# Toward testing multiple User Interface versions

Nelson Mariano Leite Neto, Julien Lenormand, Lydie du Bousquet, Sophie Dupuy-Chessa
Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France
CNRS, LIG, F-38000 Grenoble, France
{lydie.du-bousquet, sophie.dupuy}@imag.fr

Abstract:
More and more software systems are susceptible to be used in different contexts. Specific user interfaces are thus developed to take into account the execution platform, the environment and the user. The multiplication of user interfaces increases the testing task, although the core application remains the same. In this article, we explore a solution to automate testing in presence of multiple user interfaces designed for the same application (e.g. web-based, mobile, …). It consists of expressing abstract test scenarios in a high-level language, and then to apply concretization rules specific to each UI version to generate executable tests.

# 1. Introduction

With the rise of mobile devices such as notebooks, smartphones and tablets, software systems are susceptible to be used everywhere and in different contexts. A context of use involves three factors: the platform (i.e. the type of device), the environment (e.g. the level of brightness) and the user (e.g. with different levels of expertise) [1]. These factors affect the interaction between the user and the system. That is why different User Interfaces (UI) can be proposed in order to fit the user's needs. Systems with adaptable UIs are built to dynamically propose the relevant UI with respect to their interpretation of the contextual situation. In these conditions, developing a set of relevant UIs can become as complex as developing the core of the system [2].

This article is concerned with the problem of asserting quality while developing different UIs in parallel for the same system (e.g., in order to propose adaptable UIs). Quality is achieved during the development and often evaluated by testing [2, 3, 4]. Testing becomes more and more expensive and automation can be a key to reduce this cost. Creation and maintenance of test scripts have to be taken into account to make test automation cost-effective [4].

To be cost effective, our proposition is to factorize the testing process as much as possible. Our starting point is a set of different UIs for the same system. We aim at automatically generating executable test scripts for each UI from a ***single*** description. To do that, we express test scenarios in an abstract way, and concretizing them for each UI version using specific translation rules. We think that this approach has several advantages. An abstract test scenario is easier to write and maintain than different executable test scripts. It is also easier to make evolve a set of test scripts

This paper is structured as follows. First, the related work is presented. Then, we introduce an illustrative example, using three web-based UIs and one mobile UI. Next, we detail the approach. The last section concludes and draws some perspectives.

# 2. Related Work

Our work concerns the problem of automating validation of multiple UI for the same application that are developed to fit different contexts of use. Executable test scripts are specific to each UI since they have to match the widget and navigation specificities of each version. Our solution aims at factorizing the effort of testing required for the different versions. In this section, we explore some approaches that have been proposed to automate UI testing with factorization point of view.

UI test scripts can be manually written and then *automatically executed* in some testing tools such as Abbot tool[1] or Selenium WebDriver[2]. The oracle is implemented as assertions in the code of the scripts. The automation relies mainly on the execution part.
Writing scripts is a laborious task. To ease it, “capture and replay” tools can be used to *record* user’s interactions with the UI. The recorded interactions can then be replayed. Many tools propose this feature, both for web-based or mobile application testing [5]. Oracle can rely on visual inspection during re-execution of the captured scenarios, image comparison or manual added assertions [6, 7].
Direct scripting and capture and replay approaches provide no direct factorization possibilities for test generation. Each interface has to be analyzed separately.

Model-based approaches have also been proposed to *automate test generation* for UIs [8, 9, 10, 11, 12, 13, 14, 15, 16]. They offer more

[1] http://abbot.sourceforge.net/
[2] http://www.seleniumhq.org/

possibilities of factorization, especially when the model is built manually during the development process. But such a construction is quite difficult to carry out since it may require a high-level expertise [13]. Moreover, it is often difficult to maintain the equivalence of a model and the implementation during the application evolutions.
To deal with this problem, different authors propose *to extract automatically* the model from the existing interfaces [8, 9, 10, 12]. This type of approach is less adapted to our needs, since each interface has to be analyzed separately. However, being able to extract a specific model from each interface can then allow checking automatically the equivalence of interfaces [17].
When tests are generated from an abstract model, mapping from the model to the code has to be expressed in order to produce executable tests [14]. In [11], authors use a keyword machine to transform abstract test cases into executable ones.

No related work directly addresses the problem of generating executable test scripts for each UI version from a single description. However the idea of transforming abstract test cases into executable ones can be of interest to factorize. In this article, our high-level scenarios that are common to all interfaces were produced manually. But a model-based approach to generate them should be possible if a model is available. The next section describes small example of application.

# 3. Illustrative example

The illustrative example used in this work is a prototype for a smart home energy management system. It allows users to control and monitor the energy consumption in a home from different devices.

We have developed four UIs for this system: a mobile application for Android (named “mobile”), a web interface for desktop browser (named “web0”), and two web interfaces for mobile browsers (one with a menu page named “web1”, the other with a menu bar, named “web2”). The web-based versions are implemented in HTML5, JavaScript and JQuery. The mobile version is developed in Java 1.8. All versions have the same features. The differences between the different web versions are of two sorts. First widgets are different. Second navigation among pages is different.

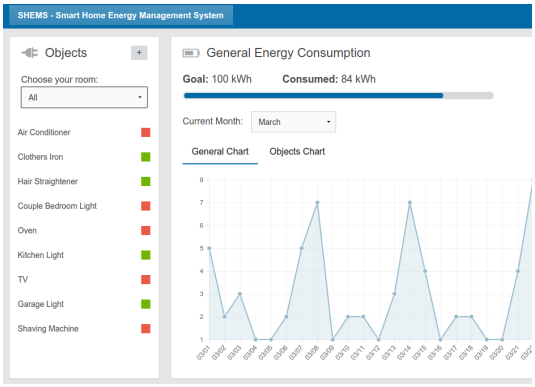


**Figure 1: goal and filter features on the web0 interface**

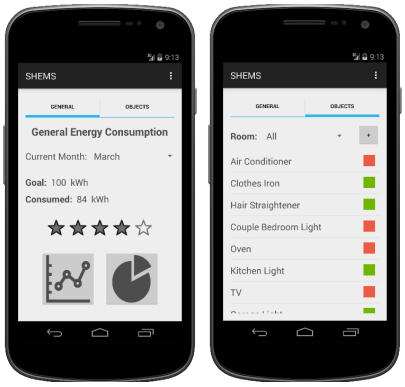


**Figure 2: goal and filter features on the mobile interface**

In this paper, we focus on three features.

- Goal: the user can check information about the general energy consumption per month. He can access the goal and actual consumption in kWh for the chosen month (Fig. 1 and 2).
- Filter: the user has access to a list of all the objects in the house, having the possibility to filter them per room. An object means any component that can be controlled and whose energy consumption can be registered, such as lights and electronic devices (Fig. 1 and 2).
- Comparator: the user can choose two objects and compare, side by side, their energy consumption charts (Fig. 3).

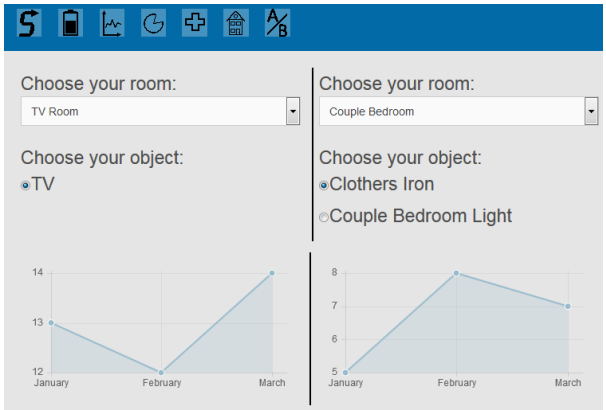


**Figure 3: compare feature on the web2 interface**

Test scripts for web versions are executed in Selenium [20], a tool following a capture and playback approach for web-based application. The Selendroid framework completes the Selenium environment for the mobile version[3].

[3] http://selendroid.io/

# 4. Approach

## 4.1 Principles

As said previously, the four UI versions share the same features but have different widgets and navigation paths. For this reason, to test them, it is necessary to write four specific executable test script sets. To avoid this tedious work, we propose the approach illustrated Fig. 4.

Abstract test “scenarios” are expressed in a high-level language. These scenarios are common to all the UI versions. A scenario is composed of a sequence of abstract instructions. A set of translation rules is used to transform the abstract test scenario into executable test scripts. The rules explicitly associate executable code to abstract descriptions. The translation rules are specific to each interface and defined manually. A translation rule simply rewrite an abstract instruction into an executable one, taking into account the implementation specificities. Translation rules can be the same for different UIs if they share the same widgets. A tool is used to translate the abstract scenario into executable test scripts.

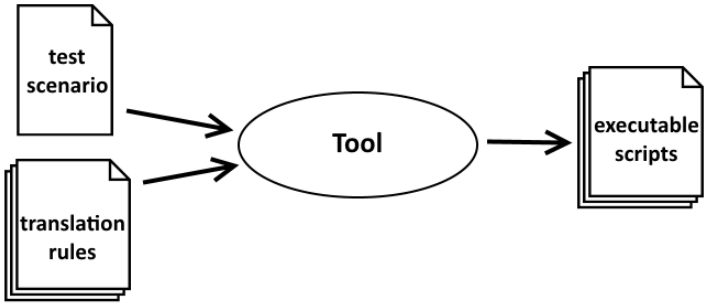


**Figure 4: our approach to test script generation**

## 4.2 Principles put into Practice

To express the high level scenarios, we use an existing language called TSLT (Test Schema Language for Tobias [18]). It is a textual language that contains several types of constructs allowing the definition of complex system scenarios. It is used as input of a testing tool called Tobias [18], which is responsible of the translation into the executable scripts.

Tobias is a test generator based on combinatorial testing. Combinatorial testing performs combinations of selected input parameter values for given operations and given states. Tobias adapts this principle to the generation of operation call sequences. It allows exploring system-

atically a large set of behavior sequences from a single abstract description, called scenario. An on-line version of Tobias is available at http://tobias.liglab.fr/

In our use of Tobias for UI testing, we start by expressing the abstract scenarios. Then we identify the executable code corresponding to each abstract instruction. This correspondance is expressed by a translation rule in TSLT. For the moment, this is carried out manually.

For instance, Listing 1 shows an abstract scenario in TSLT, designed to check that the displayed values are the expected ones for each goal. It consists of a sequence of three abstract instructions that allows to (1) navigate to the appropriate view (@goToGoal), (2) choose a month (@selectMonth) and (3) check that the displayed value is the expected one (@verifyValues).
Instruction “Integer month = [1-3]” indicates to Tobias to repeat the sequence for the first three months (combinatorial approach). Listings 2 to 5 show the four specific translations rules for “@goToGoal” abstract instruction.

```
group testMonthValue[us=true] {
   Integer month = [1-3];
   @goToGoal;
   @selectMonth;
   @verifyValues;
}
```

**Listing 1: Abstract scenario of the Goal test case**

```
group goToGoal[us=false] {
   // does nothing
}
```

**Listing 2: Translation rule of @goToGoal for Mobile**

```
group goToGoal[us=false] {
   driver.get(siteAddress);
}
```

**Listing 3: Translation rule of @goToGoal for Web0**

```
group goToGoal[us=false] {
   driver.get(siteAddress);
   WebElement goalButton =
driver.findElement(By.xpath("/html/body/div/section/ul/li[1]/div/a"));
   goalButton.click();
}
```

**Listing 4: Translation rule of @goToGoal for Web1**

```
group goToGoal[us=false] {
   WebElement goalButton = driver.findElement(By.id("menu_goal"));
   goalButton.click();
}
```

**Listing 5: Translation rule of @goToGoal for Web2**

From these TSLT rules, Tobias is able to translate the abstract scenarios into executable scripts. The executable scripts are executed in JUnit with Selenium or Selendroid frameworks.

For testing the three features of our illustrative example, six scenarios were designed, using 11 abstract instructions. Scenarios were translated into 21 executable test scripts in JUnit. The difference between the number of abstract scenarios and the number of executable tests is due to the combinatorial nature of Tobias. For example, the test case shown in Listing 1 is translated into three JUnit tests, each one corresponding to a different value for the month (1, 2, 3). By only changing "Integer month = [1-3]" into "Integer month = [1-12]" it is possible to generate the 12 test cases necessary to check all the months. It can also be changed into "Integer month = [0-13];" and then generate robustness tests.

As said previously, it is important to have test suites easy to maintain. The size of the description is one factor that impacts the cost of maintenance. Table 1 shows the number of lines written for the abstract scenarios with the translation rules for each feature. It also displays the number of line of code for the executable test scripts. Without the approach, those executable test scripts should have been written by hand. It can be observed that the number of lines to write has been at least halved.
This diminution of code between the abstract scenario and the executable test case is also due to the fact that the JUnit syntax is not described in the abstract scenarios nor in the translation rules. Tobias tool automatically generate the JUnit packaging.

| Feature tested | Implementation | Mobile | Web0 | Web1 | Web2 |
|---|---|---|---|---|---|
| Goal | Test scripts | 161 | 143 | 151 | 147 |
| | Abstract scenario | 76 | 67 | 69 | 68 |
| Filter | Test scripts | 529 | 350 | 402 | 369 |
| | Abstract scenario | 117 | 88 | 93 | 92 |
| Compare | Test scripts | 231 | 235 | 235 | 227 |

| | | | | | |
|---|---|---|---|---|---|
| | Abstract scenario | 65 | 60 | 59 | 58 |

**Table 1: Number of lines for the abstract scenarios and generated tests**

| Feature | Rule | Mobile | Web0 | Web1 | Web2 |
|---|---|---|---|---|---|
| Goal | @goToGoal | 0 | 0 | 3 | 2 |
| | @selectMonth | 5 | 3 | 3 | 3 |
| | @verifyValues | 4 | 4 | 4 | 4 |
| | @veryfyMonthsCount | 6 | 3 | 3 | 3 |
| Filter | @goToObjects | 2 | 0 | 3 | 2 |
| | @selectRoom | 3 | 5 | 5 | 5 |
| | @verifyObjectsFiltered | 25 | 5 | 5 | 5 |
| | @selectRoomUncorrect | 3 | 3 | 3 | 3 |
| Compare | @goToCompare | 2 | 3 | 3 | 2 |
| | @selectRoom | 3 | 3 | 3 | 3 |
| | @verifyWidget | 10 | 10 | 10 | 10 |
| | @verifyChart | 0 | 0 | 0 | 0 |
| | Total | 63 | 39 | 45 | 42 |

**Table 2 : Number of lines for each rule for each version of each feature**

Translation rules are quite simple. They consist in associating executable code to abstract instruction. For our example, the executable code corresponds to 0 up to 25 lines of code, for a total of 189 lines of code (see Table 2). Variation implementation details are thus expressed in a very concise way and localized. It becomes easy to make them evolve.

## 4.3 Discussion and Analysis

Our focus is to show the feasibility to express test scenarios for multiple UI versions of the same application, and to measure the effect of the factorization. The factorization effect can be evaluated through the difference of size between abstract and executable tests (Table 1). The factorization contribution is clearly visible. With Tobias, it is easy to increase artificially this difference, by playing on the combinatorial feature of the tool. But we deliberately limit the combinatorial exploration (e.g. we check only three months, instead of the twelve).

TSLT language does not allow expressing directly loops, return statements, exception handling nor proper functions in the translation rules. This constraint has for origin to guaranty that the combinatorial engine of Tobias will always succeed in the process of translating abstract scenarios into executable tests. Here, those constructions are necessary to express oracle condition and for scrolling handling navigation on the mobile version. The limitation has been bypassed during the experiment by separating code of the loops in a different file. To be

able to express all the translation rules in TSLT, the language has to be extended. This does not affect the relevance of the approach.

As it can be seen on Table 2, some translation rules correspond to zero line of code. The reason is that there is no corresponding instruction within Selenium/Selendroid (e.g. it is not possible to check that an image is the one which is expected). This is directly linked to the testing framework expression power. It is independent of the approach.

# 5. Conclusions and Perspectives

We are concerned by the validation of several UIs provided for the same application for different contexts. Our motivation is to prepare the validation of adaptive applications, where tests have to be chosen with respect to the context. To do that, we would like to be able to automate test in a cost effective way.

The work described here is a first step toward this goal and should be considered as a feasibility study. Abstract scenarios and translation rules were both expressed in an existing language called TSLT, associated to a testing tool called Tobias. It is a combinatorial tool which aims at unfolding scenarios to explore all combinations that are defined by the scenario. It was not originally designed for UI testing but for JUnit test script generation. That was the main reason why it was chosen. The fact that the translation of abstract scenario into executable scripts can be done in a combinatorial way is an advantage since it helps in the process of factorizing code (and thus being cost effective).

Even if our illustrative example is simple, the different versions were built to explore a variety of widgets. Different navigation paths were also considered. This helps us to be confident in the fact that the factorization can be generalized. Moreover, translating abstract scenarios into executable ones can also be carried out for other testing framework than JUnit, since Tobias is designed to fit other testing frameworks. However, the example shows that the TSLT language is not fully appropriate as it is designed now (see. Sect. 4)

The example also shows that the approach can decrease the work of creating and maintaining testing suites. The size of the abstract scenarios with the translating rules is much smaller than the size of the final test scripts, which share a lot of identical code. Writing them is not simpler, but definitively shorter.

Our perspectives are to consolidate the work by exploring larger examples, other versions of interfaces and testing frameworks. This will help us to evaluate more precisely the amount of manual effort required with respect to the automated one and the approach genericity. Once this step is achieved, we will explore the possibility to associate translation rules to a context definition, and then to provide a framework that is able to generate tests during the execution, to fit the current execution context. The final step will be to generate automatically the abstract tests from a model, like those proposed in the Cameleon framework [19] and/or to produce automatically the translation rules such as in TESTAR [10].

# Bibliography

[1] Coutaz, J., and Calvary, G. HCI and software engineering: Designing for user interface plasticity. Human-Computer Interaction: Development Process (2009).

[2] Muccini, H., Francesco, A. D., and Esposito, P. Software testing of mobile applications: Challenges and future research directions. In 7th Int. Workshop on Automation of Soft. Test (AST), IEEE (2012), 29-35.

[3] Beizer, B. Software testing techniques. Dreamtech Press, (2003)

[4] Grechanik, M., Xie, Q., and Fu, C. Maintaining and evolving GUI-directed test scripts. In IEEE 31st Int. Conf. on Software Engineering (ICSE) (2009), 408-418.

[5] Gao, J., Bai X., Tsai W. T., and Uehara T. Mobile application testing: a tutorial. IEEE Computer, 47:2 (2014), 26-35.

[6] Jung, H., Lee, S., and Baik, D.-K. An Image Comparing-based GUI Software Testing Automation System. In SERP (2012), 318-322.

[7] Xie, Q., and Memon, A. M. Designing and comparing automated test oracles for GUI-based software applications. ACM Transactions on Software Engineering and Methodology (TOSEM) 16, 1 (2007).

[8] Aho, P., Suarez, M., Kanstrén, T., and Memon, A. M. Industrial adoption of automatically extracted GUI models for testing. In EESS-MOD@ MoDELS (2013), 49–54.

[9] Aho, P., Suarez, M., Kanstren, T., and Memon, A. M. Murphy tools: Utilizing extracted GUI models for industrial software testing. In IEEE 7th Int. Conf. on Software Testing, Verification and Validation Workshops (ICSTW), (2014), 343–348.

[10] Vos, T. E., Kruse, P. M., Condori-Fernández, N., Bauersfeld, S., and Wegener, J. TESTAR: Tool support for test automation at the user interface level. Int. Journal of Information System Modeling and Design (IJISMD) 6, 3 (2015), 46–83.

[11] Nieminen, A, Jääskeläinen, A., Virtanen, H., Katara, M. A Comparison of Test Generation Algorithms for Testing Application Interactions, 11th Int. Conf. on Quality Software (QSIC), (2011), 131-140.
[12] Amalfitano, D., Fasolino, A.R., Tramontana, P. A GUI Crawling-Based Technique for Android Mobile Application Testing. IEEE 4th Int. Conf. on Software Testing, Verification and Validation Workshops (ICSTW), (2011), 252-261.
[13] Holzmann, G. J., and Smith, M. H. An automated verification method for distributed systems software based on model extraction. IEEE Trans. on Software Engineering (TSE), 28, 4 (2002), 364-377.
[14] Grilo, A., Paiva, A., and Faria, J. Reverse engineering of GUI models for testing. In 5th Iberian Conf. on Information Systems and Technologies (CISTI), (2010), 1-6.
[15] Nguyen, B., Robbins, B., Banerjee, I., and Memon, A. Guitar: an innovative tool for automated testing of GUI-driven software. Automated Software Engineering (ASE) 21, 1 (2014), 65-105.
[16] Yuan, X., Cohen, M. B., and Memon, A. M. Towards dynamic adaptive automated test generation for graphical user interfaces. In IEEE Int. Conf. on Software Testing, Verification and Validation Workshops (ICSTW) (2009), 263–266.
[17] Oliveira, R., Dupuy-Chessa, S., and Calvary, G. Equivalence checking for comparing user interfaces. The 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, ACM, (2015).
[18] Triki, T., Ledru, Y., du Bousquet, L., Dadeau, F., and Botella, J. Model-based filtering of combinatorial test suites. In Fundamental Approaches to Software Engineering (FASE). Springer, (2012), 439-454.
[19] Calvary, G., Coutaz J., Thevenin, D., Limbourg, Q., Bouillon, L., Vanderdonckt, J. A Unifying Reference Framework for Multi-Target User Interfaces, Interacting With Computers, Vol. 15/3, (2003), 289-308