Check for updates

Ensuring the Consistency Between User Requirements and Graphical User Interfaces: A Behavior-Based Automated Approach

Thiago Rocha Silva[1(✉)], Marco Winckler[2], and Hallvard Trætteberg[1]

[1] Department of Computer Science, Norwegian University of Science and Technology (NTNU), Trondheim, Norway
{thiago.silva,hal}@ntnu.no
[2] SPARKS-I3S, Université Nice Sophia Antipolis (Polytech), Sophia Antipolis, France
winckler@unice.fr

Abstract. Ensuring the consistency between Graphical User Interfaces (GUIs) and user requirements is a critical aspect of the design process since it is through the GUIs that users perceive the system and experience the available features in order to achieve their goals. This paper presents an approach based on Behavior-Driven Development (BDD) which employs an ontology in order to provide automated assessment for web GUIs. The approach has been evaluated by exploiting user requirements described by a group of experts in the flight tickets e-commerce domain. Such requirements gave rise to a set of User Stories that have been used to automatically assess the GUIs of an existing web system for booking business trips. The results have shown our approach was able to identify different types of inconsistencies in the set of GUIs analyzed, allowing to build an effective correspondence between user requirements and their representation on the GUI.

Keywords: Behavior-Driven Development (BDD) · User Stories · Graphical User Interfaces · User Requirements Assessment

1 Introduction

In most of existing computing systems, Graphical User Interfaces (GUIs) are the part of the system that mediates the interaction between the user and the system's core functions. A GUI can be decomposed in two parts: (*i*) the presentation, that describes how graphical elements are organized to visually inform the functions/services available; and (*ii*) the interactive behavior, that processes user's actions transforming them into requests to core functions, and ultimately providing feedback about the current state of the system. In the users' point of view, if some feature/core function is not available through the GUI, this feature does not exist at all. Being the main bridge between the system and the end user, GUIs are a crucial artifact to be assessed against user requirements. As Hellmann [1] pointed out, the simplest way to perform GUI testing is with manual testing, wherein a human tester interacts with an application to verify that its responses are correct. A human

© Springer Nature Switzerland AG 2019
S. Misra et al. (Eds.): ICCSA 2019, LNCS 11619, pp. 616–632, 2019.
https://doi.org/10.1007/978-3-030-24289-3_46

tester can easily interact with an application and recognize when an error occurs, but manual testing is very slow and error-prone. If testing should be done frequently, then manually testing a GUI quickly becomes unfeasible.

Automated tools exist to simplify and automate this process. Most GUI testing tools work on the capture/replay paradigm [2–5]. In this paradigm, testing tools monitor the set of interactions between a human tester and the system and record these steps so that they can be replayed later as automated tests. However, capture/replay tools (CRTs) do not tend to record tests in a human-readable manner, meaning that it is much more difficult to modify an existing test than to record a new one [6]. Other tools are designed to make direct calls to the system using the native support for automation of each user interface environment. When testing user interfaces presented by means of a web browser, for example, how these direct calls are made, and the features they support depends on the target browser. Such approaches tend to be much more flexible to implement automated testing for GUIs, and tests specified by them tend to be easier to maintain, but they carry the same problem of low human-readability.

The low human-readability of tests is part of a wide problem related to the communication gap between domain experts and developers, which is a long-time issue in software engineering and one of the main reasons for project failure. Misunderstandings about the user requirements and their design on user interfaces are a frequent source of modeling issues and final products which not meet the expected system behavior. Motivated by such a gap, some approaches and tools have emerged to raise the abstraction level of requirements and test specifications. Behavior-Driven Development (BDD) [7] is one of these approaches which has stood out in the software engineering community as an effective means to provide automated acceptance testing by specifying natural language user requirements and their tests in a single textual artifact. BDD benefits from a requirements specification based on User Stories [8] which are easily understandable for both technical and non-technical stakeholders.

A common drawback of these approaches is that the set of behaviors specified for requirements and tests are not directly executable, i.e., a developer will be required to implement the code for such behaviors in order to run them on GUIs. This motivated us to investigate the use of a predefined set of interactive behaviors on GUIs which could be implemented once and then reused to allow automated tests without the intervention of a developer. This paper presents an approach based on BDD and User Stories to support the specification and the automated assessment of user requirements on web GUIs of interactive systems. The set of user-system interactive behaviors is provided by means of an ontology [9, 10]. The following sections present the foundations of this work, the proposed approach with its technical implementation, and the results we got when using the approach to assess the GUIs of a web system to book business trips.

2 Foundations

2.1 Behavior-Driven Development

Behavior-Driven Development (BDD) is a specialization of Test-Driven Development (TDD) [11], and is intended to make the practice of writing automated testing more accessible and intuitive to newcomers and experts alike. It shifts the vocabulary from

being test-based to behavior-based. It positions itself as a development paradigm, emphasizing communication and automation as equal goals. In BDD, the behaviors represent both the requirements specification and the test cases.

BDD drives development teams to a requirements specification based on User Stories in an understandable natural language format. This format allows specifying executable requirements by means of a Domain-Specific Language (DSL) provided by Gherkin[1]. Gherkin is a DSL that lets users and developers describe software behavior without detailing how that behavior is implemented. Gherkin serves two purposes: documentation and automated tests. By using this language, requirements specifications can be used to implement automated tests, conducting to living documentation and making easier for clients and other stakeholders to set their final acceptance tests.

User Stories were firstly proposed by Cohn [8] and provide in the same artifact a narrative, briefly describing a feature in the business point of view, and a set of scenarios to give details about business rules and to be used as acceptance criteria, giving concrete examples about what should be tested to consider a given feature as done. Cohn and North [8, 12] propose a useful template for that:

```
Title (one line describing the story)
Narrative: As a [role], I want [feature], So that [benefit]
Scenario 1: Title
Given [context], When [event], Then [outcome]
```

This structure is largely used in Behavior-Driven Development (BDD) and has been named by North [12] as "BDD story". According to this template, a User Story is described with a *title*, a *narrative* and a set of *scenarios* representing acceptance criteria. The title provides a general description of the story, referring to a feature this story represents. The *narrative* describes the referred feature in terms of the role that will benefit from the feature, the feature itself, and the benefit it will bring to the business. The acceptance criteria are defined through a set of *scenarios*, each one with a *title* and three main clauses: "*Given*" to provide the context in which the scenario will be actioned, "*When*" to describe events that will trigger the scenario and "*Then*" to present outcomes that might be checked to verify the proper behavior of the system. Each one of these clauses can include an "*And*" statement to provide multiple contexts, events and/or outcomes. Each statement in this representation is called a *step*.

2.2 Ontological Support for GUI Automated Testing

Our approach is based on previous works which explore an ontology to describe common behaviors with a standard vocabulary for writing User Stories [9, 10]. The main benefit of this strategy is that User Stories using this common vocabulary can support specification and execution of automated test scenarios on GUIs. The ontology covers concepts related to presentation and behavior of interactive components used in web and mobile applications. It also models concepts describing the structure of User Stories, tasks, scenarios, and GUIs.

[1] https://cucumber.io/docs/gherkin/.

The dialog part of a GUI, as illustrated by Fig. 1, is described using the ontology by means of concepts borrowed from abstract state machines. The User Story *scenario* meant to be run in a given GUI is represented as a *transition*. *States* are used to represent the original and resulting GUIs after a transition occur (states A and B in Fig. 1). Scenarios in the transition state always have at least one or more *conditions* (represented in scenarios by the "*Given*" clause), one or more *events* (represented in scenarios by the "*When*" clause), and one or more *actions* (represented in scenarios by the "*Then*" clause). The presentation part of a GUI is described in the ontology through *interaction elements* which represent an abstraction of the different widgets commonly used in web and mobile user interfaces.

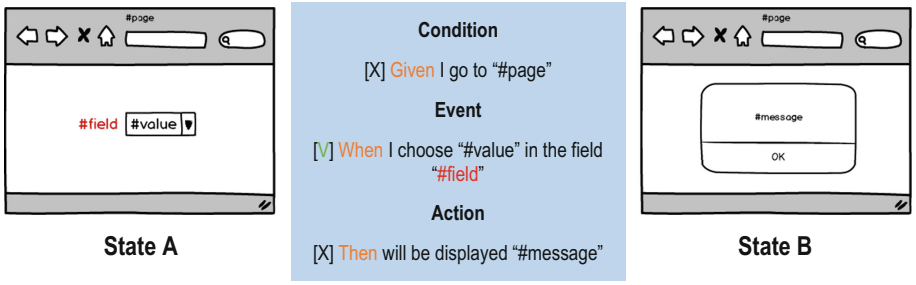


Fig. 1. Representation of a User Story scenario using the state machine concepts.

The common behaviors in the ontology describe textually how users could interact with the system whilst manipulating graphical elements of the user interface. An example of behavior specification is illustrated by Fig. 2. The specification of behaviors encompasses when the interaction can be performed (using "*Given*", "*When*" and/or "*Then*" clauses), and which graphical elements (i.e. *CheckBoxes*, *TextFields*, *Buttons*, etc.) can be affected. Altogether, behaviors and interaction elements are used to implement the test of expected system behavior. In the example of Fig. 2, the behavior *"I choose '<value>' referring to '<field>'"* has two parameters: "*<value>*" and "*<field>*". The first parameter is associated to data, whilst the second parameter refers to the interaction elements supported by this behavior: *Radio Button*, *CheckBox*, *Calendar* and *Link*.

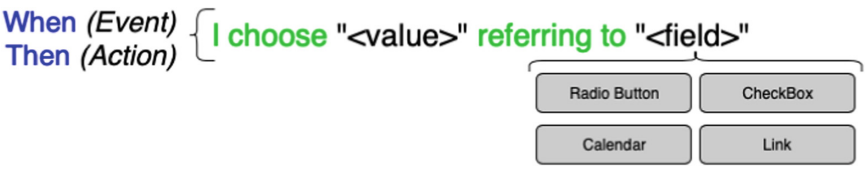


Fig. 2. Structure of a behavior as specified in the ontology.

The ontological model describes only behaviors that report steps performing actions directly on the user interface through interaction elements. This is a powerful resource because it allows keeping the ontological model domain-free, which means it is not subject to particular business characteristics in the User Stories, promoting the reuse of steps in multiple scenarios. Thus, steps can be easily reused to build different behaviors for different scenarios in different business domains.

When representing the various interaction elements that can attend a given behavior, the ontology also allows extending multiple design solutions for the UI while still keeping the consistency of the interaction. For example, even if a *Dropdown List* has been chosen to attend, for example, a behavior *setInTheField* in the first version of a prototype, an *Auto Complete* field could be chosen to attend this behavior on a next version, once both UI elements share the same ontological property for this behavior. This kind of flexibility keeps the consistency of the interaction, leaving the designer free for choosing the best solutions in a given time of the project, without modifying the behavior specified for the system. The current version of the ontology covers more than 60 interactive behaviors and almost 40 interaction elements for both web and mobile user interfaces.

3 The Proposed Approach for Automated Assessment

GUIs are fully functional versions of a user interface implemented in a given programming language for a given platform. The assessment of a GUI is made by dynamically running tests on its presentation layer with the aid of external testing frameworks. Such frameworks are able to simulate a user interacting with a GUI by running the set of scenarios described in the User Stories. Our premise for assessing such GUIs is thus the availability of an external testing framework able to run tests on a given environment. Despite the support ontology supports a specification for both web and mobile environments, so far, our approach only implements tests with Selenium WebDriver for running directly on a web browser. For assessing GUIs implemented for other environments, our approach would require integration with other testing frameworks.



Fig. 3. Flow of execution in the proposed architecture.

Besides using a framework to control navigation on a web browser, other frameworks are required to parse the text on User Stories, to build the test suit, or even to generate reports from the execution. Therefore, we have built an architecture of tools to bring together the multiple set of required frameworks for performing our testing approach on GUIs. The integrated tools architecture we propose for testing GUIs is essentially based on Demoisele Behave, JBehave, Selenium WebDriver, JUnit and Maven. We use Selenium WebDriver to run navigational behaviors, and JBehave and

Demoiselle Behave to parse the scenario script. Test results provided by the JUnit API indicate visually which tests are passed and which ones failed and why. Execution reports of User Stories, scenarios and steps can also be obtained by using the JBehave API.

Such an architecture allows users to automate testing on web user interfaces by following our behavior-based approach. The architecture has three main modules: *Core*, *Parser*, and *Runner*. The *Core* is responsible for the main interfaces of the framework by orchestrating the information among the other modules. The *Parser* is responsible for the abstraction of the component that will transform the story into Java code to send to the *Runner* through standard or project-specific sentences. The *Runner* is responsible for the abstraction of the component that will perform navigation on the user interface, such as Selenium WebDriver or even JUnit directly. The framework identifies stories written in TXT to be sent to the *Parser* module and later to the *Runner*, which is responsible for interacting with a web browser using the Selenium WebDriver. To run tests in such an architecture, story files are charged as inputs for the parser, that translates the natural language behaviors into Java methods, and then selects a runner to perform the navigational commands on a given targeted web browser. This flow of execution is illustrated in Fig. 3.

3.1 Implementation

We have structured packages and classes in different layers to implement our architectural approach. Pre-defined behaviors charged from the ontology are implemented in a class called *CommonSteps*. New extended behaviors, that are not initially covered by the ontology, can be implemented in the *MySteps* class. These new behaviors will then be parsed to Java methods as illustrated in Fig. 4. So, steps in User Stories are mapped to either *CommonSteps* or *MySteps* behaviors in order to be run.



Fig. 4. Parsing a step from a TXT file to a Java method.

The Presentation Layer includes the *MyPages* class which implements the link between abstract UI components defined in the ontology and the concrete UI components instantiated on the interface under testing. This link is crucial to allow the Selenium WebDriver and other external testing frameworks to automatically run scenarios in the right components on the UI. To link these components, the *MyPages* class identifies a screen map ("*@ScreenMap*") which address the web page location, and several element maps ("*@ElementMap*") which link the various abstract UI elements in the User Stories with their concrete UI siblings on the user interface. This link is made by manually associating the name of each abstract UI element with their concrete locators (such as IDs, XPaths, or any other web element identifier). This is illustrated by Fig. 5.

Finally, the *MyTest* class is a JUnit class in charge of triggering the tests, pointing which scenarios should be executed at a time. These three basic classes (*MySteps*, *MyPages* and *MyTest*) can also be modeled with different names into packages "*steps*", "*pages*" and "*tests*", in order to separate concerns and implements different classes for different pages or features.

```
@ScreenMap(name = "Flight Search", location = "https://www.aa.com/homePage.do?locale=en_US")
public class MyPages {

    @ElementMap(name = "Round trip", locatorType = ElementLocatorType.XPath, locator = "//*[@id='journeyTypeRT']")
    private Radio RoundTrip;

    @ElementMap(name = "One-way", locatorType = ElementLocatorType.XPath, locator = "//*[@id='journeyTypeOW']")
    private Radio OneWay;

    @ElementMap(name = "Departure Date", locatorType = ElementLocatorType.XPath, locator = "id('depDate')")
    private TextField DepartureDate;
```

Fig. 5. *MyPage* Java class.

The structure of the Java project is presented in Fig. 6. Notice that the three aforementioned classes are packed in the package "*java*" and the User Stories in the package "*resources*". On the right side of the figure, the structure of the *MyTest* class is presented highlighting the addition of the new extended behaviors in the *MySteps* class, and all the stories in the "*/stories*" folder being triggered by a JUnit test method.



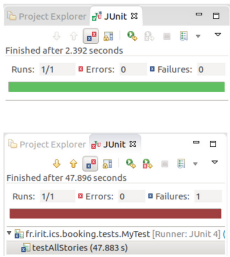
Fig. 6. Package tree (on the left) and *MyTest* class (on the right).

A resource that facilitates the written of User Stories is the immediate feedback concerning the existence of behaviors in the ontology to address the step that is being written. Figure 7 illustrates this resource. Notice that all the steps in the scenario have been recognized, i.e. there are equivalent behaviors in the ontology to address them, except the step *"When I set the date '12/20/2017' in the field 'Return'"* that has been underlined to alert that such a step is not recognized by the ontology (actually the right step in this case is generic: *"... I set '<value>' in the field '<element>'"*, like has been used in the following line). When clicking on the alert icon, a message to say that "no step is matching" will be shown. Additional feedback is also given recognizing in the step the mention to values and interactive elements when they are surrounded by quotation marks.

```
Scenario: Return Tickets Search
Given I go to "Find Flights"
When I choose "Round trip"
And I type "New York" and choose "NYC - New York, NY" in the field "From"
When I type "Los Angeles" and choose "LAX - Los Angeles International, CA" in the field "To"
And I choose the option of value "1" in the field "Number of passengers"
When I set the date "12/15/2017" in the field "Depart"
And I set "12/20/2017" in the field "Return"
When I click on "Search"
Then will be displayed "..."
```

Fig. 7. Writing a User Story and getting instant feedback of unknown steps.

The testing results are presented through the classical JUnit green/red bar within the Eclipse IDE. By the end of the tests, a JBehave detailed report is automatically generated in the project folder. Additionally, for each error found, screenshots are taken and stored to allow a better analyze of the results afterwards. Examples of these features are presented in Fig. 8.



Story Reports

Stories						Scenarios					GivenStory Scenarios					Steps					
Name	Excluded	Total	Successful	Pending	Failed	Excluded	Total	Successful	Pending	Failed	Excluded	Total	Successful	Pending	Failed	Not Performed	Ignorable	Duration (hh:mm:ss.SSS)	Vie
Access	0	2	2	0	0	0	0	0	0	0	0	9	9	0	0	0	0	00:00:00.000	stats | [html]
AfterStories	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	00:00:00.000	stats | [html]
BeforeStories	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	00:00:00.000	stats | [html]
Choice	0	1	0	0	1	0	0	0	0	0	0	21	18	0	1	2	0	00:00:00.000	stats | [html]
Confirm Flight Selection	0	2	2	2	0	0	0	0	0	0	0	26	23	2	0	1	0	00:00:00.000	stats | [html]
Dbehave	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	00:00:00.000	stats [html]
Flight	0	1	1	0	0	0	0	0	0	0	0	18	18	0	0	0	0	00:00:00.000	stats [html]
Flight Tickets Search	0	5	5	0	0	0	0	0	0	0	0	72	72	0	0	0	0	00:00:00.000	stats [html]
Search	0	1	1	0	0	0	0	0	0	0	0	17	17	0	0	0	0	00:00:00.000	stats | [html]
Select A Suitable Flight	0	1	1	0	0	0	0	0	0	0	0	26	26	0	0	0	0	00:00:00.000	stats | [html]
Test	0	2	2	2	0	0	0	0	0	0	0	17	4	4	0	9	0	00:00:00.000	stats | [html]
Voo	0	1	1	0	0	0	0	0	0	0	0	18	18	0	0	0	0	00:00:00.000	stats [html]
12	0	17	16	5	1	0	0	0	0	0	0	224	205	6	1	12	0	00:00:46.000	Totals

Fig. 8. JUnit green/red bar at the left, and JBehave detailed report at the right. (Color figure online)

4 Case Study

To evaluate our approach, we have conducted a case study with an existing web system for booking business trips. In a previous study [13], domain experts were invited to produce some User Stories to describe a feature they considered important to that system. In the present study, the User Stories produced by the participants were refined to get a representative set of user requirements to be assessed on the GUIs of the existing system. Thus, the main goal was investigating which types of inconsistencies our approach would be able to identify when assessing the GUIs.

4.1 Methodology

With the aim of simulating a software testing lifecycle, we first set up an initial version of User Stories and their respective acceptance criteria for assessment. After that, we manually located the identifiers of each interaction element on the considered GUIs to assign them to the "*MyPage*" class. Finally, we automatically ran the scenarios on the GUIs. The implemented strategy for running tests parses each step of the scenario at a time, so if an error is found out, the test stops until the error is fixed. That requires to

run several batteries of tests until having the entire scenario tested. It leads us to fix all the inconsistencies step-by-step, and consequently getting fully consistent scenarios at the end of running. However, when analyzing the reason related with each inconsistency, we can eventually conclude that the origin of the inconsistency is actually in the specification of the step in the User Story scenario, and not in the GUI itself. As a result, to fix such an inconsistency, steps of User Story scenarios may also be modified along the battery of tests to comply with a consistent specification of the user requirements. An immediate consequence of this fact is that the steps used to test a given version of a GUI can be different than that ones used to test another GUI previously. It means that regression tests are crucial to ensure that a given modification in the set of User Stories scenarios did not break some previous test on other GUIs and made some of them (that so far was consistent with the requirements) inconsistent again.

For the purpose of this study, we assumed that the released version of the existing booking system (and of course its GUIs) represents the unequivocal statement of the user requirements, once, for the purpose of this study, we cannot modify it. As such, we had not the opportunity to eventually redesign the GUI to comply with the User Stories we set up. As a consequence, all the identified inconsistencies necessarily resulted in modifications in the steps, not in the GUI.

4.2 Results

In total, we set up for assessment 3 User Stories with 15 different scenarios and tested 7 different GUIs. The sequence of GUIs (*a–f*) in Fig. 9 shows screenshots of the GUIs. The GUI *(a)* represents the interface for searching flights based on a round trip (and *(b)* based on a one-way/multidestination trip). The GUI *(c)* presents the next GUI in sequence, showing the list of flights matching a selection criterion. When the user selects one of the available flights, then the system turns out to the state shown in *(d)*. The user, at this state, can confirm his/her selection or change the fare profile of his/her flight. The GUI *(e)* finally shows the interface to confirm a flight selection. The user can accept the general terms and conditions and confirm the booking or withdraw the trip. In the latter case, the system asks the user to confirm the choice *(f)*. If the user does not confirm the withdrawing or opt to confirm the trip at the first stage, then the system shows a message confirming the booking.

To discuss the results that we got when testing the system, we present in Table 1 results of several batteries of tests in each GUI to perform the set of scenarios considered for the User Stories. We present sequentially each step of the target scenario and the errors that have been found throughout the batteries. Notice that successfully-run, repeated steps in sequential scenarios were omitted and some entire scenarios have been reused to reach a given state for the system before the test begins.

The first battery of tests has identified an error at the step 10 in the first scenario. The actual message displayed by the system was different, so the step was updated. Following this, when running the second scenario, the second battery of tests identified a problem with the identification/location of the field *"Departure Time Frame"*. The same occurred with the field *"Arrival Time Frame"* (third battery). In the fourth battery, the test identified the absence of the field *"Number of Passengers"*, which was

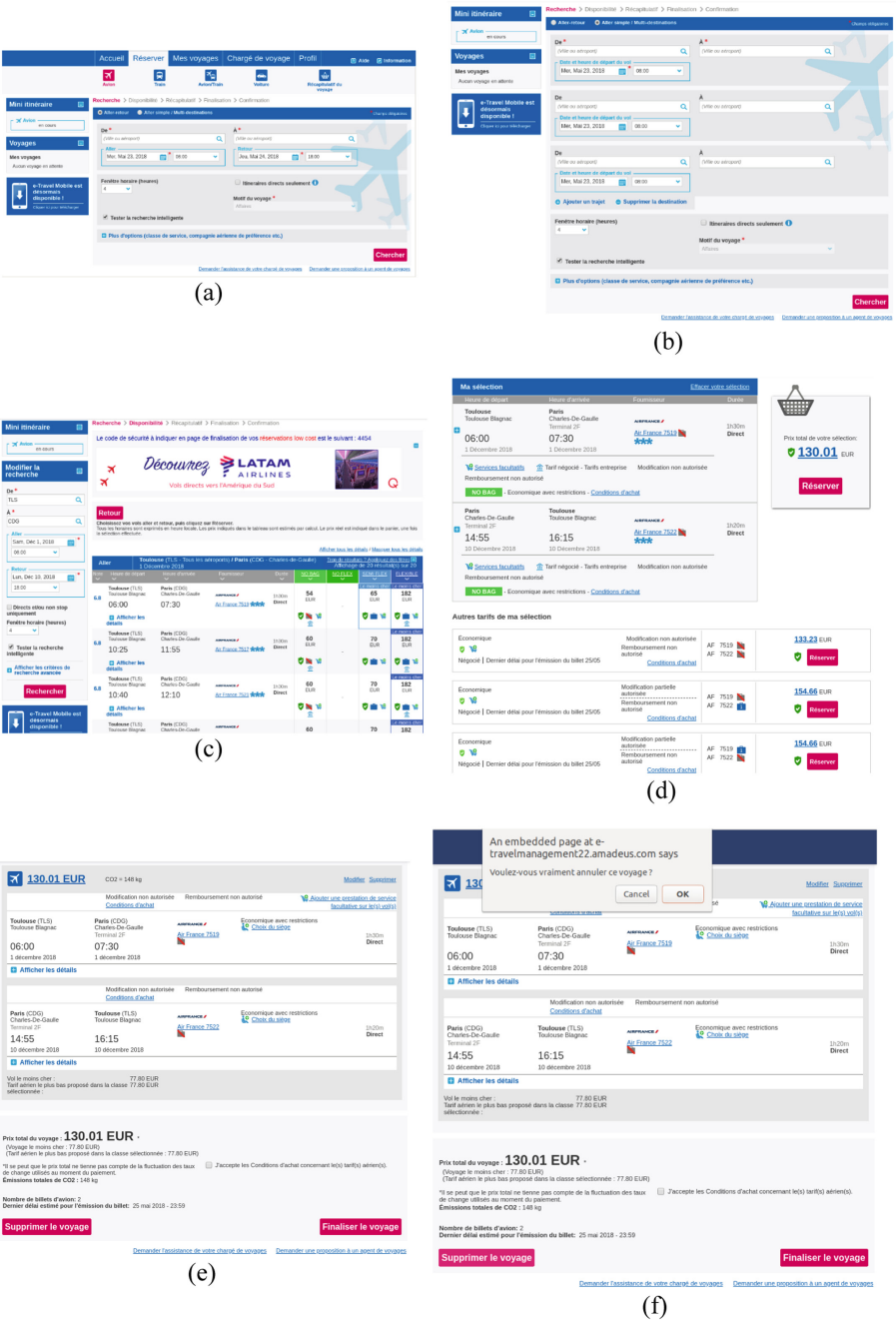


Fig. 9. GUIs of the existing web system for booking business trips.

Table 1. Test results on the GUIs. ✓ indicates that the step passed the test, ✗ indicates that an error has been identified, and ⚠ indicates that the test of that step has not been performed.

Battery		Steps
1		**Scenario**: *Successful Roundtrip Tickets Search*
	✓	1 - *Proceed to Login*
	✓	2 - *Reach the Travel Planet Search Page*
	✓	3 - *Given I go to "Flight Search"*
	✓	4 - *When I select "Round Trip"*
	✓	5 - *And I inform "Toulouse" and choose "Toulouse, Blagnac (TLS)" in the field "Departure"*
	✓	6 - *When I inform "Paris" and choose "Paris, Charles-de-Gaulle (CDG)" in the field "Destination"*
	✓	7 - *And I set "Sam, Déc 1, 2019" in the field "Departure Date"*
	✓	8 - *When I set "Lun, Déc 10, 2019" in the field "Arrival Date"*
	✓	9 - *And I submit "Search"*
	✗	10 - *Then will be displayed "**2. Sélectionner un voyage**"* (**Message not identified**)
2		**Scenario**: *Successful Roundtrip Tickets Search With Full Options*
	✓	Steps 2-3 and 5-7
	✗	16 - *And I set "08:00" in the field "**Departure Time Frame**"* (**Element not identified**)
3	✓	Steps 4 and 8
	✗	19 - *When I set "10:00" in the field "**Arrival Time Frame**"* (**Element not identified**)
4	✗	20 - *And I choose the option of value "2" in the field "**Number of Passengers**"* (**Element not found**)
5	✗	21 - *When I set "6" in the field "**Timeframe**"* (**Element not identified**)
6	✓	22 - *And I select "Direct Flights Only"*
	✗	23 - *When I choose the option of value "Economique" in the field "**Flight Class**"* (**Element not identified**)
7	✗	24 - *And I set "Air France" in the field "**Company 1**"* (**Value does not fit the field**)
8	✓	Step 9
	✓	26 - *Then will be displayed "Choisissez vos vols aller et retour, puis cliquez sur Réserver."*
		Scenario: *Successful One-way Tickets Search*
	✓	Steps 2-3 and 5-7
	✓	32 - *And I choose "One-way / Multidestination"*
	✓	Step 9
	✗	34 - *Then will be displayed "**Choisissez vos vols aller et retour, puis cliquez sur Réserver.**"* (**Message not identified**)
9		**Scenario**: *Successful Multidestination Tickets Search*
	✓	Steps 2-3, 32, and 5-7
	✗	41 - *When I inform "Paris" and choose "Paris, Charles-de-Gaulle (CDG)" in the field "**Departure**"* (**"Departure" already filled**)
10	✓	42 - *And I inform "Nice" and choose "Nice, Côte D'Azur (NCE)" in the field "Destination 2"*
	✓	Steps 8 and 9
	✗	45 - *Then will be displayed "**Choisissez vos vols, puis cliquez sur Réserver.**"* (**Message not identified**)
11		**Scenario**: *Search for Flights More Than One Year in Advance*
	✓	Steps 2-3 and 5-6
	✓	50 - *When I set "Dim, Déc 1, 2021" in the field "Departure Date"*
	✓	Steps 32 and 9
	✓	53 - *Then will be displayed "Erreur : Vous devez choisir une date de départ ultérieure comprise entre 4 heures et 11 mois. Veuillez sélectionner une autre date. (10032)"*
		Scenario: *Search for a Return Flight Before a Departure Flight*
	✓	Steps 2-3 and 5-6
	✓	58 - *When I set "Lun, Déc 10, 2019" in the field "Departure Date"*
	✓	Step 4
	✓	60 - *When I set "Sam, Déc 1, 2019" in the field "Arrival Date"*
	✓	Step 9
	✓	62 - *Then will be displayed "Erreur : La date de retour ne peut pas être antérieure à la date de départ."*
12		**Scenario**: *Select a Return Flight Searched Without Full Options*
	✓	63 - *Successful Roundtrip Tickets Search*
	✓	64 - *Given "Availability Page" is displayed*
	✓	65 - *When I click on "No Bag" referring to "Air France 7519"*
	✗	66 - *And I click on "**No Bag**" referring to "Air France 7522"* (**"No Bag" already filled**)
	✓	67 - *When I click on "Book"*
	✓	68 - *Then will be displayed "J'accepte les Conditions d'achat concernant le(s) tarif(s) aérien(s)."*
		Scenario: *Select a Return Flight Searched With Full Options*
	✓	69 - *Successful Roundtrip Tickets Search With Full Options*

(*continued*)

Table 1. (*continued*)

	✓	Step 64
	✓	71 - *When I click on "Air France 7519" referring to "No Bag"*
	✓	72 - *And I click on "Air France 7522" referring to "No Bag"*
13	✓	Steps 67-68
		Scenario: *Select a One-way Flight*
	✓	75 - *Successful One-way Tickets Search*
	✓	Steps 64, 71 and 67-68
		Scenario: *Select a Multidestination Flight*
	✓	80 - *Successful Multidestination Tickets Search*
	✓	Steps 64 and 71
	✗	83 - *And I click on "Air France 7700" referring to "No Bag"* (**Element "Air France 7700" not found**)
14	✓	Steps 67-68
		Scenario: *Confirm a Flight Selection*
	✓	86 - *Select a Return Flight Searched Without Full Options*
	✓	87 - *Given "Confirmation Page" is displayed*
	✓	88 - *When I choose "I accept the General Terms and Conditions."*
	⚠	89 - *And I click on "Finalize the trip"* (**Pending**)
	⚠	90 - *Then will be displayed "Votre voyage a été confirmé!"* (**Pending**)
		Scenario: *Confirm a Flight Selection (Full Version)*
	✓	91 - *Select a Return Flight Searched With Full Options*
	⚠	Steps 87-90 (**Pending**)
		Scenario: *Confirm a Flight Selection for a One-Way Trip*
15	✓	96 - *Select a One-way Flight*
	⚠	Steps 87-90 (**Pending**)
		Scenario: *Confirm a Flight Selection for a Multidestination Trip*
	✓	101 - *Select a Multidestination Flight*
	⚠	Steps 87-90 (**Pending**)
		Scenario: *Decline a Flight Selection*
	✓	Step 96 and 87
	✓	108 - *When I click on "Decline the trip"*
	✓	109 - *Then will be displayed "Voulez-vous vraiment annuler ce voyage ?" in the dialog box*
	✓	110 - *When I confirm the dialog box*
	✓	111 - *Then will be displayed "Votre voyage a été annulé."*

not implemented on the interface. In fifth and sixth batteries, the fields *"Timeframe"* and *"Flight Class"* were not located as well, due to the same reason of the fields *"Departure Time Frame"* and *"Arrival Time Frame"*. We noticed that these four fields are *Selects* (*Combo Boxes*), so, for some unknown reason, the implementation of such fields on the GUI does not allow to identify them either using IDs or XPaths. As during this study, we had no access to the source code of the application to implement some correction and run the test again, we decided to cut the respective steps off the scenario.

In the seventh battery, the test identified an error with the length of the field *"Company 1"*. The *Text Field* implemented on the GUI supports only two characters, so the value "Air France" does not fit. In fact, the user must inform a two-character internal code for the company he/she wants to select. In this case, the appropriate code for "Air France" is "AF", so the value in the step was updated to this value. In the eight battery of tests, all the steps for the second scenario succeeded running, and the third scenario started to run. An error was identified just in the last step due to conflicting messages.

In the ninth battery running the fourth scenario, the step 41 failed once the field *"Departure"* had already been field before with *"Toulouse, Blagnac (TLS)"* as the first departure of a multidestination trip. The field had to be renamed to correctly identify

the second departure field. It was named as *"Departure 2"*. The same solution was applied to the second instances of *"Destination"* (that was renamed to *"Destination 2"*), and *"Departure Date"* (that was renamed to *"Departure Date 2"*). In the tenth battery of tests, an error was identified just at the last step (45) once more due to conflicting messages. Finally, the eleventh battery got all the scenarios passed, which validated a User Story named *"Flight Ticket Search"*.

In the twelfth battery of tests, an error was found at the step 66 for the seventh scenario. The field *"No Bag"* has already been filled by the previous step (65), so the test fails. Besides that, the flight Air France 7522 was not available for booking anymore, so we changed for the flight Air France 7518. At the end, the solution was to give different names for each field referencing each mentioned flight. So, both steps were rewritten to *"When I click on '<flight>' referring to 'No Bag'"* in order to use unique identifiers for the flights.

The thirteenth battery of tests run successfully the eighth and ninth scenarios but stopped with an error at the step 83 of the tenth scenario. For multidestination trips, the GUI contains an additional step before reaching the second flight leg. The user must select the first flight leg, put the flight in a basket, and only then select the flight for the second flight leg. For the fourteenth battery of tests, we updated the respective scenario to add this additional step. That got all the scenarios passed, which validated a User Story named *"Select a Suitable Flight"*. In the fifteenth and last battery of tests, we got all the remaining scenarios for a User Story named *"Confirm Flight Selection"* passed. Nonetheless, we intentionally did not conclude the four first scenarios once they would effectively register a fake business trip in the system, so they were set as pending.

4.3 Discussion and Limitations

By summarizing the results presented above, below we can categorize the types of inconsistencies found by our testing approach when assessing the GUIs as follows:

- Message not identified
- Element or value not found
- Inexistent elements
- Values that do not fit the field
- Fields already filled in
- Element not identified

As presented above, we identified 6 different types in the tested scenarios. *"Elements not identified"* was the most frequent type and refers to elements that do not carry a unique and single identifier (or carry a dynamic generated one) and/or cannot be reached by using their XPaths. When observing the unsuccessful tries to find the fields *"Departure Time Frame"* and *"Arrival Time Frame"*, for example, we remarked that it is a recurrent problem when automating testing on GUIs. Some web frameworks for developing the presentation layer dynamically generates different identifiers each time the GUI is charged, which makes very hard the work of previously identifying them to implement the test. Besides that, some developers skip informing unique identifiers for the fields. XPath identifiers help in most cases, but there still are some situations where the identification of locators gets very compromised.

Constant changing, or conflicting messages is another frequent issue (type of inconsistency *"message not identified"*). Messages sometimes change in the GUI and the requirements specification is not updated accordingly. As a consequence, the message specified in the step to be verified in the user interface is not found on the screen. Not identifying elements or values due to dynamic data behavior is also an issue. The type of inconsistency *"element or value not found"* refers to fields or values that are expected to be shown on the user interface (and are able to be identified by the locators there) but, due the dynamic data behavior in the system, are not shown up. An example from the case study is a flight, which was mentioned to be verified as an example of data value in the step and was not identified in the list of resultant flights because it was not available for booking anymore. There is also the case of elements that are really inexistent on the user interface (type of inconsistency *"inexistent elements"*). These elements are mentioned in the step as part of the requirements specification, but simply have not been implemented in the GUI.

Fields that were already filled in when a given step tries to reach them are also a source of inconsistencies (type of inconsistency *"fields already filled in"*). As happened in the case study when testing the fields *"Departure"* and *"Destination"* for a multidestination trip, due to the second flight leg, the elements were referenced with the same name more than once. When the test tried to fill in the same field a second time for the second flight leg, the inconsistency has shown up. In this case, both the step and the mapping of interaction elements on the user interface must be updated to reference unequivocally different elements for each desired interaction.

The last type of inconsistency identified refers to values mentioned in the step that do not fit the field they were designed to fill in (type of inconsistency *"values that do not fit the field"*). During the case study, the field *"Company 1"* was expected to receive the value "Air France" as described in the step, but the concrete field *"Company 1"* on the GUI had been modeled to support only two characters. This type of inconsistency can also be extended to other incompatibilities between the type of data expected and what the field actually supports. Examples include strings to be filled in number-only fields, unformatted numbers to be filled in date-time fields, and so on.

By analyzing the variety of inconsistency problems that have been identified in this case study, we can remark that some types of inconsistencies have shown to be more critical than others. While simple inconsistencies like messages and elements not found (or even inexistent) are easy to solve, some other inconsistencies can reveal crucial problems of modeling or important incompatibilities between the requirements specification and its design in GUIs. Fields already filled-in, for example, denotates inconsistencies that expose important design errors.

It is also worth mentioning that our approach implements a strategy of co-execution to assess user requirements and GUIs. Unlike static approaches, which benefit from an instantaneous consistency checking to analyze several hundreds of source files at the same time; co-execution approaches benefit from allowing running models simultaneously with visual feedback at real-time about the correspondence of entities that are being assessed in each model. Co-execution approaches usually have the drawback of demanding a high investment to prepare and adapt the artifacts for testing. In our approach however, such an investment is restricted to the mapping of widgets on the respective GUIs under testing. As the great benefit of co-execution on GUIs is

providing visual feedback during the execution simulating a real user interacting with interaction elements at real-time, this process can end up being very slow with the growing number of user interfaces and scenarios to be tested. As far as the simulation of real user actions is not a concern, such a drawback can be reduced by using GUI-less browser implementations such as HtmlUnit, which benefits from high-level manipulation of web pages without the need of actually running the browser and co-executing the simulated user actions. Like static approaches, this strategy is suitable for environments demanding high-availability and continuous testing.

As a threat to validity, we report that both the conduction of the study and the interpretation and analysis of the results have been initially made by the authors. So, to mitigate a bias in the interpretation of such results, they were cross-checked by independent reviewers.

5 Conclusion

The approach we present in this paper has the main advantage of ensuring a reliable automated acceptance testing of user requirements. Due to its flexible architecture, the approach also succeeds to provide automated testing for GUIs developed under whatever technology for designing the presentation layer of web pages. For other environments, it would be enough to use external testing frameworks adapted to running tests on such environments. The gain in terms of reuse, maintenance, and coverage that can be obtained with the approach is another crucial advantage to highlight. Except for the initial manual assignment of interaction element identifiers on the considered GUIs to the "*MyPage*" class, the approach runs in a fully automated process. When compared with manual assessment, our automated approach benefits from high-availability of tests with a high human-readability. The automation also helps to cover a larger set of scenarios that could hardly and costly be covered by manual inspections.

This approach has also been extended and adapted to assess early user interface design artifacts [14–18]. As an integrated approach, User Stories can also be assigned to automatically assess task models and GUI prototypes in different levels of abstraction, ensuring a consistent verification, validation, and testing (VV&T) approach for interactive systems with immediate feedback about the consistency of artifacts and user requirements since the early stages of development.

Next steps on this research include evaluating the impact of maintaining and successively evolving GUIs throughout a real software development process where issues related to scalability should be addressed. Concerning the tools, they are not yet freely available for download and general use since the interface for testing is still running on console mode. When evolving it in the future, a technological transfer to the industry could be envisioned. Concerning the features, in addition to the current resource of providing immediate feedback for the recognized steps, it is also envisioned the development of an Eclipse plugin to suggest and autocomplete steps of the User Stories scenarios based on the interactive behaviors of the ontology.

References

1. Hellmann, T.D.: Automated GUI Testing for Agile Development Environments. University of Calgary (2015)
2. Chen, W., Tsai, T., Chao, H.: Integration of specification-based and CR-based approaches for GUI testing. In: 19th International Conference on Advanced Information Networking and Applications (2005). https://doi.org/10.1109/AINA.2005.223
3. Bowen, J., Reeves, S.: UI-driven test-first development of interactive systems. In: Engineering Interactive Computing Systems, pp. 165–174 (2011). https://doi.org/10.1145/1996461.1996515
4. Holmes, A., Kellogg, M.: Automating functional tests using selenium. In: AGILE 2006 (2006). https://doi.org/10.1109/AGILE.2006.19
5. Meszaros, G.: Agile regression testing using record & playback. In: Object-Oriented Programming, Systems, Languages, and Applications (2003). https://doi.org/10.1145/949344.949442
6. Andersson, J., Geoff, B.: The video store revisited yet again: adventures in GUI acceptance testing. In: Extreme Programming and Agile Processes in Software Engineering (2004). https://doi.org/10.1007/978-3-540-24853-8_1
7. Chelimsky, D., Astels, D., Helmkamp, B., North, D., Dennis, Z., Hellesoy, A.: The RSpec Book: Behaviour Driven Development with RSpec, Cucumber, and Friends. Pragmatic Bookshelf (2010)
8. Cohn, M.: User Stories Applied for Agile Software Development. Addison-Wesley (2004)
9. Silva, T.R., Hak, J.-L., Winckler, M.: A behavior-based ontology for supporting automated assessment of interactive systems. In: Proceedings of the 11th IEEE International Conference on Semantic Computing (ICSC 2017), pp. 250–257 (2017). https://doi.org/10.1109/ICSC.2017.73
10. Silva, T.R., Hak, J.-L., Winckler, M.: A formal ontology for describing interactive behaviors and supporting automated testing on user interfaces. Int. J. Semant. Comput. **11**(04), 513–539 (2017). https://doi.org/10.1142/S1793351X17400219
11. Beck, K.: Test Driven Development: By Example, 1st edn. Addison-Wesley Professional (2002)
12. North, D.: What's in a Story? (2019). https://dannorth.net/whats-in-a-story/. Accessed 01 Jan 2019
13. Silva, T.R., Winckler, M., Bach, C.: Evaluating the usage of predefined interactive behaviors for writing user stories: an empirical study with potential product owners. Cogn. Technol. Work 1–21 (2019, in press). https://doi.org/10.1007/s10111-019-00566-3
14. Silva, T.R., Winckler, M.A.A.: Towards automated requirements checking throughout development processes of interactive systems. In: 2nd Workshop on Continuous Requirements Engineering (CRE), REFSQ 2016, pp. 1–2 (2016)
15. Silva, T.R.: Definition of a behavior-driven model for requirements specification and testing of interactive systems. In: Proceedings of the 24th International Requirements Engineering Conference (RE 2016), pp. 444–449 (2016). https://doi.org/10.1109/RE.2016.12
16. Silva, T.R., Hak, J.-L., Winckler, M.: Testing prototypes and final user interfaces through an ontological perspective for behavior-driven development. In: Bogdan, C., Gulliksen, J., Sauer, S., Forbrig, P., Winckler, M., Johnson, C., Palanque, P., Bernhaupt, R., Kis, F. (eds.) HCSE/HESSD -2016. LNCS, vol. 9856, pp. 86–107. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44902-9_7

17. Silva, T.R., Hak, J.-L., Winckler, M.: An approach for multi-artifact testing through an ontological perspective for behavior-driven development. Complex Syst. Inform. Model. Q. **7**, 81–107 (2016). https://doi.org/10.7250/csimq.2016-7.05
18. Silva, T.R., Winckler, M.: A scenario-based approach for checking consistency in user interface design artifacts. In: Proceedings of the 16th Brazilian Symposium on Human Factors in Computing Systems (IHC 2017), vol. 1, pp. 21–30 (2017). https://doi.org/10.1145/3160504.3160506