by splitting the actions.

E. *Applying parameterization*

The application of parameters is the last step to be taken in the refactoring and optimization of existing test cases. The use of parameters must be done with care, because misuse of them can lead to a loss of knowledge about the test repository. An example would be to confuse two types of data, an A data and a B data that require different processing. If we refactored such test steps into a single parameterized step, we treat these data in a similar way, which may confuse readers – it would be better to keep the test steps clearly separate.

For this and other reasons, we recommend applying the parameters at the end of refactoring, because it is at this time that we have the best understanding of the test cases. In addition, parameterization is easier to carry out thanks to automatic proposals for applying the parameters. When adding a parameter, the tool proposes a list of keyword matches that are good candidates to apply the same parameter as well. The problem is that if parameterization is applied before reaching a certain level of uniformity, the proposed keywords matches are not relevant, as they potentially contain business errors, misspellings, etc.

Following the above recommended process allows us to simplify, rework and optimize existing test cases step by step in the most efficient way possible. The next section will present our evaluations of this approach.

VI. Industry Evaluation

The evaluation of the refactoring approach presented in this paper was carried out on three different projects, in the context of the maintenance of large applications in the railway domain. The total size of the existing test repositories was several thousand test cases, which are documented in natural language. We applied assisted refactoring techniques on subsets ranging from several dozen test cases to several hundred test cases (703 test cases in the largest subset) depending on the application and scope we had to cover. The objective of the refactoring performed was at several levels: reduce existing redundancies to reduce the number of test cases; identify steps performing the same action to merge these steps into a single test action to facilitate maintenance; and parametrize these steps to prepare for possible automation.

A. *Research Questions*

The questions we would like to answer are as follows:

- To what extent does this refactoring process and tool-support provide an efficient solution to support the optimization and refactoring of existing manual test cases?
- To what extent does this refactoring approach fit into an agile development approach, and can it be efficiently integrated into development cycles with short iterations?

B. *Evaluation Metrics*

In order to test the effectiveness of our approach, we used the following set of metrics:

- Time (in hours) taken to perform refactoring with Orbiter;
- An evaluation of the time required to perform refactoring without tools;
- The initial and final counts of tests, steps, and keywords, in order to calculate the percentage reductions.

Our estimation of the manual refactoring time (without using tools like Orbiter) is based on several input parameters:

- the refactoring times for a simple, medium and complex test step;
- the number of test cases in the test suite;
- the complexity of the test suite (based on an index chosen by the user);
- the refactoring objective (a percentage chosen by the user, giving an estimate of the standardization they would like to achieve);

The complexity and refactoring-objective numbers are used to adjust the refactoring time for typical steps (easy, medium and complex). The higher the complexity and refactoring-objective numbers, the longer each step will take to refactor, and vice versa. Once these times have been calculated, with the total number of steps, the percentage of refactoring targeted, and the refactoring times of the steps, we are able to estimate the total refactoring time without tools for the whole test suite. The overall complexity of test referentials varies. They all have a variable number of test cases, with test cases of complexity ranging from simple, medium to complex. For each of the referentials, a complexity of 1 to 5 has therefore been defined. (visible in the table I) To conduct the evaluation, the process described in Section 3 was followed. On each experiment an expert of the Orbiter tool and a functional expert on the application worked together.

C. *Evaluation Results*

We studied a range of different test repositories in order to test the tool on heterogeneous environments. The various experiments were conducted on real existing test cases associated with personnel and rolling stock management applications on a French railway company and containing a panel of varied test cases of different levels of complexity.

We performed nine refactoring experiments using Orbiter, with the results shown in Table I. There was an average decrease of 14% in test cases, 18% in the number of steps and 22% in keywords. We estimated an average time saving of 42% with the tool compared to an approach without.

However, these results vary across the different experiments. For example, the reduction in the number of test cases ranges from 0% to 90%. This is explained by the fact that the objectives and degree of maturity of the test repositories are different. When refactoring test suites used for non-regression testing, there was an average decrease of 30% in the number of test cases, which was higher than the general average. For these non-regression test suites, the average reduction in test steps and in keywords was also higher (38% and 43% respectively), and the time required to carry out the tool-based refactoring was halved (53% decrease compared to manual refactoring).

Exp#	Testing Type	#Tests	Δ%	#Steps	Δ%	#Keywords	Δ%	Manual (mins)	Orbiter (mins)	Δ%	Complexity
1	non-regression	11	0.0	278	-31.6	151	-48.3	104.0	45.0	-56.0	2
2	non-regression	9	-33.3	334	-29.9	201	-13.4	52.4	20.0	-61.8	1.5
3	non-regression	63	0.0	1380	-7.2	920	-18.8	565.8	330.0	-41.7	3
4	functional	44	0.0	350	0.6	301	23.9	61.2	40.0	-34.7	1
5	functional	39	0.0	369	0.0	313	-14.7	49.0	30.0	-38.8	1
6	functional	547	-1.3	4105	-2.5	2473	-5.9	5583.3	205.0	-96.3	1
7	functional	302	0.0	6212	0.0	1444	-28.7	956.6	600.0	59.4	1
8	functional	703	-4.0	16648	-2.0	5440	-4.2	782.9	338.0	-56.8	1
9	non-regression	31	-90.3	402	-84.6	249	-92.8	726.3	338.0	-53.5	4
	Average:	194.3	-14.3	3342.0	-17.5	1276.9	-22.6	986.8	216.2	-42.2	1.75

TABLE I
EXPERIMENTAL RESULTS OF REFACTORING USING ORBITER. THE Δ% COLUMNS SHOW THE PERCENTAGE OF CHANGE IN THE PRECEDING COLUMN(S).

On the other hand, for experiment 8 that worked on a large number of test cases, without reviewing all the test cases or by taking small test suites (less than 50 test cases), different results were found. In this experiment there was no specific objective, but the goal was just to try to correct the test cases. Lower results were obtained: 4% reduction in the number of test cases, 2% reduction of steps and 4.2% of keywords. The time saved compared to a manual approach was 57%. These results are explained by the fact that on a large suite of test cases, very few are reviewed, and on those smaller number of test cases, only the steps are modified without necessarily trying to delete them.

Our conclusion from these experiments is that in all the experiments, with a range of different refactoring objectives, Orbiter saved time compared to a manual approach, but the time savings varied from 34% to 96%. Depending on the fineness of the refactoring objectives, it is possible to obtain good refactoring results. This is done by simplifying the test repository by reducing the number of test cases and steps, and thus obtaining a gain in uniformity through the reduction of the number of keywords.

The tool therefore provides a solution to support the optimization and refactoring of existing test cases. In addition, we found that the tool support can be integrated into Agile development modes by adapting effectively to cycles with short iterations, as the time spent using Orbiter can be shorter because it is being applied iteratively to different subsets of the test repository. On average, it takes 4 hours to perform refactoring on a given scope.

The advantage of using this technique is that it provides significant visibility on the tests through the graphical representation, quickly indicating redundancies and errors. However, even if this technique is faster than any other approach, it requires in an industrial context to be included in existing cycles.

D. Threats to Validity

Better results are always observed on smaller test suites (less than 65 test cases, which represents 66% of the test repositories reviewed). This is explained by the fact that on a reduced set we are able to generally review all test cases, whereas on a large scale (at least 300 test cases) we do not review all test cases. One point of attention would be to extend the experiments and try to cover larger areas as a whole. This would allow more test cases to be treated, in order to obtain more detailed results and not mainly targeted at smaller areas.

Finally, our estimations on the time required to complete the manual refactoring are based on a calculation that includes a set of metrics, and the validation of testers who know the test repository. Although the times evaluated seem consistent, it does not replace a real experimentation in a manual way with the recording of the time spent. The absence of manual comparison was due to the difficulty of having the experiment carried out by two different people (to avoid bias and to reduce the learning effect) who would have had the same degree of maturity and domain knowledge. This would also have doubled the costs and time spent on the refactoring, which was not acceptable in the industry context.

VII. CONCLUSION

A recent survey, 'The state of Software Testing - Report 2019' [17], shows that in the context of enterprise information systems, in more than 60% of cases, less than 50% of functional tests are automated. The rework and optimization of manual test cases is therefore a very common situation faced by testers. This is especially true when migrating a system or application to a new platform, or when existing manual test cases need to be automated.

The approach presented in this paper implements several clustering and natural language analysis techniques to support and partially automate test step and test case refactoring activities in order to improve the maintainability of existing test cases. The results obtained in terms of refactoring quality and productivity gains show the significant contribution of the tool support provided compared to refactoring without any tools. We have therefore begun to extend the use of the approach so that it can be used in a routine way on projects requiring refactoring of existing manual test cases.

REFERENCES

[1] CFTL, "Observatoire 2019 sur les pratiques des tests logiciels en France (2019 survey on software testing practices in France)," 2019, available from http://www.cftl.fr/wp-content/uploads/2019/04/JFTL-2019-Enqułte-CFTL-2019.pdf.

[2] S. U. R. Khan, S. P. Lee, R. W. Ahmad, A. Akhunzada, and V. Chang, "A survey on test suite reduction frameworks and tools," *International Journal of Information Management*, vol. 36, no. 6, Part A, pp. 963 – 975, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0268401216303437

[3] M. Alian, D. Suleiman, and A. Shaout, “Test case reduction techniques - survey,” *International Journal of Advanced Computer Science and Applications*, vol. 7, pp. 264–275, 06 2016.

[4] A. Gupta, N. Mishra, and D. S. Kushwaha, “Rule based test case reduction technique using decision table,” in *2014 IEEE International Advance Computing Conference (IACC)*, Feb 2014, pp. 1398–1405.

[5] Z. Chen, B. Xu, X. Zhang, and C. Nie, “A novel approach for test suite reduction based on requirement relation contraction,” in *Proceedings of the ACM Symposium on Applied Computing*, 01 2008, pp. 390–394.

[6] A. Coutinho, E. G. Cartaxo, and P. D. Machado, “Test suite reduction based on similarity of test cases,” in *7st Brazilian workshop on systematic and automated software testingCBSoft*, 2013.

[7] B. Subashini and D. JeyaMala, “Reduction of test cases using clustering technique,” *International Journal of Innovative Research in Science*, vol. Engineering and Technology Vol 3, no. Special Issue 3, pp. 1992 – 1995, 2014.

[8] H. M. Sneed, “Testing against natural language requirements,” in *Seventh International Conference on Quality Software (QSIC 2007)*, Oct 2007, pp. 380–387.

[9] H. M. Sneed and C. Verhoef, “Natural language requirement specification for web service testing,” in *2013 15th IEEE International Symposium on Web Systems Evolution (WSE)*, Sep. 2013, pp. 5–14.

[10] A. Ansari, M. B. Shagufta, A. Sadaf Fatima, and S. Tehreem, “Constructing test cases using natural language processing,” in *2017 Third International Conference on Advances in Electrical, Electronics, Information, Communication and Bio-Informatics (AEEICB)*, Feb 2017, pp. 95–99.

[11] S. Gnesi, G. Lami, and G. Trentanni, “An automatic tool for the analysis of natural language requirements,” *Comput. Syst. Sci. Eng.*, vol. 20, 01 2005.

[12] A. E. V. B. Coutinho, E. G. Cartaxo, and P. D. de Lima Machado, “Analysis of distance functions for similarity-based test suite reduction in the context of model-based testing,” *Software Quality Journal*, vol. 24, no. 2, pp. 407–445, 2016.

[13] V. I. Levenshtein, “Binary codes capable of correcting deletions, insertions, and reversals,” *Soviet Physics Doklady*, vol. 10, no. 8, pp. 707–710, feb 1966.

[14] W. E. Winkler, “String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage,” *Proceedings of the Section on Survey Research Methods*, pp. 354–359, 1990.

[15] G. Bisson and R. Blanch, “Stacked trees: A new hybrid visualization method,” in *Proceedings of the International Working Conference on Advanced Visual Interfaces*, ser. AVI ’12. New York, NY, USA: ACM, 2012, pp. 709–712. [Online]. Available: http://doi.acm.org/10.1145/2254556.2254690

[16] Smartesting, “Orbiter,” 2019, available from http://orbiter.smartesting.com/.

[17] PractiTest, “State of testing survey 2019,” 2019, available from https://qablog.practitest.com/state-of-testing/.