Ensuring the Consistency Between User Requirements and GUI Prototypes: 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. In a user-centered design process, graphical user interface (GUI) prototypes may be seen as an important early artifact to design and validate user requirements before making strong commitments with a full-fledged version of the user interface. Ensuring the consistency of GUI prototypes with other representations of the user requirements is then a critical aspect of the design process. This paper presents an approach which extends Behavior-Driven Development (BDD) by employing an ontology in order to provide automated assessment for GUI prototypes as design artifacts. 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 check the consistency of Balsamiq prototypes which were reengineered from 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 analyzed artifacts, allowing to build an effective correspondence between user requirements and their representation in GUI prototypes.

Keywords: Behavior-Driven Development (BDD) · User Stories · GUI Prototypes · User Requirements Assessment

1 Introduction

In iterative processes, the design of graphical user interfaces (GUIs) can evolve all along the software development process as a result of requirements evolution and change, or the need of understanding and validating a given interpretation of requirements [1]. While the beginning of the project usually requires a low-level of formality with GUI prototypes being hand-sketched to explore design solutions and clarify user requirements, the development phase requires more refined versions frequently describing presentation and dialog aspects of interaction. Full-fledged versions of user interfaces are generally produced only later in the design process, and frequently corresponds to how the user “see” the system. In the users’ point of view, if some feature is not available through the presentation of a user interface, this feature does not exist for them.

Behavior-Driven Development (BDD) [2] has stood out in the software engineering community as an effective approach 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 [3] which are easily understandable for both technical and non-technical stakeholders. In addition, User Stories allow specifying “executable requirements”, i.e. requirements that can be directly tested from their textual specification. Despite its benefits providing automated testing of user requirements, BDD and other testing approaches focus essentially on assessing interactive artifacts that are produced late in the design process, such as full-fledged versions of user interfaces. As far as early artifacts such as rough GUI prototypes are a concern, current approaches offer no support for automated assessment.

Motivated by such a gap, this paper presents an approach based on BDD and User Stories to support the specification and the automated assessment of user requirements on low-fidelity web GUI prototypes designed along the development cycle of interactive systems. The approach helps to align methods for GUI design and assessment with methods for engineering interactive systems. On one hand, our method proposes to engineer GUIs by guiding the development team (especially designers) to avoid design solutions that conflict with the user requirements. On the other hand, the method also helps to pinpoint eventual violations of user requirements at the user interface level, helping designers to have a better understanding of where and when violations of user requirements occur.

The common-ground of concepts for describing the prototypes as well as the set of user-system interactive behaviors is provided by means of an ontology [4, 5]. Requirements in BDD stories can be formulated and tested at two levels: the domain level and the interaction level. Our approach targets requirements at the interaction level, and this paper shows how the ontology can support test automation of requirements at this level without manually coding the tests. The following sections present the related works and foundations, the proposed approach with its technical implementation, and the results we got by assessing the reengineered GUI prototypes from an existing web system to book business trips.

2 Related Works

Artifacts other than final versions of user interfaces are not commonly tested. A common argument is that they cannot be “executed” in order to be tested. Since long time ago, the design aspect of early representations of user interfaces is usually only inspected manually in an attempt to verify its adequacy [6]. Inspections can be of different types including formal technical reviews, walkthroughs, peer desk check, informal ad-hoc feedback, and so on [7]. When evaluation of the user requirements representation on such artifacts is considered, requirements traceability techniques are employed as a way to trace such requirements along their multiple versions (horizontal traceability) or along their representation in another artifacts (vertical traceability) [8].

Some approaches concentrated efforts in providing automated tools to keep compatibility between requirements and their own artifacts. Luna et al. [9], for example, propose WebSpec which is a requirement artifact that can be used in conjunction with mockups to provide UI simulations, allowing some level of requirements validation, but not for out-of-approach UI prototypes like Balsamiq. Buchmann and Karagiannis [10] presents a modeling method for the elicitation of requirements for mobile apps that enables semantic traceability for the requirements representation. The method however is not focused on UI prototypes and can only validate requirements modeled within the approach. As far as a common vocabulary for the dialog aspect of a UI is at a concern, SWC [11] and SXCML [12] offer a language based on the state machine concepts. PANDA [13] is a tool which exploits the aforementioned ontology [4, 5] to design medium-to-high fidelity executable prototypes allowing the use of interactive behaviors semantically meaningful to the user interface elements. However, PANDA does not support the design and assessment of low-fidelity wireframes and sketches. Other solutions focused on generating UIs from other software models, which in theory would keep them consistent, is also a topic that has received attention for long time [14–17].

When considering user requirements specified through BDD to evaluate software models, studies have been conducted to explore its use within user-centered [18] and agile [19] approaches to support enterprise modeling, when analyzing automated acceptance testing to support BDD traceability [20], as well as its compatibility with business modeling [21, 22] and BPMN [23].

There is also an intrinsic relationship between user interface design and task modeling, when considered in a user requirements perspective. Some authors have even tried to establish linguistic task modeling for designing user interfaces [24] where a notation enables identification of task input elements based on the task state diagram and dynamic tasks. Martinie et al. [25], followed by Campos et al. [26], propose a tool-supported framework and a model-based testing approach to support linking task models to an existing, executable, and interactive application, defining a systematic correspondence between the user interface elements and user tasks. The problem with this approach is that it only covers the interaction of task models with a concrete fully-functional user interfaces, not covering user interface prototypes.

Finally, previous studies [27, 28] which analyzed the current state-of-the-art prototyping tools have concluded that features to support (at some level) the assessment of prototypes and scenario-based specifications have been covered by less than 10% of the tools analyzed.

3 Foundations

3.1 Behavior-Driven Development and User Stories

Behavior-Driven Development (BDD) is a specialization of Test-Driven Development (TDD) [29, 30], 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. User Stories were firstly proposed by Cohn [3] 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 [3, 31] 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 BDD and has been named by North [31] as a “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 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 *step*.

This format allows specifying executable requirements by means of a Domain-Specific Language (DSL) provided by Gherkin [32]. Gherkin is a DSL that has been developed for BDD to let users and developers describe software behavior without detailing how that behavior is implemented. By using this language, requirements specifications can be used to implement automated tests, which can conduct to living documentation, making easier for clients and other stakeholders to set their final acceptance tests. The drawback of using plain-vanilla Gherkin to specify requirements at the domain level, as proposed in [31], is to require that a developer manually implements the tests corresponding to each individual step of the scenarios.

3.2 Ontological Support for GUI Automated Testing

A GUI prototype is an early representation of a graphical user interface for an interactive system. In a software development perspective, GUI prototypes can be seen as concrete and tangible design artifacts [33]. By running simulations on prototypes, we can also determine potential scenarios that users can perform in the system and relate them to the requirements. When such requirements are specified through User Stories, a recurrent problem is that they often contain semantic inconsistencies. For example, it is not rare to find scenarios that specify an action such as a selection to be made in a widget such as a *Text Field* that does not support that action. To tackle this problem, previous works explored the use of an ontology describing common behaviors with a standard vocabulary for writing User Stories as scenario artifacts [4, 5]. The main benefit of this strategy is that User Stories using this common vocabulary can support specification and execution of automated test scenarios on GUI prototypes. 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 prototypes.

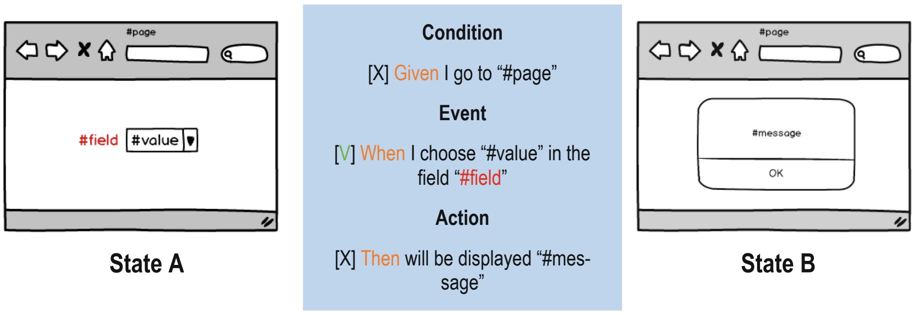

#page
#field
#value
State A
Condition
[X] Given I go to "#page"
Event
[V] When I choose "#value" in the field "#field"
Action
[X] Then will be displayed "#message"
#page
#message
OK
State B

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

The dialog part of a GUI, as illustrated by Fig. 1, is described in 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.

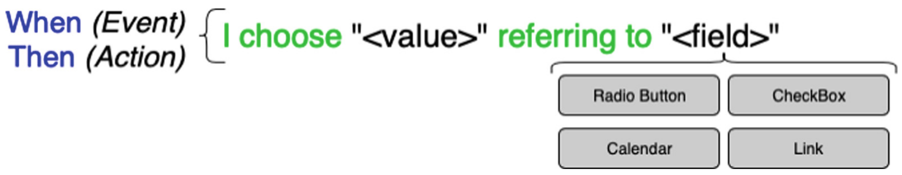


Fig. 2. Structure of a behavior as specified in the ontology.

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 element supported by this behavior: "*Radio Button*", "*CheckBox*", "*Calendar*" and "*Link*".

The ontological model describes only behaviors that report steps performing actions directly on the user interface through interaction elements, i.e. behaviors referring to the interaction level in the user requirements description. 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.

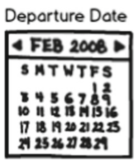
Table 1. Example of interactive behaviors described in the ontology. *Transition*: *(C)ontext*, *(E)vent*, *(A)ction*.

Behavior	Transition			Interaction Elements
	C	*E*	*A*	
choose ≡ *select*		▨	▨	*Calendar*, *Checkbox*, *Radio Button*, and *Link*
chooseByIndexInTheField		▨	▨	*Dropdown List*
chooseReferringTo		▨	▨	*Calendar*, *Checkbox*, *Radio Button*, and *Link*
chooseTheOptionOfValueInTheField		▨	▨	*Dropdown List*
clickOn		▨	▨	*Menu*, *Menu Item*, *Button*, and *Link*
clickOnReferringTo		▨	▨	*Menu*, *Menu Item*, *Button*, and *Link*
doNotTypeAnyValueToTheField ≡ *resetTheValueOfTheField*		▨		*Text Field*
goTo	▨	▨	▨	*Browser Window*
isDisplayed	▨	▨	▨	*Browser Window*
setInTheField ≡ *tryToSetInTheField*		▨	▨	*Dropdown List*, *Text Field*, *Autocomplete*, and *Calendar*
typeAndChooseInTheField ≡ *informAndChooseInTheField*		▨	▨	*Autocomplete*
willBeDisplayed			▨	*Text*

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 a 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. Table 1 exemplifies some of these interactive behaviors, the transition component during which they can be triggered and the set of corresponding interaction elements.

4 The Proposed Approach for Automated Assessment

There are multiple notations and tools with different implementations for designing and modeling GUI prototypes [27]. Among these, we have chosen to implement a proof of concept of our approach with the wireframe sketching tool Balsamiq[1] in its current version (2.2.28), since it is a wide-spread and highly-regarded prototyping tool and uses a documented XML format for persisting the prototypes. Nonetheless, we have designed a flexible and open architecture where other notations and tools could benefit from our approach by just implementing new classes in accordance with their own patterns to implement and model prototypes.



```
   <control  controlID="15"  controlTypeID="__group__"  x="588"
y="403"  w="96"  h="117"  measuredW="96"  measuredH="117"  zOr-
der="6" locked="false" isInGroup="-1">
     <groupChildrenDescriptors>
       <control controlID="0" controlTypeID="com.balsamiq.mock-
ups::Label"  x="0"  y="0"  w="-1"  h="-1"  measuredW="92"  mea-
suredH="21" zOrder="0" locked="false" isInGroup="15">
         <controlProperties>
           <text>Departure%20Date</text>
         </controlProperties>
       </control>
       <control controlID="1" controlTypeID="com.balsamiq.mock-
ups::Calendar" x="0" y="21" w="96" h="96" measuredW="96" mea-
suredH="96" zOrder="1" locked="false" isInGroup="15"/>
     </groupChildrenDescriptors>
   </control>
```

Fig. 3. Grouped field "Departure Date" and its XML source file.

```
   <control  controlID="14"  controlTypeID="com.balsamiq.mock-
ups::Button" x="1051" y="459" w="-1" h="-1" measuredW="63" mea-
suredH="27" zOrder="8" locked="false" isInGroup="-1">
     <controlProperties>
       <text>Search</text>
     </controlProperties>
   </control>
```

Fig. 4. Button "Search" and its XML source file.

The assessment of GUI prototypes in our approach is an automated process. Our strategy for testing Balsamiq prototypes is parsing their XML source files and

[1] balsamiq.com.

identifying UI elements that match the ontology description for each behavior. The first step for assessing such prototypes is therefore getting from the ontology the list of UI elements which support the behavior under testing. Taking a step "*And I set 'Valid Departure Date' in the field 'Departure Date'*" as an example, according to the ontology, the associated interactive behavior "*setInTheField*" is supported by the UI elements "*Dropdown List*", "*Text Field*", "*Autocomplete*" and "*Calendar*", when performing an action (*Then*) or an event (*When*) in a state machine transition.

After getting such a list of supported UI elements, we analyze the Balsamiq XML file to identify firstly if a field named "Departure Date" exists (Fig. 3). This is made by reading the tag "*<text>*" identified in the parent tag *"<controlProperties>"* for a given *"<control>"* element. If such a field exists, i.e. there is a tag "*<text>*" carrying its name (case insensitive), so we retrieve which interaction element is associated with it. At this point, we implemented a reference file containing the mapping between the abstracted interaction elements in the ontology and the Balsamiq concrete implementation of such elements.

Notice at the left side of Fig. 3 that the field "Departure Date" has been modeled with a "*Calendar*", i.e. the UI designer has chosen the UI element "*Calendar*" to design the field "Departure Date". Thus, by checking the list of supported UI elements in the ontology, we find that the behavior "*setInTheField*", addressed by the field "Departure Date", is supported by a "*Calendar*" element, so the test passes. If other elements than "*Dropdown List*", "*Text Field*", or "*Autocomplete*" had been chosen, the test would fail.

```
foreach step from US Scenarios do
    supportedUIElements <- correspondent UI Elements from the ontology
    fieldName <- name of the UI Element from the step
    foreach UI Element from the Balsamiq prototype do
        if the attribute text is equal to fieldName && is not in group then
            if the attribute controlTypeID is equal to one of the
                supportedUIElements then
                    numElements++
            endif
        else if the attribute text is equal to fieldName && is in group then
            if the attribute controlTypeID of some member of the group is
                equal to one of the supportedUIElements then
                    numElements++
            endif
        endif
    endforeach
endforeach

if numElements == 1 show Success
else show Fail
```

Fig. 5. Testing algorithm for assessing Balsamiq prototypes.

Our algorithm (see Fig. 5) must take into account that Balsamiq has two methods for representing UI elements on its XML source files. They can be directly assigned with a unique "*controlID*" (Fig. 4) or be part of a group that encompasses a *label* and the UI element itself (Fig. 3). In the first case, the *label* "Search" in Fig. 4 is directly associated with the element "*Button*" itself (*com.balsamiq.mockups::Button*). In the second case, we can notice the *label* for "Departure Date" in Fig. 3 is part of a group (*isInGroup='15'*). In the same group, but with other "*controlID*", we find the element

"*Calendar*" itself (*com.balsamiq.mockups::Calendar*). When looking for matching elements, the algorithm identifies which Balsamiq method has been used to design the element. If the parent tag is a *label*, it means that the element is part of a group that contains the element itself in a sibling tag. This sibling tag is then identified by reading the attribute "*isInGroup*". If the parent tag is not a *label*, so it is already the element itself. After identifying it, the algorithm checks if some of the UI elements received from the ontology matches with the element from the prototype that is being investigated. If so, the variable "*numTasks*" is increased by one. After investigating the whole set of tags, the value of this variable is returned and must be equal to "1", which means only one UI element for representing the "*fieldname*" has been found. If this value is equal to "0", it means that no UI element has been found in the prototype with that "*fieldname*", while if it is greater than "1", it means that more than one UI element has been found with the same "*fieldname*". In both cases, the algorithm identifies the failure and the test does not pass. This process is conducted for each step of the scenario.

Notice that for GUI prototypes at this level of refinement, we only assess the presentation aspect of the prototype. We are not considering for testing at this level the dialog aspect and the consequent dynamic aspect of the interaction. It means that to check the consistency of the UI elements modeled in the prototype, we only consider the presence (or the absence) of the right kind of interaction elements on the GUI prototype where the interaction is supposed to occur. Behaviors that perform a state transition (e.g. navigating from one screen to another or getting mock values from the fields as a result of an interaction) are not being taken into account in the results.

4.1 Tool Support

The algorithm presented in the previous section has been implemented in Java as an open source project and integrates different frameworks such as JUnit and JDOM. Figure 6 represents the flow of calls we have designed for running tests on Balsamiq prototypes. The flow starts with the class "*MyTest.java*" that is a JUnit class in charge of triggering the battery of tests (its content is illustrated in Fig. 7). This class indicates which files will be used for testing (flow 1). These files are distributed in two packages. The first one contains the User Story files ("*.story*" files where the scenarios for testing are), and the second one contains the Balsamiq files (which are the BMML source files of Balsamiq prototypes). Both the User Story and the Balsamiq files remain separate files in each package and are tested individually. In the example provided in Fig. 7, it has been indicated for testing the User Story "*Flight Ticket Search.story*" on the Balsamiq prototype "*Book Flights.bmml*".

Each one of the steps in the User Story under testing makes calls to the class "*MySteps.java*" (flow 2) that knows which behaviors are supported. Based on the behavior referenced by the step, this class makes a call to the class "*Balsamiq.java*" to get the list of Balsamiq interaction elements that supports such a behavior (flow 3). The class "*Balsamiq.java*" in its turn makes a call to the class "*MyOntology.java*" (flow 4) in charge of reading the OWL file of the ontology and recovering the list of abstract interaction elements supported by a given behavior. Such a list is then returned to the class "*Balsamiq.java*" (flow 5) that checks, for each abstract element returned by the ontology, which are the corresponding concrete interaction elements in Balsamiq in charge of implementing the mentioned behavior (flow 6). This mapping is recovered from the file "*Balsamiq.mapping*" (flow 7).

Afterward, the class "*Balsamiq.java*" returns such a list with the concrete Balsamiq elements to the class "*MySteps.java*" (flow 8) that originally made the call. With the list of supported Balsamiq elements for the step under testing, the class "*MySteps.java*" calls to the class "*MyXML.java*" (flow 9) in charge of parsing the Balsamiq "*.bmml*" file (flow 10). This parsing aims to check if the prototype carries the interaction element mentioned in the step under testing, and if so, if such an element supports the behavior mentioned in the step. The result of this parsing is then returned to the class "*MySteps.java*" (flow 11). At this point, based on the algorithm presented in the previous section, we verify how many instances have been found for the searched element. Finally, the class "*MySteps.java*" asserts the value and returns the result to the class "*MyTest.java*" (flow 12) that indicates if the test has failed or not.

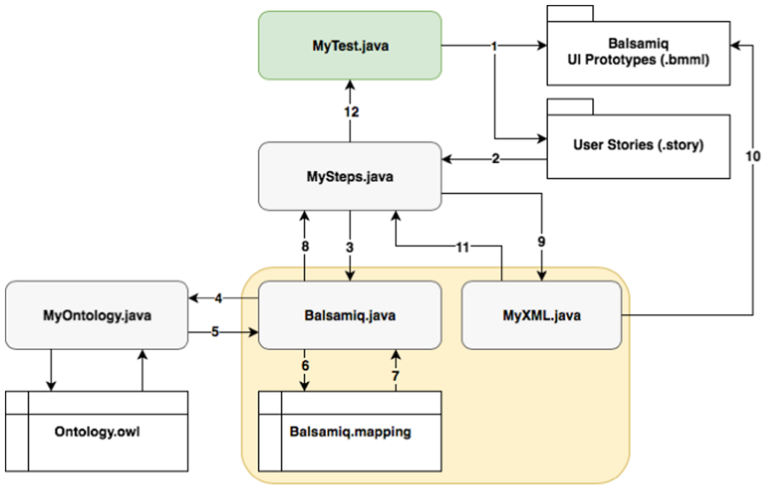
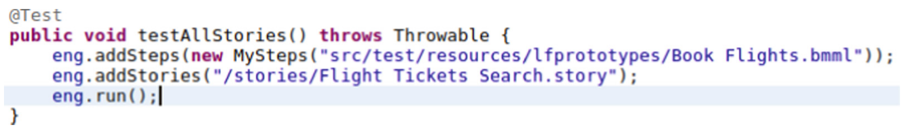


Fig. 6. Flow of calls for running tests on Balsamiq prototypes. (Color figure online)



```
@Test
public void testAllStories() throws Throwable {
    eng.addSteps(new MySteps("src/test/resources/lfprototypes/Book Flights.bmml"));
    eng.addStories("/stories/Flight Tickets Search.story");
    eng.run();
}
```

Fig. 7. "*MyTest.java*": class for running tests on Balsamiq prototypes.

Notice the independence of the components assigned at the core of the structure represented in Fig. 6 (highlighted in yellow). Those components are related to the particularities of test implementation for Balsamiq prototypes. "*Balsamiq.java*" treats the demands for getting the correspondent abstract interaction elements from the ontology and translates them to the concrete interaction elements implemented by Balsamiq. "*Balsamiq.mapping*" provides such a translation. Finally, "*MyXML.java*" is in charge of parsing the BMML files of Balsamiq, searching for the element under testing. By this way, we deliver a flexible architecture allowing, in the future, that UI prototypes modeled by other prototyping tools could also be tested by just implementing new interfaces for this core.

In summary, considering the presented architecture, to setup and run a battery of tests, we must: (*i*) place the set of BMML files that will be tested in the package "Balsamiq UI Prototypes", (*ii*) place the set of User Stories files ("*.story*") that will be tested in the package "User Stories", (*iii*) indicate in the "*MyTest*" class which prototype will be tested with which User Story (only a prototype with a User Story at a time), and (*iv*) run the "*MyTest*" class as a JUnit test.

5 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 [34], domain experts were invited to produce some User Stories to describe a feature they considered important to that system. This previous study aimed at analyzing how the experts write User Stories (and the difficulties they have) whilst using a predefined template. In the present study, the gathered User Stories were refined to get a representative set of user requirements to be assessed on the GUI prototypes of the existing system, showing how we can automate the test of Balsamiq prototypes. This refinement was only necessary because, in the context of the previous study, the experts were deliberately not trained to use the ontology vocabulary, so we had to refine the User Stories produced by them to use this vocabulary and to include additional test scenarios.

To obtain the GUI prototypes for testing, we have studied the current implementation of the existing system, and by applying reverse engineering [35], we redesigned the targeted GUI prototypes in Balsamiq. The aim of this software reengineering was to have such artifacts to run our tests and examine which types of inconsistencies our approach would be able to identify.

5.1 Methodology

To conduct the case study, we first set up an initial version of the User Stories and their test scenarios. We then reengineered initial versions of Balsamiq prototypes from the existing web system. Following this step, we ran the initial version of User Stories to initial versions of the prototypes designed with Balsamiq.

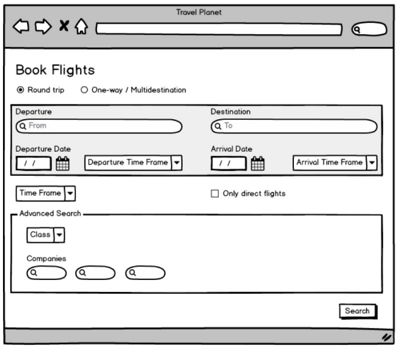
The strategy we follow for running tests on the prototypes parses each step of the scenario at a time, so if an error is found out, the test stops until the error is fixed. That requires to run several batteries of tests until having the entire scenario tested. It leads us to fix all the inconsistencies step-by-step, and consequently to get 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 artifact 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 an artifact can be different than that ones used to test another artifact 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 in other artifacts and made some artifact (that so far was consistent with the requirements) inconsistent again.

5.2 Results

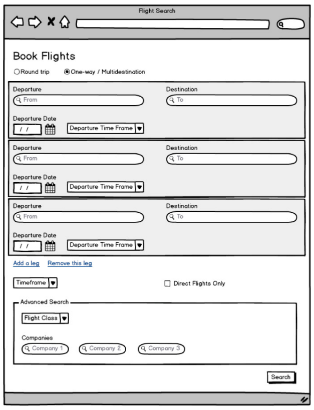
In total, we set up for assessment 3 User Stories with 15 different scenarios and reengineered 11 Balsamiq prototypes. The sequence of prototypes (*a–h*) in Fig. 8 shows the different states and designs of some of the developed prototypes. The prototype *(a)* presents our first design for a UI prototype to search flights. The figure represents a UI for searching flights based on a round trip (and *(b)* based on a one-way/multidestination trip). The prototype *(c)* presents the next UI in sequence, showing the list of flights matching the selection criteria. 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 prototype *(e)* finally shows the UI of confirmation of 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 *(g)*, and if confirmed, cancels the trip *(h)*. 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 *(f)*. To discuss the results that we got by testing different versions of Balsamiq prototypes, we present in Table 2 results of several batteries of testing in each version of the prototypes developed to perform a successful roundtrip booking (one of the possible full scenarios). We present sequentially each step of the target scenario, the corresponding elements in the Balsamiq source files, errors that have been found in a given battery, and finally the subsequent battery of tests following the fixes. Once the goal is to assess the most possible number of interaction elements in the prototype, we have chosen to run our tests presented below on the full versions of the scenarios, i.e. in those ones interacting with all the optional fields.

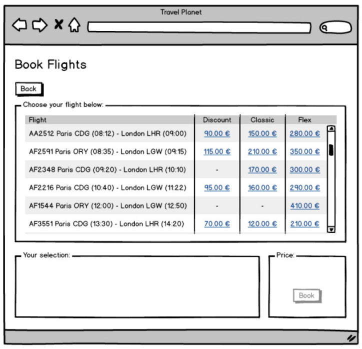
We notice that the first battery of tests found an error already in the first step (*"Given I go to 'Book Flights'"*). It was expected a correspondent element "*BrowserWindow*" associated to the name "Book Flights" in the prototype, but the element found was a "*SubTitle*". The "*BrowserWindow*" was named "Travel Planet", the name of the system under testing. As the behavior "*goTo*" is supposed to be performed only in a window (and its variants), such a step could not be performed in a field describing a "*SubTitle*", which is a semantically inconsistent field for that behavior. At the end, the window ended up being named "Flight Search" and both the scenario and the prototype have been updated accordingly.



(a)

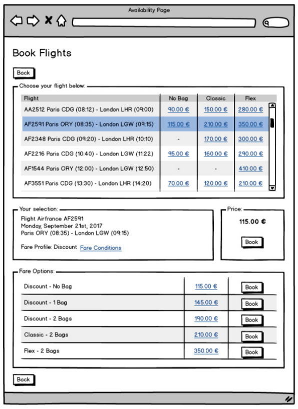


(b)



Flight	Discount	Classic	Flex
AA2512 Paris CDG (08:12) - London LHR (09:00)	90.00 €	150.00 €	280.00 €
AF2591 Paris ORY (08:35) - London LGW (09:15)	115.00 €	210.00 €	350.00 €
AF2348 Paris CDG (09:20) - London LHR (10:10)	-	170.00 €	300.00 €
AF2216 Paris CDG (10:40) - London LGW (11:22)	95.00 €	160.00 €	290.00 €
AF1544 Paris ORY (12:00) - London LGW (12:50)	-	-	410.00 €
AF3551 Paris CDG (13:30) - London LHR (14:20)	70.00 €	120.00 €	210.00 €

(c)



Flight	No Bag	Classic	Flex
AA2512 Paris CDG (08:12) - London LHR (09:00)	90.00 €	150.00 €	280.00 €
AF2591 Paris ORY (08:35) - London LGW (09:15)	115.00 €	210.00 €	350.00 €
AF2348 Paris CDG (09:20) - London LHR (10:10)	-	170.00 €	300.00 €
AF2216 Paris CDG (10:40) - London LGW (11:22)	95.00 €	160.00 €	290.00 €
AF1544 Paris ORY (12:00) - London LGW (12:50)	-	-	410.00 €
AF3551 Paris CDG (13:30) - London LHR (14:20)	70.00 €	120.00 €	210.00 €

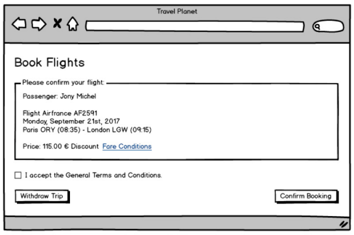
Flight Airfrance AF2591
Monday, September 21st, 2017
Paris ORY (08:35) - London LGW (09:15)

Fare Profile: Discount Fare Conditions

115.00 €

Discount - No Bag	115.00 €	Book
Discount - 1 Bag	145.00 €	Book
Discount - 2 Bags	190.00 €	Book
Classic - 2 Bags	210.00 €	Book
Flex - 2 Bags	350.00 €	Book

(d)

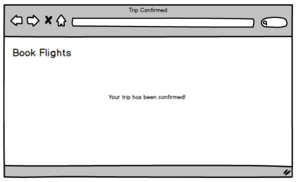

Travel Planet
Book Flights
Please confirm your flight:
I accept the General Terms and Conditions.
Withdraw Trip
Confirm Booking

Passenger: Jony Michel

Flight Airfrance AF2591
Monday, September 21st, 2017
Paris ORY (08:35) - London LGW (09:15)

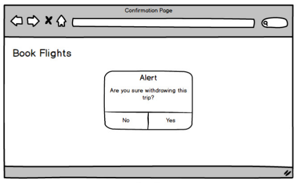
Price: 115.00 € Discount Fare Conditions

(e)



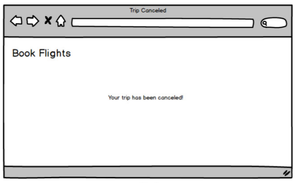
Your trip has been confirmed!

(f)



Are you sure withdrawing this trip?

(g)



Your trip has been canceled!

(h)

Fig. 8. Balsamiq prototypes reengineered for testing.

Table 2. Test results in Balsamiq prototypes.

Scenario: *Successful Roundtrip Tickets Search With Full Options*			
Battery	**Step**	**Balsamiq source file**	**Error**
1	1 - *Given I go to "**Book Flights**"* (**FAILED**, assertion error: expected <1> but was <0>).	Element: ***SubTitle*** Name: **Book Flights**	Expected "*BrowserWindow*", but the element was "*SubTitle*".
2	2 - *When I inform "Toulouse" and choose "Toulouse, Blagnac (TLS)" in the field "Departure"*		-
	3 - *And I inform "Paris" and choose "Paris, Charles-de-Gaulle (CDG)" in the field "Destination"*		-
	4 - *When I set "Sam, Déc 1, 2018" in the field "**Departure Date**"* (**FAILED**, assertion error: expected <1> but was <0>).	Element: ***Label***, Group: **0** Name: **Departure Date** Element: ***DateChooser***, Group: **22**	The label "Departure Date" and the element "*DataChooser*" are in different groups.
3	5 - *And I set "08:00" in the field "Departure Time Frame"*		-
	6 - *When I choose "Round Trip"*		-
	7 - *And I set "Lun, Déc 10, 2018" in the field "**Arrival Date**"* (**FAILED**, assertion error: expected <1> but was <0>).	Element: ***Label***, Group: **0** Name: **Arrival Date** Element: ***DateChooser***, Group: **23**	The label "Arrival Date" and the element "*DataChooser*" are in different groups.
4	8 - *When I set "10:00" in the field "Arrival Time Frame"*		-
	9 - *And I choose the option of value "2" in the field "**Number of Passengers**"* (**FAILED**, assertion error: expected <1> but was <0>).	?	The field "Number of Passengers" does not exist.
5	10 - *When I set "6" in the field "**Timeframe**"* (**FAILED**, assertion error: expected <1> but was <0>).	Element: ***ComboBox*** Name: **Time Frame**	Expected field "Timeframe" but was "Time Frame".
6	11 - *And I select "**Direct Flights Only**"* (**FAILED**, assertion error: expected <1> but was <0>).	Element: ***CheckBox*** Name: **Only direct flights**	Expected field "Direct Flights Only" but was "Only direct flights".
7	12 - *When I choose the option of value "Economique" in the field "**Flight Class**"* (**FAILED**, assertion error: expected <1> but was <0>).	Element: ***ComboBox*** Name: **Class**	Expected field "Flights Class" but was "Class".
8	13 - *And I set "Air France" in the field "**Companies**"* (**FAILED**, assertion error: expected <1> but was <**3**>).	Element: ***Label***, Group: **27** Name: **Companies** Element: ***SearchBox***, Group: **27** Element: ***SearchBox***, Group: **27** Element: ***SearchBox***, Group: **27**	Three elements "*SearchBox*" to address the same field "Companies".
9	14 - *When I submit "Search"*		-
	15 - *Then will be displayed "2. Sélectionner un voyage"* (**FAILED**, assertion error: expected <1> but was <0>).	-	Dynamic behavior between screens. Untraceable interaction.
Scenario: *Select a Return Flight Searched With Full Options*			
Battery	**Step**	**Balsamiq source file**	**Error**
1	16 - *Given "**Availability Page**" is displayed* (**FAILED**, assertion error: expected <1> but was <0>).	?	"Availability Page" does not exist.
2	17 - *When I click on "**No Bag**" referring to "Air France 7519"* (**FAILED**, assertion error: expected <1> but was <0>).	Element: ***DataGrid*** Name: Flight, **Discount**, Classic, Flex	Expected field "No Bag" but was "Discount".
3	18 - *And I click on "No Bag" referring to "Air France 7522"*		-
	19 - *When I click on "Book"*		-
	20 - *Then will be displayed "J'accepte les Conditions d'achat concernant le(s) tarif(s) aérien(s)."* (**FAILED**, assertion error: expected <1> but was <0>).	-	Dynamic behavior between screens. Untraceable interaction.

(*continued*)

Table 2. (*continued*)

Scenario: *Confirm a Flight Selection (Full Version)*			
Battery	**Step**	**Balsamiq source file**	**Error**
1	21 - *Given "Confirmation Page" is displayed* (**FAILED**, assertion error: expected <1> but was <0>).	?	"Confirmation Page" does not exist.
2	22 - *When I choose "J'accepte les Conditions d'achat concernant le(s) tarif(s) aérien(s)."*		-
	23 - *And I click on "**Finalize the trip**"* (**FAILED**, assertion error: expected <1> but was <0>).	Element: ***Button*** Name: **Confirm Booking**	Expected field "Finalize the trip" but was "Confirm Booking".
3	24 - *Then will be displayed "Votre voyage a été confirmé!"* (**FAILED**, assertion error: expected <1> but was <0>).	-	Dynamic behavior between screens. Untraceable interaction.

During the second battery of tests, the steps 1, 2 and 3 passed, and an error was found at the step 4 (*"When I set 'Sam, Déc 1, 2018' in the field 'Departure Date'"*). This error refers to the label "Departure Date" that has been found in a different group than the element "*DataChooser*" which was used to model it. As detailed in Sect. 3, Balsamiq implements UI elements either as independent instances (i.e. with the name and the interaction element defined in the same tag), or as part of a group (i.e. defining the name in the tag "label" and the interaction element itself in another tag). In the second case, the group must be modeled as a single unit, with a unique identifier. The label "Departure Date" was found in a given group and its interaction element "*DataChooser*" in another one, so they could not be recognized as a single unit. To fix the error, they were regrouped.

During the third battery of tests, the steps 4, 5 and 6 passed, and the same error was found at the step 7 ("And I set 'Lun, Déc 10, 2018' in the field 'Arrival Date'"). The label "Arrival Date" and its correspondent element "DataChooser" were found in different groups. The same solution to fix it was applied. During the fourth battery of tests, the steps 7 and 8 passed, and an error was found at the step 9 ("And I choose the option of value '2' in the field 'Number of Passengers'"). The field "Number of Passengers" was not found in the prototype. It was added to fix the error.

During the fifth battery of tests, the step 9 passed, and an error was found at the step 10 ("When I set '6' in the field 'Timeframe'"). The field "Timeframe" was named as "Time Frame". The field was renamed in the prototype to fix the inconsistency. The same occurred during the sixth and seventh battery of tests, respectively with the fields "Direct Flights Only" (step 11) and "Flight Class" (step 12). They were named as "Only direct flights" and "Class" respectively. They were also renamed, so the test passed.

During the eighth battery of tests, an error was found at the step 13 (*"And I set 'Air France' in the field 'Companies'"*). Three elements "*SearchBox*" were found to address the same field named only as "Companies". The solution was to identify uniquely each one of the fields "*SearchBox*", once each one of them is able to receive different values during the interaction. If we redesign the step to call specifically one of the fields (e.g. Company 1) the test passes, as we are interacting with just a unique and determined field. If otherwise we call the group Companies as a whole, we do not know with which field we should interact. The three fields were named respectively as "Company 1", "Company 2" and "Company 3", leaving the name "Companies" to reference only the group as a whole. Once again, both the scenario and the prototype have been updated.

During the ninth battery of tests, the steps 13 and 14 passed. For the step 15, at the end of the first scenario, the message referenced by the last step is supposed to be displayed in another screen as a result of the interaction. As stated in Sect. 3, tests on prototypes at this level of refinement do not consider the dynamic aspect of the interaction, so tests like this, involving navigation between screens, will always fail.

Following the booking process, the second scenario (*"Select a Return Flight Searched With Full Options"*) ran only 3 batteries of tests before getting a consistent prototype. The first battery found an error in the element "Availability Page" that is missing in the prototype. During the second battery, the field "No Bag" was named as "Discount" in the grid. Finally, the third battery fell in the case mentioned previously, which consists in checking a message that is supposed to be displayed in the next screen as a result of the interaction.

The third and last scenario to conclude the booking (*"Confirm a Flight Selection Full Version"*), also ran only 3 batteries of tests before getting a consistent prototype. The first one found the same error related to the name of the page. In the second one, the button "Finalize the trip" was named as "Confirm Booking", and the third and last battery felt in the case of dynamic behavior between screens. Notice that, for testing purposes, the message *"I accept the General Terms and Conditions"* in English was considered equivalent to the message *"J'accepte les Conditions d'achat concernant le (s) tarif(s) aérien(s)."* in French.

In our further test batteries with other scenarios, we got errors when testing steps such as *"And I choose 'One-way Trip'"* and *"When I choose 'Multidestination Trip'"* because these options do not exist in the UI prototypes for searching flights. In fact, the corresponding option was named *"One-way/Multidestination"* (Fig. 8, (a)). Here we get an important inconsistency related with the design options. During the specification of scenarios, we can notice that three options were planned to select the trip type: one-way, roundtrip, or multidestination. However, in this version of the prototype, it has been modeled only two options: one for choosing a roundtrip, and another for choosing a one-way/multidestination trip. This option has been made for the prototype because, in terms of interaction, the action required for providing data for multidestination flights is actually the same as the one for providing data for a set of one-way flights. In terms of user requirements, this is a conflicting specification, so such an inconsistency needs to be identified and fixed. Thus, either the prototype should follow what has been specified in the scenarios, or the scenarios should be fixed to comply with the interaction supported by the prototype.

In a scenario for describing a successful multi-destination ticket search, our algorithm has identified, as expected, three fields named "Departure" and "Destination" (see Fig. 8, (b)). For each element, the results returned the count of 3, when it was expected to be 1. When designing such a UI, as the three fields have been just replicated (copied and pasted) with the same name, with the purpose of illustrating the change on the UI when the "One-way/Multidestination" option is selected, the group to which such fields belonged has been maintained, so this set up the inconsistency. Otherwise, if the fields had the same name, but belonged to different groups, an inconsistency would not be signalized as it would indicate that the fields were intentionally modeled as different objects.

Finally, for the step *"And I set 'Sam, Déc 1, 2018' in the field 'Departure Date'"* in the same scenario, the field "Departure Date" was also replicated, but the pair of elements (labels and actual fields) has not been associated to a group, i.e. each element (label and field) has been found belonging to distinct groups in each instance of the field "Departure Date". The inconsistency was also detected and signalized.

5.3 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 Balsamiq prototypes as follows:

- Conflict between expected and actual elements
- Element and label in different groups
- Missing elements
- Element semantically inconsistent
- More than one element to represent the same field
- Untraceable interaction between screens

As presented above, we identified 6 different types in the tested scenarios. *"Conflict between expected and actual elements"* was the most frequent type and refers to elements that are specified with different names in the step and in the prototype. *"Missing elements"* and *"untraceable interaction between screens"* comes next and refer respectively to the real absence in the prototypes of elements that are specified in the step, and to the cases where the interaction changes the state of the interface (e.g. transitioning between screens or making appear a given value in a field). *"Untraceable interaction between screens"* is a particular type of inconsistency due to the level of refinement we are considering for prototypes. Balsamiq is a prototype tool that actually supports a basic dialog description, using links between prototypes and simulating a real navigation on user interfaces. However, we have chosen to not cover such a feature for now, since the ontology we use can already support more robust interactions in other levels of prototype refinements, such as the one that has been implemented by PANDA [13].

"Elements and labels in different groups" is the next type in line and refers to one of the mechanisms of modeling used by Balsamiq when there is an absence of group links between labels and the actual interaction element in the prototype. When a given UI element is composed by a label name and the interaction element itself, this encompassed structure is modeled by an entity named "group". Thus, to be considered as a unique and single element, both the label and the interaction element itself must be

placed at the same group. If it is not the case, we are not able to reach the element and then an inconsistency is detected. This inconsistency leads to a misidentification of elements in Balsamiq but could eventually not be an issue in other prototyping tools.

"More than one element to represent the same field" is a type of inconsistency caused when there are at least two elements (or more) in the prototype which are of the same type and are placed in the same group (or have the same name) of the searched field. Finally, the type of inconsistency named *"elements semantically inconsistent"* refers to the core problem we address with the ontology, i.e. the use of interaction elements in the prototype that are semantically inconsistent with the behavior they are supposed to model. This kind of inconsistency is detected when we get the list of supported interaction elements from the ontology and check if the interaction element used in the prototype is equivalent to one of them.

The types of inconsistencies identified by our approach are useful to provide information about the consistency between user requirements and GUI prototypes at any stage of a user-centered design process. Especially at the early stages, where user requirements still have a high level of uncertainty, the identification of these inconsistencies is an important resource to both designers and domain experts to validate a given understanding of the requirements and to ensure that the multiple proposed design options still remain consistent with the requirements. A conducive factor to that is our strategy based on a static analysis for implementing the assessment of GUI prototypes. When opting for a static analysis of Balsamiq source files, we gain in performance and availability of tests. Unlike approaches implementing co-execution, and specially in environments requiring a high-availability of tests to be executed continuously along multiple iterations, static approaches benefit from an instantaneous consistency checking by analyzing in seconds several hundreds of source files at the same time.

As limitations, we point out that this study has been conducted performing a manual reverse engineering of the existing system currently in production to obtain the respective prototypes for testing. Therefore, as a manual process, it was expected that inconsistencies would be naturally introduced during the modeling. Indeed, these inconsistencies were identified and that allowed us to evaluate our approach. Nonetheless, if an automated approach of reverse engineering had been used instead, such inconsistencies would probably not have taken place. Future studies should confirm this hypothesis. Since both the conduction of the study and the interpretation and analysis of the results have initially been made by the authors, a possible bias should be considered as a threat to validity. To mitigate that, the results were cross-checked by independent reviewers, experts in software engineering and modeling. They examined both the reengineered GUI prototypes and the testing results, then they performed a qualitative analysis of the types of inconsistencies identified. The results presented in this paper are thus a consolidated and revised version of the testing outcomes.

5.4 Conclusion

In this paper, we describe a novel approach for assessing low-fidelity wireframes and sketches developed by commercial prototyping tools like Balsamiq. Our approach has the main advantage of ensuring a reliable correspondence between the different interaction elements modeled in GUI prototypes and the user requirements specified by stakeholders. By using a supporting ontology, the approach provides automated testing for Balsamiq prototypes, implementing an open and flexible architecture which allows other GUI prototyping tools fitting in the future. For that, it is enough to implement a new core interface for describing the way such tools deal with their interaction elements and how they can be identified in their source files.

This approach has also been extended and adapted to assess other early artifacts such as task models, as well as late artifacts such as final UIs [36–40]. As an integrated approach, User Stories can also be assigned to automatically assess both task models, UI prototypes in different levels of abstraction, and final UIs, ensuring a consistent verification, validation and testing (VV&T) approach for interactive systems with high-availability of tests and 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 UI prototypes throughout a real software development process. Future studies should also explore the assessment of GUI prototypes using new interaction techniques, which has the potential to bring new challenges. Concerning the tools, the development of an Eclipse plugin to suggest and autocomplete steps of the User Stories scenarios based on the interactive behaviors of the ontology is also envisioned. It would allow experts and other stakeholders to directly create their own User Stories by following the proposed vocabulary.

References

1. Wood, D.P., Kang, K.C.: A Classification and Bibliography of Software Prototyping. Pittsburgh, Pennsylvania (1992)
2. 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, New York (2010)
3. Cohn, M.: User Stories Applied for Agile Software Development. Addison-Wesley, Boston (2004)
4. 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
5. 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

6. Jeffries, R., Miller, J.R., Wharton, C., Uyeda, K.: User interface evaluation in the real world: a comparison of four techniques. In: Proceedings of CHI 1991 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 119–124 (1991). https://doi.org/10.1145/108844.108862
7. Tian, J.: Software inspection. In: Jeff, T. (ed.) Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement, pp. 237–250. John Wiley & Sons, Inc., New York (2005). https://doi.org/10.1002/0471722324.ch14
8. Ebert, C.: Global Software and IT: A Guide to Distributed Development, Projects, and Outsourcing. Wiley, Hoboken (2011)
9. Luna, E.R., Garrigós, I., Grigera, J., Winckler, M.: Capture and evolution of web requirements using webspec. In: Benatallah, B., Casati, F., Kappel, G., Rossi, G. (eds.) ICWE 2010. LNCS, vol. 6189, pp. 173–188. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13911-6_12
10. Buchmann, R.A., Karagiannis, D.: Modelling mobile app requirements for semantic traceability. Requir. Eng. **22**(1), 41–75 (2017). https://doi.org/10.1007/s00766-015-0235-1
11. Winckler, M., Palanque, P.: StateWebCharts: a formal description technique dedicated to navigation modelling of web applications. In: Jorge, J.A., Jardim Nunes, N., Falcão e Cunha, J. (eds.) DSV-IS 2003. LNCS, vol. 2844, pp. 61–76. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39929-2_5
12. Barnett, J.: State Chart XML (SCXML): State Machine Notation for Control Abstraction. W3C (2017). http://www.w3.org/TR/scxml/
13. Hak, J., Winckler, M., Navarre, D.: PANDA: prototyping using annotation and decision analysis. In: Proceedings of the 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, pp. 171–176 (2016). https://doi.org/10.1145/2933242.2935873
14. Elkoutbi, M., Khriss, I., Keller, R.K.: Generating user interface prototypes from scenarios. In: Proceedings of the IEEE International Symposium on Requirements Engineering (Cat. No. PR00188), pp. 150–158 (1999). https://doi.org/10.1109/ISRE.1999.777995
15. Han, L., Yang, J., Zhao, W., Sheng, Q.Z.: User interface derivation for business processes. IEEE Trans. Knowl. Data Eng. (2019). https://doi.org/10.1109/TKDE.2019.2891655
16. Schlungbaum, E., Elwert, T.: Automatic user interface generation from declarative models. Comput. Aided Des. User Interfaces (CADUI) **5**, 3–18 (1996)
17. Wolff, A., Forbrig, P., Dittmar, A., Reichart, D.: Linking GUI elements to tasks – supporting an evolutionary design process. In: Proceedings of the 4th International Workshop on Task Models and Diagrams, pp. 27–34 (2005). https://doi.org/10.1145/1122935.1122941
18. Valente, P., Silva, T.R., Winckler, M., Nunes, N.J.: The goals approach: enterprise model-driven agile human-centered software engineering. In: Bogdan, C., et al. (eds.) HCSE/HESSD-2016. LNCS, vol. 9856, pp. 261–280. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44902-9_17
19. Valente, P., Silva, T., Winckler, M., Nunes, N.: The goals approach: agile enterprise driven software development. In: Gołuchowski, J., Pańkowska, M., Linger, H., Barry, C., Lang, M., Schneider, C. (eds.) Complexity in Information Systems Development. LNISO, vol. 22, pp. 201–219. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-52593-8_13
20. Lucassen, G., Dalpiaz, F., Van Der Werf, J.M.E.M., Brinkkemper, S., Zowghi, D.: Behavior-driven requirements traceability via automated acceptance tests. In: Proceedings - 2017 IEEE 25th International Requirements Engineering Conference Workshops, REW 2017, pp. 431–434 (2017). https://doi.org/10.1109/REW.2017.84

21. de Carvalho, R.A., Manhães, R.S., de Carvalho e Silva, F.L.: Filling the gap between business process modeling and behavior driven development (2010). arXiv: https://arxiv.org/abs/1005.4975
22. de Carvalho, R.A., de Carvalho e Silva, F.L., Manhaes, R.S.: Mapping business process modeling constructs to behavior driven development ubiquitous language (2010). arXiv: https://arxiv.org/abs/1006.4892
23. Lübke, D., Van Lessen, T.: Modeling test cases in BPMN for behavior- driven development. IEEE Softw. **33**, 15–21 (2016). https://doi.org/10.1109/MS.2016.117
24. Khaddam, I., Mezhoudi, N., Vanderdonckt, J.: Towards task-based linguistic modeling for designing GUIs. In: 27th Conference on l'Interaction Homme-Machine (2015). https://doi.org/10.1145/2820619.2820636
25. Palanque, P., Martinie, C., Winckler, M.: Designing and assessing interactive systems using task models. In: Bernhaupt, R., Dalvi, G., Joshi, A., KB, D., O'Neill, J., Winckler, M. (eds.) INTERACT 2017. LNCS, vol. 10516, pp. 383–386. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68059-0_35
26. Campos, J.C., Fayollas, C., Martinie, C., Navarre, D., Palanque, P., Pinto, M.: Systematic automation of scenario-based testing of user interfaces. In: Proceedings of the 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems - EICS 2016, pp. 138–148 (2016). https://doi.org/10.1145/2933242.2948735
27. Silva, T.R., Hak, J.-L., Winckler, M., Nicolas, O.: A comparative study of milestones for featuring GUI prototyping tools. J. Softw. Eng. Appl. **10**(06), 564–589 (2017). https://doi.org/10.4236/jsea.2017.106031
28. Silva, T.R., Hak, J.-L., Winckler, M.A.: A review of milestones in the history of GUI prototyping tools. In: IFIP TC.13 International Conference on Human-Computer Interaction – INTERACT 2015 Adjunct Proceedings, pp. 267–279 (2015)
29. Beck, K.: Test Driven Development: By Example, 1st edn. Addison-Wesley Professional, Boston (2002)
30. Astels, D.: Test-Driven Development: A Practical Guide, 1st edn. Prentice Hall, Upper Saddle River (2003)
31. North, D.: What's in a Story? (2019). https://dannorth.net/whats-in-a-story/. Accessed 01 Jan 2019
32. Gherkin. Gherkin Reference. https://cucumber.io/docs/gherkin/reference/
33. Beaudouin-Lafon, M., Mackay, W.E.: Prototyping tools and techniques. In: Prototype Development and Tools, pp. 1–41 (2000)
34. 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. Cognit. Technol. Work, 1–21 (2019). https://doi.org/10.1007/s10111-019-00566-3
35. Chikofsky, E.J., Cross II, J.H.: Reverse engineering and design recovery: a taxonomy. IEEE Softw. **7**, 13–17 (1990). https://doi.org/10.1109/52.43044
36. 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)
37. 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
38. 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., et al. (eds.) HCSE/HESSD -2016. LNCS, vol. 9856, pp. 86–107. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44902-9_7

39. 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
40. 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