Supporting efficient test automation using lightweight MBT

Elodie Bernard, Fabrice Ambert, Bruno Legeard

Supporting efficient test automation using lightweight MBT

Elodie BERNARD*†, Fabrice AMBERT*, Bruno LEGEARD*‡
*FEMTO-ST Institute, Univ. Bourgogne Franche-Comté, CNRS Besançon, France
† Sogeti, Lyon, France
‡ Smartesting, Besançon, France
[elodie.bernard|fabrice.ambert|bruno.legeard]@femto-st.fr

***Abstract*—The Agile and DevOps transformation of software development practices enhances the need for increased automation of functional testing, especially for regression testing. This poses challenges both in the effort that needs to be devoted to the creation and maintenance of automated test scripts, and in their relevance (i.e. their alignment with business needs). Test automation is still difficult to implement and maintain and the return on investment comes late while projects tend to be short. In this context, we have experimented a lightweight model-based test automation approach to address both productivity and relevance challenges. It integrates test automation through a simple process and tool-chain experimented on large IT projects.**

I. Introduction

Software development and delivery drive organizational and business performance in a wide range of industrial and application sectors, for a growing number of companies. Accelerating the pace of bringing software changes to production is a key element of competitiveness. This acceleration is now supported by a continuous approach to development and deployment (DevOps), and requires a deep transformation of development processes, technologies, practices and organizational culture. Testing of IT Systems has become a major bottleneck for many large companies and SMEs. Besides the ever-growing complexity of software systems, their indispensable quality requirements have led to dramatically increased verification and validation costs[1]. A fine-grained analysis of existing testing procedures reveals however lack of alignment between the business view of testing and implementation in automated testing[2].

The approach presented in this paper addresses the dual question of the relevance of automated tests and the productivity in their creation and maintenance: how to improve the efficiency of automated functional tests by aligning a business-oriented vision of the workflows to be tested and the implementation of test actions?

Model-Based testing (MBT) has been an important research focus for at least two decades, but as stated by Arcuri [1], despite strong academia effort and intensive research work, these techniques are used rarely in industry, because of the complexity of using symbolic model-based approaches for software engineers. Our research work aimed at using simple workflow notation to link workflow modeling, intuitive for functional testers, and keywords to automate low-level test actions (at GUI or API level). The main contributions of this paper are:

1) a lightweight modeling approach for test generation.
2) the management of several abstraction level to facilitate the alignment of tests with business needs.
3) the mapping between high level test action and low level keyword.
4) the management of abstract and concrete test data.

In the rest of this paper, we start in Section II by presenting the related state of the art, then in Section III we describe in detail the lightweight MBT as we use it. In section IV, the test automation process is detailed, section V discuss about some best practices to reduce maintenance effort on automation artifacts. The section VI illustrate the use of these practices through the presentation of two experiments in real context. Section VII presents some conclusion and point out future work.

II. Related work

Model-based-testing (MBT) and keyword driven testing (KDT) are two techniques commonly used in the field of testing. In the literature, some papers have been written about these techniques in a decorrelated way, i.e. only MBT techniques and for others only KDT techniques. We have therefore studied on the one hand the papers dealing with these two subjects independently and on the other hand those combining them. In order to respond to the problems and challenges of time to market and automation we study recent work that adopts similar approaches to the one presented in this article.

- Lightweight model: Business Process Model and Notation (BPMN) rather than UML
- Automation: Data driven / Keyword driven

In the context of the MBT, several notations, textual or graphic, can be used to express the model [2]. Among the graphic models, some are attached to the representation of the system under test by a state machine (Statechart for example). Others, and the work presented here, focus on process modelling. For this purpose, BPMN models are often

[1]see World Quality Report 2019-2020 - Available online: https://www.sogeti.com/explore/reports/world-quality-report-2019.

[2]see State-of-DevOps Report 2019 - Available online: https://services.google.com/fh/files/misc/state-of-devops-2019.pdf.

used. In [3], the authors present the ETAP-Pro platform. The system under test is expressed through a BPMN model. The tool then evaluates the structure of the model and determines the execution paths in the model. Each path and activity in the BPMN model is identified by a keyword to facilitate its reuse. The tool then produces a set of test cases, in the form of Gherkin scripts, to cover the identified paths. The scripts in Gherkin language, are then completed by the test automation engineer in order to be executed. The test execution verdicts are injected into the BPMN model by colouring the different elements. There are also another work on BPMN [4], which focuses on test case generation in an XML format. Our modeling approach is close to BPMN but allowing the generation of test cases directly in test tools or more simply in Excel format (see next section). For KDT, the objectives frequently put forward are the facilitation of automation by facilitating the management of the scripts [5], [6]. These two solutions only manage the KDT part and not the MBT part, but today more and more solutions tend to combine the two by integrating test generation, as which evokes the integration of MBT in these future works [7]. Nevertheless, MTB and KTD are already often combined as presented above for the ETAP-Pro platform, and other [8].

III. Lightweight test modelling

The adoption of MBT by industry is progressing slowly, and at the same time, the need for better and more automated software testing methods grow. So we believe that experiment and develop new methodologies and good practices will strongly help testers to support efficient test automation.

The main objectives of our approach of lightweight test modelling [9] is to simplifying the modelling notation and increasing the user experience of the MBT. To achieve this, we use an MBT tool: Yest, produced by Smartesting.

Yest is a test design and implementation tool based on visual representations of workflows and business rules. It supports an efficient software testing process for both manual and automated test execution.

In our tool, the approach is to represent the workflow to be tested (and only the part to be tested) and to keep it as simple as possible depending on the test objectives (Figure 1). In the same way, the tool provides a test case generation based on the coverage of nodes and paths, allowing to cover only what is specified by the user, without proposing all possible combinatorics.

To guarantee the simplicity of use of our tool we work with a limited number of modelling elements (Figure 2). There are three different kinds of inner nodes: task, choice point and subprocess. In the case of external nodes, one of them is the start point and the others are ending points. All nodes are linked by connectors to describe the flow. Task nodes describe actions on the SUT while choice points control the flow within the workflow. Subprocess is used to introduce a new process as a node of the current process. The subprocess has its own flow graph. Within a workflow, the tasks and choice points are

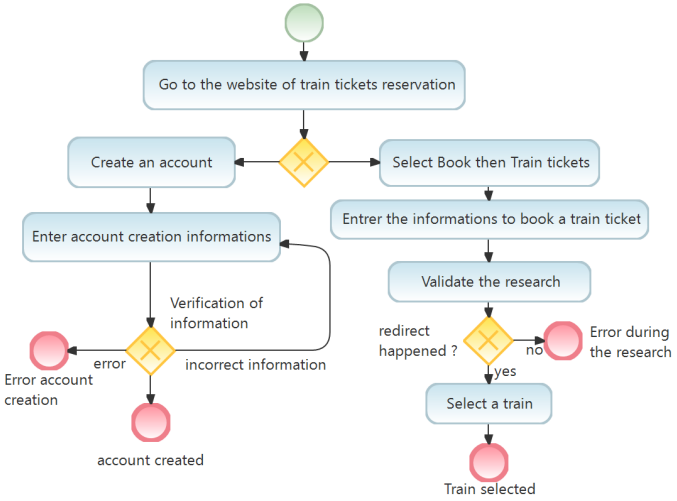

Fig. 1. Example of workflow

associated to decision tables witch detail the cases to be tested and manage the flow for test generation.

Artefacts	Descriptions
	Start point: marks the start of the process
	End point: marks the end of the process
	Task: describes actions to execute on the SUT to test a behavior, and corresponding expected results
	Choice point: select the appropriate following flow depending on previous actions
	Sub process: creates a sub process
	Subset: defines a subset
	Connector: joins the different artefacts

Fig. 2. Artefacts of the Lightweight MBT tool

In order to increase the user experience of the MBT tool we will present in this paper a set of methodologies and good practices to facilitate the automation process. We work notably to improve the use of the MBT approach by keeping a certain level of abstraction during the modelling phase and by facilitating the transition between test cases and test scripts.

IV. Test automation process

This section describes the automation process implemented (Figure 3). Although there are many different kinds of automation technologies, the process described below can be applied to different types of software applications: web, desktop, mobile apps. The first step is to model the test process (see Figure 4), the objective is to make a visual representation of the system under test by integrating requirements, data and other relevant elements. This makes possible the automatic production of a test case that can be used for manual testing. If we want to limit the test process to manual tests, the process can be stopped here, otherwise we can continue by completing the adaptation layer in order to automatically produce test scripts.

Our test automation approach is based on keyword-driven. The test cases generated by the visual representation are

converted into automated test scripts, where the test actions described in the model are transcribed into test automation keywords with parameters (the test data).

We will describe the automation process using an example.

Fig. 3. Test automation process

A. Modelling the process

We want to test a set of behaviour in an ERP-based application focused on project management. Here is a list of requirements that should be tested:

- It is possible to create a project.
- It is possible to consult all my projects, know their name, status and date of creation.
- It is possible to consult for each of my projects a time line indicating the phase in which the project is located (plan, design, develop, or complete).
- It is possible to change the phase of my project and observe the progress on the time line.
- It is possible to consult the tasks on each of my projects.
- It is possible to consult the latest activities on the home-page.
- It is possible to consult the projects dashboard, in the dashboards, and to have a summary of the number of projects, and the percentage of complete project, design, develop, plan.

Figure 4 shows the workflow designed with the modelling tool in order to produce the test set covering these requirements.

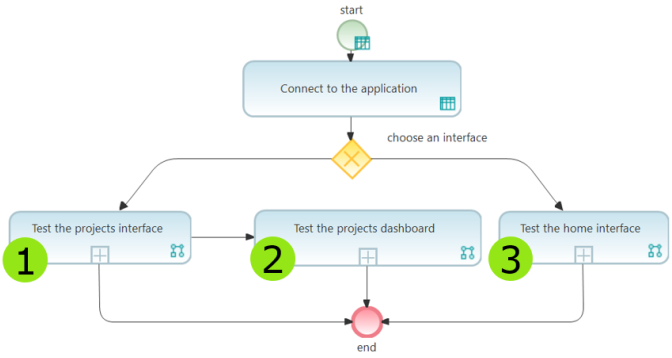

Fig. 4. Main workflow

According to the requirements, we can observe that the workflow test the different part of the application: the projects interface, the projects dashboard and the home interface. Each element annotated with a number (1, 2 and 3) is a sub-process describing in more detail the test step through tasks and decision tables. Figure 5 details the “Test the projects interface” sub-process (number 1 in figure 4): Starting by opening the project interface, then creating a new project, and checking if the project appeared in the project’s list, with the expected data. Next, the project stage can be changed and checked in the project’s list.

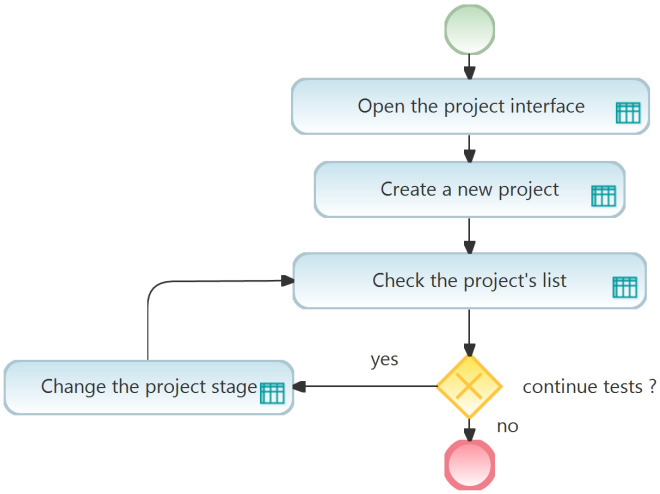

Fig. 5. project workflow

B. Automatically produce test cases

Once workflows has been modelized, the tool provides test cases, based on these and decision tables. Even if the tool offers the possibility to produce test cases, the user can change everything in these proposed test cases or can create new ones. In Figure 6, there is an example of a test case produced by the tool, it covers in figure 4 the action ’Connect to the application’ then ’Test the projects interface’. Words in blue in the test case are design data values . For example, the data ”stage” here is set to ”Plan”.

	Actions	Expected results
1	Connect to the application	The application is running
2	Open the project interface	The project interface is opened
3	Create a new project : Choose the project name : **Automation project** The status : **Not Started** The stage : **Plan** (Data : stage, Data value : Plan)	The project is created
4	Check the projects's list	The project **Automation project** is in the projects's list. Its status is **Not Started** Its stage is **Plan**
5	Check the project pipeline	The project pipeline is in line with the project stage (**Plan**)
6	Check the project date of creation	The project date of creation is today
7	For the project **Automation project** Change the project stage to **Design**	The project stage is changed
8	Check the projects's list	The project **Automation project** is in the projects's list. Its status is **Not Started** Its stage is **Design**
9	Check the project pipeline	The project pipeline is in line with the project stage (**Design**)

Fig. 6. Test case to be automated

The test cases produced by the model can be converted into test scripts. To do this, each test action (in figure 6, number from 1 to 9) will be linked to one or more keywords. In the same way, the abstract data sets (design data) in the test cases are transformed into concrete data in the scripts. To generate

the test scripts, the tool supports the mapping of each test action with a concrete automation keyword. More details are given in section IV-C2 where the link between step 3 (see figure 6) and a keyword is shown.

C. Complete the adaptation layer

In order to complete the adaptation layer, a set of steps is defined and we will present each of these steps.

1) Define a Keywords dictionary: The first step to complete the adaptation layer is the keywords definition. These keywords are set in an Excel file using a table, which includes the names of the keywords and their parameters.

Class	Keyword	param1	param2	param3
Keywords	createNewProject	projectName	projectStatus	projectStage

Fig. 7. Keywords dictionary

The Figure 7 contains the definition of the Keyword “createNewProject” with three parameters: projectName, projectStatus and projectStage. It will be used along the example.In the tool, we load the keywords Excel file, then it is possible to map the different actions of the scenarios with the adapted Keywords.

2) Link the action with the automation keywords: In Yest, the first step is to realise the mapping between test actions and the automation Keywords. Here is an example of the mapping for action 3, ”Create a new project” of the test case in figure 4:

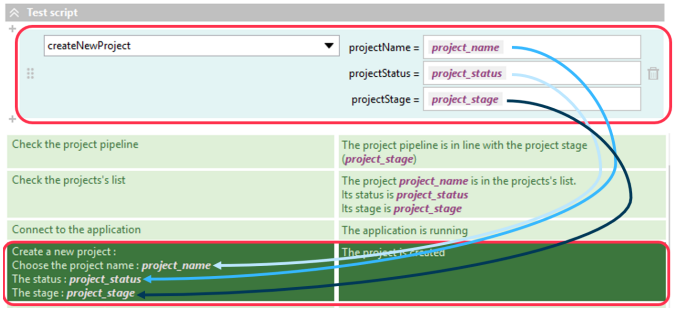

Fig. 8. Mapping with the automation keywords

Each action is linked with an automation Keywords, the action highlighted (in the Figure 8) is associated to the automation Keyword “createNewProject” with three parameters the project’s name, the status and the stage. And for each parameter of the keywords we will be able to link the parameters of the test case directly: the values of the design data (as it is the case here in the figure above), or with other values.

To use other values: two options are possible, choose new values or implementation data. For new values, it is very simple: here, during the completion of the adaptation layer, instead of using ”project_name” for the ”projectName” data, you can directly write in the cell the data value for ”project_name”. In the Figure 9, on line 3 a new data value has been used: ”my project”. Then to use implementation data, a data set must be created in the tool (shown in the Figure 9). For each design data value ”project_name” we associate an implementation data value ”project_name_db” representing base ids. These are real data from the application’s databases and that we put here by a copy-paste of an extract of the application. It is thus possible to use instead of the design data as it is the case on line 1, the implementation data corresponding to it, as it is done on line 2 using the ”project_name_db” data.

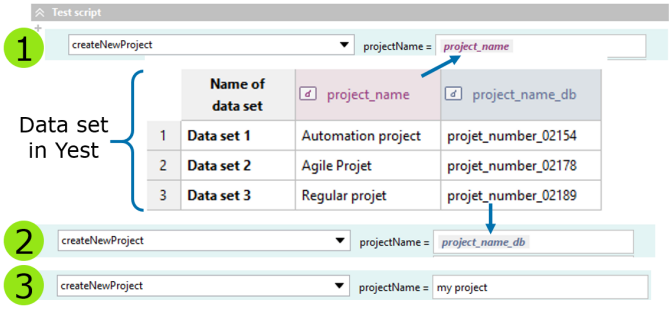

Fig. 9. Mapping data in the keywords

3) Code the automation Keywords: Once the mapping between test action and keywords has been performed or in parallel it is required to code the automation Keywords. We use Java and Selenium to concretize the Keywords.

```
public static void createNewProject
        (String projectName, String projectStatus, String projectStage){
    driver.findElement(By.id("AddNewButton")).click();
    driver.findElement(By.id("PROJECT_NAME")).clear();
    driver.findElement(By.id("PROJECT_NAME")).sendKeys(projectName);
    new Select(driver.findElement(By.id("PROJECT_STATUS"))).
            selectByVisibleText(projectStatus);
    new Select(driver.findElement(By.id("PROJECT_PIPELINE_ID"))).
            selectByVisibleText("Project Pipeline");
    new Select(driver.findElement(By.id("PROJECT_STAGE_ID"))).
            selectByVisibleText(projectStage);
    driver.findElement(By.id("btn-save")).click();
}
```

Fig. 10. Example of Keyword code

An automation Keyword (in Java code) is a sequence of actions performed on the SUT and is corresponding with a specific Keywords used in the test cases. The Figure 10 takes an overview of a keyword implementation. The first action of the Keyword above is to click on the button allowing to create a new project. Then entering the project’s name, status and stage, and finally click on the save button. Each automation keyword is used to run the test script.

D. Automatically produce test scripts

Finally, once the keyword code produced, and each test action of the test case is connected with one or more automation keywords, the generation of test scripts is possible. The Figure 11 is the script produced by Yest with the associated test case. It is substantially similar to the test case presented in

the Figure 6 which is the test case chosen for the automation. The script is the sequence of keywords corresponding to the mapping performed previously. Each green number preceding code lines correspond to the action on the test case. A Keyword can cover several actions (as for the actions 4, 5 and 8, 9). The blue number correspond to special Keywords call, not directly in line with the test case but useful to the execution of the script. Their calls are set up in Yest and directly generated, no additional operations on the script has to be done to allow the script execution. In the tool, the production of the script is done quickly once all the previous steps are completed. A simple click allows the generation of the script in the runtime environment (via the path of a Java project, for the use of Selenium, or directly in UFT or Ranorex solution for these technologies).

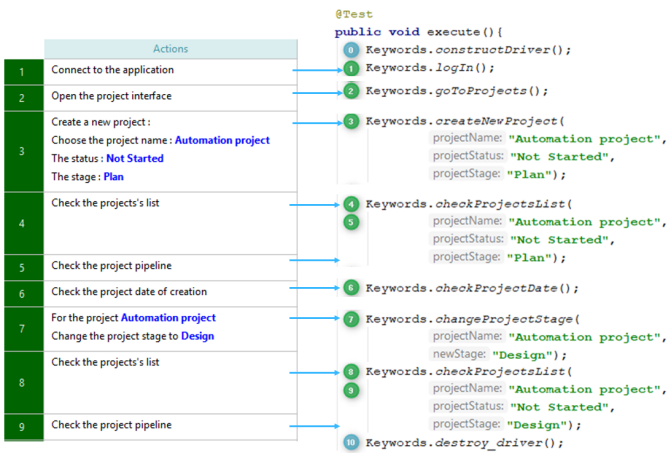

Fig. 11. test script

V. Best practice to reduce maintenance workload for automation artifacts

In order to maintain a set of automated scripts it is necessary to managing different kind of artifact, such as data, models, object repository, keywords and scripts.

A. Managing test data

Test data have different levels:

- Data from the model: this is the data derived from requirements, or useful to manage the workflow (data who causes changes in the process).
- Data in Keywords: this is the data derived from test cases (Data from the model), or added by the tester for automation purposes.
- "Implementation data": this is the data extract from the database application.

The main challenge in test data management, in the context of automation, is to ensure the link between these different levels of data.

1) Data from the model: This kind of data are created in the model by decision tables, they are called design data. Each new column in decision table creates a new data (if the column does not exist in a previous table) and each new row creates a new data value. These values are visible in a dictionary that provides an overview of all the data used in the workflow. In this way, it is possible to manage the use of data globally, in the workflow as well as in test cases. The modification of the data values in the dictionary is allowed, which permit to update both the data values in the paths and in the existing test cases. In order to limit the volume of data, it is a good practice to define only the data that are necessary, i.e. data derived from business needs or useful to manage the workflow. Typically, in the test process, if there is a connection phase requiring a login and password, if the value of their data is not relevant, they should not be present in the table. However, if there are requirements for the ID and password, the data can be defined in the table.

2) Implementation Data: The difficulty with implementation data is that it cannot always be used directly in test cases because it is not always persistent. For example we want to check an age in an application, let's say older than 25 years. This data in the application is a date in mm/dd/yy format. If we want to control the user's age when registering, each year the date of birth will have to change to cover the acceptance criteria.

However, in the test case, the data can be directly set to "25 years old" and thus be perennial. Here the example is simple and the correspondence between the design data "age" and the implementation data "date_of_birth" is rather easy to do. In other cases this exercise can be rather more complex.

In all cases it is crucial to be successful in linking the design data and the implementation data. With the Yest tool, this is possible. After generating the test cases, it is possible to link each design data with an implementation data through data sets created in the tool. In this way a clear view of the traceability between the data in the test cases and the data in the SUT is kept.

3) Data in the keyword: Keywords manipulate 3 types of data:

- processing data
- design data: from test cases
- implementation data: from the SUT

Each of these data can be passed as a parameter to each of the keywords. The processing data is not or only slightly dependent on other data. They are really specific to each keyword and are designed to make particular treatments. (Logging, passing of particular parameters, etc.) On the other hand, design and implementation data are closely linked and have different roles. Design data can be used as it is in keywords or can be transformed to allow business rule processing. On the example of the control on the data "age" with the values "more than 25 years" or "less than 25 years" this can lead to two different treatments. For the implementation data, they can be used with little or no transformation. The data in the keywords is the result of design and implementation data management.

Without an accurate view of the data in the SUT and in test cases, it is complicated to manage data in keywords. With Yest you can choose to pass design data or implementation data as keyword parameters depending on the operations you want to perform. Moreover, this choice is made at the step level and not at the full test case level in order to ensure granularity of data. The advantage here is that one can directly link the data in the test design tool and in the keywords for automation. This advantage is not negligible because the tool allows to generate the test scripts and thus update the data used in the keyword parameters.These data and their interactions are summarized in the Figure 12.

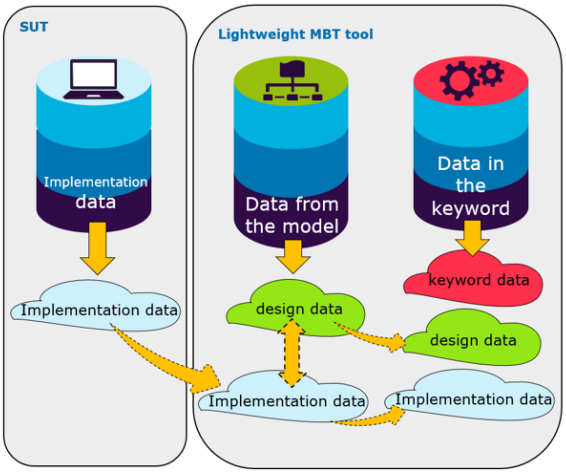

Fig. 12. Overview of data management

In order to manage the above-mentioned data, we apply a precise process. First of all, it is important to define the design data in Yest. These data can be abstract data and allow to cover the business requirements, without necessarily having concrete values existing in the SUT. Then a link will be created between each test action and one or more keywords. As mentioned the keywords manipulate 3 types of data: design data, implementation data and processing data. Through datasets in Yest we associate design data with implementation data if required. Once this is done, we can use the design or implementation data in the keywords according to the choices made previously, and add new parameters and their values if needed. Once the Yest adaptation layer is completed, we can publish the test scripts with the adapted data values. In case of changes in the Yest side data, we can quickly update the scripts to keep them up to date. It is this data management that keeps the automated scripts up to date and thus facilitates the maintenance of the automated test case assets.

These data are important because in the context of the improvement of the automation process, certain tests are executed several times with a set of different data. Typically, if we want to test a connection interface, it can be interesting to test the connection with different identifier. If the identifiers are present in the database, we execute the test in a loop to check the different profiles.

The data utilisation begins in the model, at this level this is a functional tester that entered data. So he provides data with the level of abstraction of his choice. He can provide directly data recorded in a database/application (when it is possible) or to choose to enter abstract data. The functional tester does not have to reflect about the automation when he complete his model. However, if in some specific case he can enter the data visible in the application (for example, a text in drop-down list), the gap between the different type of data is small and allow a weak maintenance. More the data in the model are far from the data in the SUT more the data management is complex.

So to enter data in the model as close as possible to the data present in the database/application is a good practice to facilitate the data management. Nevertheless, it does not need to try to enter all the data of the application. It must keep in mind that the model contains uniquely data useful to verify the requirements but not the data useful to execute the test script. There are keywords that contain these data. In data management phase, the keywords have an important role because they allow handling data collected from the test action and converting them to exploitable data in the SUT. Thus, we have to be rigorous with the way to deal with the data in the adaptation layer in order to guarantee a good data management.

B. Models

Regardless of the way a project is built, whether it is in Agile or V-Cycle, our approach is based on building a set of models that will allow us to generate test cases to verify the SUT's functionalities. In order to verify a large application, it is common to build several models that use common behaviors. By common behavior we mean a group of steps: actions and expected results that are similar. Usually this can occur for the connection phase which can be common to several test cases covering different functionalities. The good practice is to integrate in the models a "shared" "connection" sub-process that describes the connection behavior. This "shared" sub-process is a model integrating all the steps necessary to establish the connection to the application and which can be integrated into any other model. In this way, when we create a new model to test a new behavior, it is possible to reuse existing connection processes to build the process to be tested. This technique reduces the time needed to produce new scenarios using existing artifacts. In addition, having a common process for a specific behavior reduces maintenance time. Indeed, if a change occurs on the connection phase, it is easy to modify the process related to the connection and all the processes that use the connection process will be up to date, without modifying each main model one by one. A good practice is therefore to identify the different behaviors of the application and spot the ones that can be used in several test cases and thus create a set of shared sub-processes that can be used several times rather than specific and unique models In the example of the figure 13, in the "One practice" and

”Best practice” area, we consider that in models 1 and 2, the behaviors are similar at the beginning of the process. Instead of creating different sub-processes like in the ”One practice” area (sub-process 1.1 and 2.1) it is better to use the same sub-process (sub-process 1.1) as shown in the ”Best practice” area.

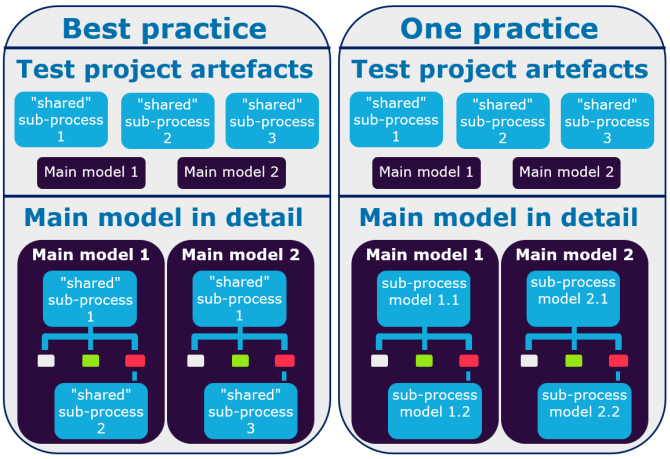

Fig. 13. Models best practice

C. The object repository

The object repository is a library containing all the objects of the SUT required for automation, which are stored in the automation solution. These objects are buttons, fields, forms or any other object on which you want to perform actions to test the application. If we want to test a connection interface, we will have in the object repository the field ”identifier”, ”password” and a button ”submit”. Depending on the automation technology used, the object repository is built differently. Nevertheless, one common point is that the objects must be precisely identified to facilitate maintenance. Usually, the identification of objects is done via a ”path” leading to the object. The path is more or less complex depending on the type of SUT. In the case of web applications, objects are identified by an xpath, or an id (among others). Having an id to identify an object is the best way to facilitate the automation process and its maintenance. Indeed, even if the object moves in the structure of the application, the script will still be able to find an object, unlike an xpath which is volatile and subject to change with application updates. So, in case we don’t have an id, and we have to use an xpath, we have to make sure that the xpath is enough global to find the object if its position changes. However, this solution is weak because an xpath should not be overly global as it can select several objects instead of one and thus lead to incorrect behavior. A good practice to facilitate the creation and maintenance of the object repository is therefore to choose the best identifier for the objects. The best identifier is probably the id. If it is not available, we need to identify the strongest property (i.e. the one that will be the most unique and less subject to change) of the object to identify it. For example, the name of the object, its text, its class or any other property that should not change and select the item uniquely. Then these objects must be organized efficiently in the automation solution to allow easy updating. The objects can be classified by interface or functionality depending on the way the application is implemented.

D. The keywords repository

The keywords are the automation functions to reproduce the behaviours of the SUT. Regardless of the technology used, they have a name, and parameters (data). These and other properties must be managed to facilitate their maintenance. Here are the properties that we have treated in order to facilitate the management of keywords:

1) the name of the keyword: This may seem trivial but it is of major importance in order to facilitate the creation of scripts and their maintainability. In our approach we consider the work of the test analyst and the test automation engineer in a similar way. It is therefore necessary to set up a naming system that allows the test analyst to understand the scripts and the automation engineer to manage his keyword library. The name of the keywords must be meaningful and correspond to the operation he is going to perform. For example ”connect”, ”bookAFlight (Departure, Arrival)” are descriptive names and use data that a test analyst can understand and use to complete the automation adaptation layer.

2) parameters (data): As in the example above for the function ”bookAFlight(Departure, Arrival)” two parameters are used: the departure city and the arrival city. The difficulty in the management of these parameters is that they have to be defined in a suitable format to be the most robust as possible to change. Here the parameters are waiting for a string: there are different properties to take into consideration. If the test analyst chooses ”paris” as the departure city and the SUT waits for ”Paris” it is possible that the test fails because of a management of upper and lower case. In this situation, the automation engineer should perform a treatment on the data passed as a parameter to ensure that a capital letter is present for the first character of the parameter and, if this is not the case, to integrate it. In other circumstances, it should not be managed because the test will be aimed at verifying the property that the SUT must receive the exact name of the city, respecting the upper case at the beginning of the word. The difficulty is the management of the parameters, which values are expected and accepted. For this reason, the test analyst and the automation engineer must work together and define correctly the expected behavior of the functions as well as the format and data values accepted in the parameters. In some cases the automation engineer may set up keywords for field completion in general, such as a ”completeField(”FieldToComplete”, ”FieldValue”)” function where the analyst may enter ”fieldLogin” as the field to be completed and ”myId” as the value of the field to be completed. In this kind of case the test automation engineer will have to provide the values from the object libraries to the test analyst to allow this technique to work. Hence the importance of managing the object repository as mentioned above.

3) the granularity of keywords: Its management is essential to facilitate their implementation and maintenance. The difficulty will be to choose which degree of granularity to implement and when: should we write a connection function, or sequence the completion of fields and the validation of a form to make the connection. In order to best manage this granularity we have established 5 factors that vary according to the granularity: the reusability of the keyword, its maintenance effort, the understanding of the script product (for the test analyst and the automation engineer) and the investment to be made on one hand by the test analyst and on the other hand by the test automation engineer. The following table (Figure 14) classifies the granularity in 3 levels: low, medium and high and according to these levels of granularity the same scales are applied to the factors. In addition 3 colors are used green, orange and red depending on whether the factor is good, medium or bad.

granularity	reusability	maintenance effort	understanding of the script	investment of the test analyst	investment of the test automation engineer
high	high	low	low	high	low
medium	medium	medium-high	medium	medium	medium
low	low	high	high	low	high

Fig. 14. Keyword's granularity and factors

A high level of granularity can be represented by a keyword of the type "completeField("FieldTocomplete", "FieldValue")": it can be used to check various functionalities and fields. Thus we can see that on a high level of granularity the reusability of keywords is high which is very good point, however it requires a high investment of the test analyst. The automation engineer has to create the keyword library but it is limited and requires less effort to maintain. On the test analyst side, he has to use a lot of keywords to detail basic functionality. This leads to poor readability scripts because they only include steps for filling fields and clicking buttons that are not representative of the business behavior behind them. The main advantage of high granularity is therefore that it is reusability.

A medium level of granularity can be represented by a keyword of the type "chooseABookingTool("URLOfBookingTool")": it can be used to check cases where we want to choose a booking site. For example, this function could allow to choose a booking site regardless of the site, therefore the code of the keyword must be adapted to identify the site where the test is carried out and adapted the keyword to allow the reservation on each of them. For a medium level of granularity, all the factors have a medium degree except the maintenance effort, which can be high. The investment of the test analyst and the automation engineer are both moderate because it requires the same effort for the analyst as for the automation engineer. This is explained by the fact that the keywords are enough understandable to bring a part of the business vision, so the scripts are rather understandable. On cons, maintenance effort can be complex because the keywords are not dedicated to a specific context, it is necessary to be careful when updating a particular context to avoid breaking the other contexts.

A low level of granularity can be represented by a keyword of the type "bookAnHotel("Arrival", "Departure", "Hotel")": it can be used to check hotel booking on a particular software. For this level of granularity the reusability is low because a very specific function will not be able to be reused on a regular way in a totally different script than the one for which it was created. Usually low granularity keywords cover a precise perimeter and cannot be used on other features but can be useful for regression testing where testing of features must be sequenced. As the functions have a low granularity level, the understanding of the scripts is facilitated because they contain keywords with meaningful names allowing to understand the actions performed in the script from the business point of view. Thus the investment of the test analyst is limited because it is easy for him step by step to reproduce the SUT's behaviour and complete the adaptation layer but not for the test automation engineer. He will have a wide number of keywords to implement and which will be dedicated to a specific behavior. Moreover the keywords is devoted to a specific behavior, so maintenance effort is harder because keyword updates must be done for each of the test perimeters, for a large number of keywords. The following table (Figure 15) gives an example of functions that can be used for different levels of granularity.

High level of granularity	Medium level of granularity	Low level of granularity
completeField ("URL","bookingURL")	chooseABookingTool ("URLOfBookingTool")	bookAnHotel ("Arrival","Departure","Hotel")
clickOnButton ("validate")	bookAnHotel ('Arrival', 'Departure', 'Hotel')	
completeField ("Arrival, "ArrivalDate")		
completeField ("Departure","DepartureDate")		
completeField ('Hotel','HotelName")		

Fig. 15. Example of keywords by granularity

Thus the best practice to apply here is to intelligently switch between keywords of low, medium and high granularity. The creation of a library of high granularity keywords is beneficial to give autonomy to the test analyst and allow him to automate simple behaviors. When the use of these keywords causes too long or incomprehensible scripts, it is necessary to switch to medium and low granularity keywords. Prefer medium granularity keywords if the function can be used in different contexts and high granularity if it will be dedicated only to the given perimeter. By combining these 3 levels of granularity it is possible to achieve a balance between each of the 5 factors and thus facilitate the implementation and maintenance of the keywords.

E. Scripts

Scripts are probably the most difficult automation artifact to maintain because they are dependent on data and keywords. So if the data or keywords are invalid, scripts cannot work. Indeed, a script is a sequence of keywords that reproduces the behavior of the SUT. If the behavior of the SUT changes, the whole script can be potentially impacted. For example,

if a change occurs just after the connection, all the scripts may need to be reviewed. If a large number of scripts exist, reviewing each script and adding new keywords can be extremely complex. In our experiment, we use Yest to generate test cases and test scripts. We advise you to keep the model up to date. If a new behavior appears in the application (or new data), we add these new behaviors and data to the model. Then we regenerate the test cases, complete the adaptation layer, to add a new keyword and data, and link it to the new test action, and regenerate the test script. This way, we don't have to modify each of our scripts, but simply replace the obsolete scripts with the new ones.

Applying these different good practices helps to maintain the test scripts, and to facilitate the management of the automation process. In the next section, we will show how we apply these methodologies on a large IT project.

VI. Experience report

In the section above we have presented a set of approaches and good practices to facilitate the automation process and maintenance. Here we will describe how we applicate all of them to test a web application on two large IT project. In order to experiment the approach in two different contexts, we carried out a study in one perimeter of an Agile project in the middle of development (after 10 months of projects by restarting at the tests on a perimeter) and in the other case, an Agile context as well but by conducting the experiment on a short cycle of 10 days to see if the approach was suitable for short iterations. In addition, we have used two different automation frameworks: Selenium for the first experimentation and UFT for the second in order to test the approach on two different technologies and make sure that good practices can be applied in both cases.

A. 1^{st} experiment

1) Description of the SUT: In our experience we focus on one aspect of the application: the evaluation process.

The principle of the evaluation process is to realise the control of the employee competences. For that purpose, the evaluator can choose among the trade, the watch-group and the evaluation type (that he has the access according to his rights) and he follows a step set to realise an evaluation. So just for the preparation phase of the evaluation an evaluator that have the access to two trades, three watch-group and two evaluation type can realise twelve different evaluations. Depending of the trades and watch-groups the evaluation is carried out differently with new actions to be produced. Moreover other particularities is applied during the evaluation process increasing the number of possible behaviour. So the test of the evaluation is complex by the number of possible behaviours due to a large combination of data.

2) Modelling the test process: In order to test the evaluation process, we have created a set of processes by applying the concept discussed previously. We have created three main processes: that we call process A, B and C. In order to present the common points existing between each of these 3 main processes we have represented in Figure 16 an overview of the way each of the main processes share common and independent sub-processes. In blue appears the common sub-processes, in green the independent sub-processes linked to the main process A, in orange those for the main process B and in purple those for the main process C.

These processes describe the evaluation process with changes for each test objectives. Each main process contains a set of five common sub-processes (in blue in Figure 16) and additional sub-processes to test specific behaviours. Each of the main processes A and B contains one different additional sub-process ("Quote a key point case A" in green for the main process A and "Quote a key point case C" in purple for the main process C).The last main process (B) contains five additional sub-processes in orange. Moreover, some sub-process contains common sub-processes in order to facilitate the maintenance and the creation of new main process. We are studying several ways to model in order to find the most effective method to model, to create new processes and to maintain. Most of sub-processes contain only one task and are dedicated to a specific functionality of the application. Indeed, whenever a behaviour (for example the connection) is used more than once for the test, we create a new sub-process and we call this new sub-process where it is needed. The creation of a sub-process of one task takes less than five minutes to be done and the same for making a change. The time passed on the modelling phase represent 30% of the automation process.

3) Automatically produce test cases: The main process A to test the evaluation in the most basic way generates a single test case of 10 steps. It does not use any particular data, hence the production of a single test case.

The main process B generates two scenarios of 55 and 61 steps. That's because two complex cases are verified: on the one hand, the overall progress and, on the other hand a synthesis.

To finish the main process C generates six scenarios on average of 19 steps each. This can be explained by the fact that a combinatorial approach to ratings and watch groups is used here. In case C we perform actions quasi similar to case A, except we produce 6 test cases in contrast to case A where only one test case is produced. Here it is the use of combinatorics that influences the number of test cases produced. Hence the importance of good data management in order not to increase the number of product test cases.

All of these nine scenarios are converted into scripts.

4) Complete adaptation layer: In order to complete the adaptation layer, we have created a Keywords dictionary containing 30 keywords. These keywords have a fine granularity and allow to cover all the 33 test actions of the scenarios. For the data mapping we use in most cases the data extracted from the scenarios and we adapt the keywords code according to the data extracted. It allows functional testers to keep an high abstraction level in the scenarios while allowing the script generation. The coding of the keywords (1400 lines of code) and the construction of the objects repository represent 60% of the time of the automation process. This percentage and the

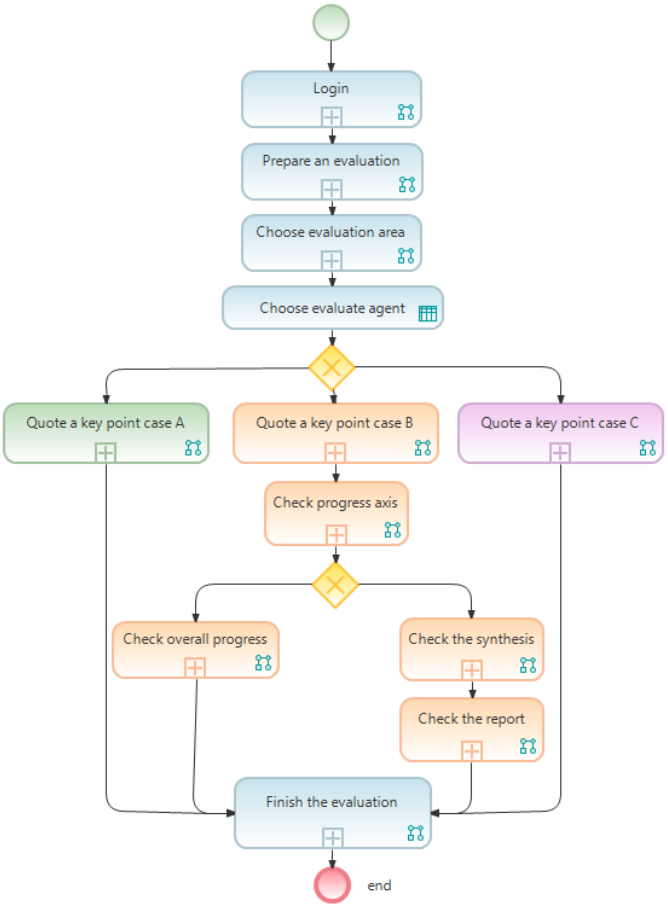

Fig. 16. Main processes content

number of lines of code are due to the difficulty to create a stable Selenium code and the large possibility of behaviours of the application.

5) Automatically produce test scripts: Finally, once the adaptation layer is completed, we success to generate 9 scripts, using the 30 keywords and generating 400 lines of code. These scripts are executed in an execution environment by using for some script only the data issued of the scenarios and for the other a set of data extracted of the database. For example, the script allowing to realise a “simple” evaluation is executed in a loop with data changes in each loop.

B. 2nd experiment

1) Description of the SUT: The second project was carried out in the field of insurance and deals with the marketing of a new offer, in the context of the sale of guarantees on various products. This results in the launch of a new web application allowing the subscription to a product. This subscription gives rise to a precise business path through a series of steps. The data manipulated are mainly numerical amounts linked to rules for managing sales turnover, calculating franchises, etc. The key point in the tests was to be able to manage the different types of data mentioned above: design, implementation and keywords.

2) Modelling the test process: During our experimentation we operated on a limited perimeter because we wanted to verify the approach in a context of short cycle (sprint of 10 days). As a result, we sought to cover a selected set of 28 business requirements in this 10-day periode. In order to cover these 28 requirements, we carried out a total of 29 processes. Each of them allows us to cover in detail each of these requirements. With this segmentation, it was easy to build regression business test cases and to cover multiple functionalities. In addition, the visual representation of the requirements provided a better understanding of the business requirements. Indeed, each requirement being represented by a visual sequence of simple tasks, such as ”Enter a turnover”, then ”Control the franchise amount”, it was simple to represent all the steps when signing a contract. During this experiment some business aspects were forgotten in the modeling of the test process. For example, elements modeled on day 1 were reviewed on day 5. In such cases it was easy to find the elements concerned in the model and thus quickly updated. As a result of our experimentation we were able to cover 100% of the requirements through the models.

3) Automatically produce test cases: In order to cover all requirements, 37 test cases were generated by the tool. This number makes it possible to cover each functionality independently and also crosswise through end-to-end testing. The tool allows the generation of ”the minimum” number of test cases in order to promote the maximum coverage of the requirements. This is a significant advantage to facilitate automation by limiting the number of test cases to be automated and later to be maintained.

4) Complete adaptation layer: In order to complete the adaptation layer, as for the first experiment we have created a dictionary of keywords. These keywords have a variable granularity in order to keep the scripts readable. We have therefore created functions to complete fields as well as to complete interfaces. In order to manage our data during the completion of the adaptation layer we used two types of data: design data and implementation data. For each of the design data existing as an equivalence class we created a data set to replace it by the real value it takes in the SUT, i.e. the value of the implementation data. Thus the scripts produced will be executable. In our study we produced 6 keywords in order to test our approach and to verify if a limited number of keywords could be used to verify a set of system properties. Here we have been able to cover all regression tests, i.e. 5 test cases verifying a set of properties.

5) Automatically produce test scripts: Similar to the 1st experiment, the generation of scripts occurs once the adaptation layer is completed. In the end, 5 test scripts were produced, covering our objectives. We were able in a given short time to build the proposed approach from scratch, proving that it can be adapted to short cycle and iterations.

VII. Conclusion and Future Works

The establishment of the automation process on a large IT project spotlights a set of observations. First, the use of

the lightweight MBT approach helps to design test cases and allows to keep a good vision of the tested business process. Indeed, by keeping a model with a high abstraction level, it is possible to discuss around the model and directly to extend test objectives by changing the model. Moreover, the creation of a set of sub-process regrouping the common behaviour of the application helps to realize rapidly new main processes and reduce the maintenance of each process. We observed a design time reduced by 30% and an update time divided by 2. Although at first sight the use of this approach seems to speed up the implementation of automation and reduce maintenance time, we have not been able to measure the exact gain between a traditional approach and the one we propose. Our future work could be to carry out in parallel two experiments on the same project but applying two different approaches and thus measure the results obtained. However the proposed approach provides an overall process for creating and maintaining automated tests. But the implementation of keywords must respect the good practices of structured keyword coding [10].

Some good practice has to be established to facilitate the work between the functional tester and the automation engineer. Notably for data management and an efficient reusability of the keywords and its parameters. Indeed, if the keywords wait for a specific data and data is different script execution will fail. So, it's important to create keywords able to adapt to minor change of data values (like upper case instead of lower case for example). Nevertheless, a good practice is to use existing data in the dictionary (in the case of the use of Yest) when it is possible, in order to guarantee the production of test cases easy to automate.

REFERENCES

[1] A. Arcuri, "An experience report on applying software testing academic results in industry: we need usable automated test generation," *Empirical Software Engineering*, vol. 23, no. 4, pp. 1959–1981, Aug 2018. [Online]. Available: https://doi.org/10.1007/s10664-017-9570-9

[2] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Software Testing, Verification and Reliability*, vol. 22, no. 5, pp. 297–312, 2012. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1002/stvr.456/full

[3] A. C. Paiva, N. H. Flores, J. P. Faria, and J. M. Marques, "End-to-end automatic business process validation," *Procedia Computer Science*, vol. 130, pp. 999 – 1004, 2018, the 9th International Conference on Ambient Systems, Networks and Technologies (ANT 2018) / The 8th International Conference on Sustainable Energy Information Technology (SEIT-2018) / Affiliated Workshops. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050918304666

[4] P. Yotyawilai and T. Suwannasart, "Design of a tool for generating test cases from bpmn," in *2014 International Conference on Data and Software Engineering (ICODSE)*, Nov 2014, pp. 1–6.

[5] Z. H. He, X. Zhang, and X. Y. Zhu, "Design and implementation of automation testing framework based on keyword driven," in *Advanced Manufacturing and Information Engineering, Intelligent Instrumentation and Industry Development*, ser. Applied Mechanics and Materials, vol. 602. Trans Tech Publications, 10 2014, pp. 2142–2146.

[6] D. Garg, A. Singhal, and A. Bansal, "A framework for testing web applications using action word based testing," in *2015 1st International Conference on Next Generation Computing Technologies (NGCT)*. IEEE, 2015, pp. 593–598.

[7] R. Hametner, D. Winkler, and A. Zoitl, "Agile testing concepts based on keyword-driven testing for industrial automation systems," in *IECON 2012 - 38th Annual Conference on IEEE Industrial Electronics Society*, Oct 2012, pp. 3727–3732.

[8] T. Takala, M. Maunumaa, and M. Katara, "An adapter framework for keyword-driven testing," in *2009 Ninth International Conference on Quality Software*. IEEE, 2009, pp. 201–210.

[9] B. Elodie, A. Fabrice, L. Bruno, and B. Arnaud, "Lightweight model-based testing for enterprise it," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, April 2018, pp. 224–230.

[10] R. Rwemalika, M. Kintis, M. Papadakis, Y. L. Traon, and P. Lorrach, "On the evolution of keyword-driven test suites," *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pp. 335–345, 2019.