Towards the Generation of End-to-End Web Test Scripts from Requirements Specifications

Diego Clerissi, Maurizio Leotta, Gianna Reggio, Filippo Ricca
Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi (DIBRIS), Università di Genova, Italy
diego.clerissi@dibris.unige.it, maurizio.leotta@unige.it, gianna.reggio@unige.it, filippo.ricca@unige.it

***Abstract*—Web applications pervade our life, being crucial for a multitude of economic, social and educational activities. For this reason, their quality has become a top-priority problem. End-to-End testing aims at improving the quality of a web application by exercising it as a whole, and by adopting its Requirements Specification as a reference for the expected behaviour. In this paper, we outline a novel approach aimed at generating test scripts for web applications from either Textual or UML-based Requirements Specifications. A set of automated transformations are employed to keep Textual and UML-based Requirements Specifications synchronized and, more importantly, to generate End-to-End test scripts from UML artefacts.**

***Index Terms*—Web Testing, Requirements Specification, Use Case, UML**

I. Introduction

In the last years, web-based software has become the key asset in a multitude of everyday activities. For this reason, effective testing approaches aimed at increasing the quality of web applications are of fundamental importance [13]. End-to-End testing is a relevant approach for improving the quality of complex web systems [7]: web applications are exercised as a whole, testing the full-stack of technologies implementing them. It is a type of black box testing based on the concept of test scenario, i.e. a sequence of steps/actions performed on the web application (e.g. insert username and password, click the login button, etc.).

A Requirements Specification expressed as use cases can be employed as a reference for the correct behaviour of a web application, and can be used to derive the test cases. Screen mockups are additional artefacts used in conjunction with use cases to represent the interface of a web application before/after the execution of each scenario step; they can improve the comprehension of functional requirements [11], and can also be used for the non-functional ones [10]. Moreover, the introduction of a glossary, to precisely describe the terminology referred by use cases, could enforce the Requirements Specification understandability, reducing also ambiguities which may originate from unclear or complex sentences. A method providing well-formedness constraints over such entities would thus produce a precise and of high quality Requirements Specification [9], also in a highly dynamic context as the Web. Indeed, use cases plus screen mockups naturally describe how a web application should be tested in terms of its behaviour, as perceived by the users, and the glossary may clarify the data used by test cases, as well as the performed instructions.

Despite their wide adoption for describing requirements, textual use cases and, more generally, natural language processing techniques, cannot directly support automated test cases generation [4]; instead, different notations (e.g. UML) may provide a more structured and formal view, exploitable by existing tools. For example, state machines integrated with screen mockups can intuitively represent the system behaviour in terms of navigational paths as basis for generating End-to-End test cases.

In this paper, we present a novel approach for generating test scripts (i.e. executable test cases) for web applications from a *precise* Requirements Specification, either Textual or UML-based. A precise Requirements Specification satisfies a set of well-formedness constraints aimed at improving the overall quality and making the Specification suitable for test scripts generation. The analyst may choose the perspective (s)he is more confident with (i.e. Textual or UML) and, by means of an automated transformation, derive the other one with a little effort. An additional automated transformation is applied on the UML artefacts to generate End-to-End test scripts.

Even though the generation of UML artefacts from use cases and consequent test cases extraction has been already investigated (for example, by Yue *et al* [14]), to the best of our knowledge our approach is the first one trying to generate End-to-End test scripts for web applications completely aligned with their Requirements Specifications, where Textual and UML-based Specifications are automatically synchronized, and screen mockups are integrated in the process.

The proposed approach is intended to be supported by a prototype tool, to assist the final user in the definition of the Requirements Specification, and in the automated artefacts generation (e.g. UML diagrams, test scripts). The tool will provide a user-friendly and step-by-step interface, where manual intervention is reduced as much as possible.

Section II provides an overview of the approach, Sections III and IV describe the Textual and the UML-based Requirements Specifications respectively, while the transformations between the Specifications and to the Testware are discussed in Sections V-VI. Related works are shown in Section VII, and finally conclusions and future work are given in Section VIII.

II. The Approach

The aim of our approach is to generate test scripts for a web application given its Requirements Specification as input.

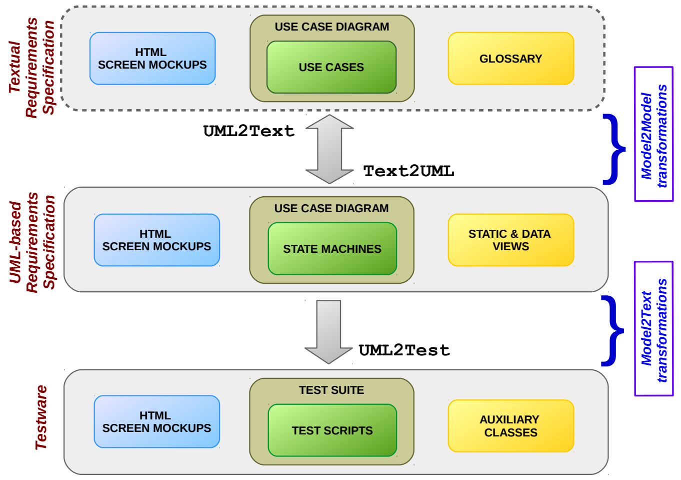


Fig. 1. The proposed approach: from Requirements Specifications (Textual or UML-based) to Testware.

Fig. 1 provides an overview of the approach, which is based on three main automated transformations.

The *Textual Requirements Specification* is usually the starting point of the approach, and is basically characterized by a use case diagram, Textual use cases, HTML screen mockups, and a glossary introducing the used terminology. Use cases are adopted since they naturally represent the systems behaviour (including web applications) as structured scenarios. Screen mockups are embodied in use cases steps to enhance the overall comprehension of the requirements [11], and to support the ensuing automated test generation and execution [2]; in fact, they visually describe how the web application GUI should appear, and how it should react to users interactions [9]. The *UML-based Requirements Specification* is instead characterized by a use case diagram, use cases given in the form of state machines, with attached screen mockups, and class diagrams defining the used data and the interactions among the actors and the web application (i.e. static and data views in Fig. 1). Finally, the *Testware* is the output of the approach; it includes the test suite, grouping the automatically generated End-to-End test scripts that cover all interesting aspects described in the Requirements Specification, and all the code (i.e. the auxiliary classes in Fig. 1) representing the operations over the data and the occurring interactions. In our proposal, both kinds of Specifications and the Testware are defined by means of a metamodel accompanied by a set of well-formedness constraints [1].

Our approach allows to skip a Textual formulation of the requirements (that is the reason of the surrounding dashed line in Fig. 1, indicating optionality), starting directly from UML, from which a Textual counterpart can be automatically derived. Having two different perspectives gives more freedom to the analyst, who may alternatively choose a simpler Textual solution to be transformed into UML models or directly adopt UML in case of high professional skills. However, the Testware is generated from UML only, as shown in Fig. 1, since UML represents use cases in a more structured and formal way.

The **`Text2UML`** and **`UML2Text`** transformations aim at moving between the Textual and the UML-based Requirements Specifications. More specifically, **`Text2UML`** transforms use cases into state machines, and the glossary into the static and the data views. Instead, **`UML2Text`** generates use cases from state machines, including a glossary retrieved from the aforementioned views. Notice that HTML screen mockups are untouched by the transformations; in fact, they are complementary elements in both kinds of Requirements Specifications. **`Text2UML`** and **`UML2Text`** are *Model2Model* transformations, since they rely on metamodels defining the form of the Textual and the UML-based Requirements Specifications. Finally, **`UML2Test`** is a *Model2Text* transformation generating the code of the Testware from the UML artefacts.

The approach is applicable from scratch, using Test Driven Development (TDD) to drive the development of a novel application, as well as to test already existing web applications, having at hand their Requirements Specifications. If screen mockups are adopted in TDD, they could be the basis for web pages development, since they are functionally complete to be exercised by the test scripts [2]. This is the main reason of including screen mockups in the Testware, as shown in Fig. 1.

From now on, we use the term WebApp to denote a generic web application we want to test after having its requirements specified. Our running example is PhoneBook, a simple web application storing phone contacts info. The complete PhoneBook Textual and UML-based Requirements Specifications can be found in [1]. Notice that most of the steps needed to produce a precise Textual or UML-based Requirements Specification are intended to be tool-supported, with the goal to reduce the manual activities as much as possible.

III. Textual Requirements Specification

A Textual Requirements Specification consists of an UML use case diagram summarizing the use cases, a glossary that lists and makes precise all the terms used in the use cases, a description of each use case, and a set of HTML screen mockups associated with use cases steps. A Textual Requirements Specification is precisely defined by a metamodel (see Fig. 2[1] accompanied by a set of well-formedness constraints [1].

A. Use Case Diagram

The *use case diagram* summarizes the WebApp use cases, making clear the actors (i.e. the users of the WebApp) taking part in them, and their mutual relationships. It is actually an UML diagram, but quite simple to understand and to produce, and useful to summarize the use cases, so there is no need of a more detailed presentation.

B. Glossary

The *glossary* is a list of entries, introducing all the terms appearing in the use cases, each one consisting of a name, and of a definition. A portion of the PhoneBook glossary is shown in Fig. 3.
The glossary entries are distinguished in:

[1] Omitted multiplicities in associations are intended to be 1.

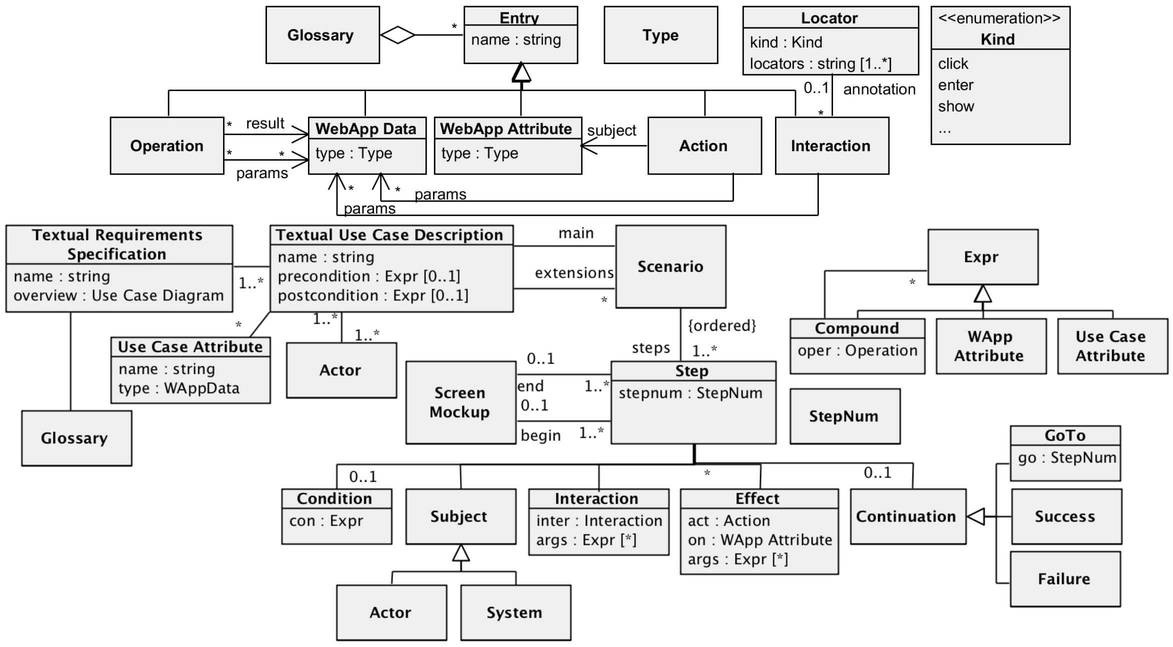


Fig. 2. Textual Requirements Specification Metamodel.

WebApp Data
Name is a string
Phone is a string
Group is a string
Entry is a Name X Phone X Group
Username is a string
...
WebApp Attributes
LoggedUser has type Username
RegisteredEntries has type sequence(Entry)
...

Operations
Name is well-formed returns bool:
size of N is less than 32
exists entry for Name returns bool:
<N, -, - > is in RegisteredEntries
...
Actions
Entry is added to RegisteredEntries:
RegisteredEntries =
RegisteredEntries adds E
...
Interactions
requests to add new entry
[click: "addNewEntry"]
enters entry details Name and Phone
[enter: "name", "phone"]

Fig. 3. A portion of PhoneBook glossary.

1) WebApp Data: the types of the data mentioned in the use cases. They have form "name is a *type*", where *type* is either a basic type (e.g. string, int, bool), a Cartesian product, or a sequence of types. For example, in Fig. 3 we have "Name is a string", and "Entry is a Name X Phone X Group".

2) WebApp Attributes: the properties abstractly describing the updatable state of the WebApp. They have form "name has type *Data*", where *Data* is a (sequence of a) previously defined *WebApp Data*. For example, in Fig. 3 we have "LoggedUser has type Username" and "RegisteredEntries has type sequence(Entry)".

3) Operations: the functions performed over data and attributes to get/check their content. They have form "namepart_1 Data_1 $\ldots\text{namepart}_n$ Data_n returns Data", where each Data is a previously defined *WebApp Data*. The semantics of the operations is given in a structured textual form. For example, in Fig. 3 we have "Name is well formed", which checks that the size of a Name is less than 32, where "size of", "is less than" and "32" are predefined functions over strings and integers.

4) Actions: the updates of the WebApp state appearing in the use cases steps. They have form "Data_1 namepart_1 $\ldots\text{Data}_n$ namepart_n webAttribute", where each Data and webAttribute have been previously defined. The semantics of the actions is given similarly to operations. For example, in Fig. 3 we have "Entry is added to RegisteredEntries".

5) Interactions: the atomic interactions between the actors and the WebApp and vice versa. They have form "namepart_1 Data_1 $\ldots\text{namepart}_n$ Data_n", where each Data has been previously defined in the *WebApp Data* part of the glossary. For example, in Fig. 3 we have requests to add new entry, and enters entry details Name and Phone. The parts enclosed by square brackets in Fig. 3 are explained in Section III-D.

C. Use Cases Descriptions

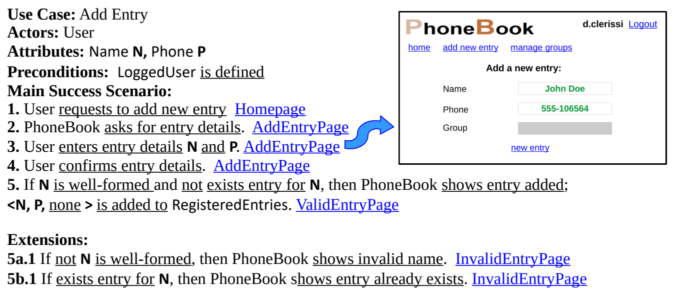
In our proposal, use cases follow a slight adjustment of Cockburn's template [3]. Each use case is described by some info, plus a set of scenarios (see an example of a PhoneBook use case in Fig. 4).

The *use case attributes* are the data needed to describe the use case, each one is characterized by a name and typed by a WebApp data introduced in the glossary. In Fig. 4, two attributes are declared: Name N and Phone P, where Name and Phone are WebApp data defined in the glossary (see Fig. 3).

The *pre/post conditions* state what we assume about the current state of the WebApp before/after the (successful) execution of the use case (optional). They are expressions built using the operations of the glossary, the WebApp attributes (i.e. the current state of WebApp), and the use case attributes. In Fig. 4, a precondition concerning the authentication of the user is introduced.
The *main success scenario* describes the basic execution of the use case, whereas the *extensions* (any number, also none) define

all the other possible executions of the use case. Scenarios are sequences of uniquely numbered and ordered steps, each one structured as the following, where square brackets indicate optionality:

[if *Condition*, then] *Subject Interaction*; *Effect**. [*Continuation*.] The *Condition* determines the step executability and is formulated as the previously described Pre/Post conditions; for example, in Fig. 4 a condition is associated with step 5 and checks if the entry is not already present in the registered entries and if the name is well-formed (the definitions for these operations are given in the glossary, see Fig. 3). The *Subject* of a step is either an actor or the WebApp, while the *Interaction* is a sentence describing either what flows from the actor towards the WebApp or vice versa; interactions must be formulated by using those listed in the glossary, like the underlined sentences in the use case steps of Fig. 4. The *Effects* of a step are sentences written by using the actions listed in the glossary describing how the WebApp state changes depending on the *Interaction*; for example, at the end of step 5 in Fig. 4, a new entry is added to the registered ones. Finally, the *Continuation* defines how the use case flow continues after the end of the step; it may be a jump to a step different from the following one or a sentence declaring the success or the failure of the use case execution. If there is no *Continuation*, it means that the flow continues to the following step.



Use Case: Add Entry
Actors: User
Attributes: Name **N,** Phone **P**
Preconditions: LoggedUser is defined
Main Success Scenario:
1. User requests to add new entry Homepage
2. PhoneBook asks for entry details. AddEntryPage
3. User enters entry details **N** and **P**. AddEntryPage
4. User confirms entry details. AddEntryPage
5. If **N** is well-formed and not exists entry for **N**, then PhoneBook shows entry added;
<N, P, none **>** is added to RegisteredEntries. ValidEntryPage

Extensions:
5a.1 If not **N** is well-formed, then PhoneBook shows invalid name. InvalidEntryPage
5b.1 If exists entry for **N**, then PhoneBook shows entry already exists. InvalidEntryPage

PhoneBook d.clerissi Logout
home add new entry manage groups
Add a new entry:
Name John Doe
Phone 555-106564
Group
new entry

Fig. 4. PhoneBook Add Entry use case.

D. Screen Mockups

The *screen mockups* are GUI sketches representing accurately - from a functional point of view - the interfaces of a WebApp. Our approach requires to produce the screen mockups using the HTML, since it is the most convenient way to describe a WebApp GUI in terms of interactive web elements. For each use case step, a begin and an end screen mockup can be linked as placeholders [9] (i.e. hyperlinks to the corresponding files) to represent how the GUI looks before and after the step execution. At least one mockup is needed for each step, since a step describes the interactions performed over the web elements. For example, in Fig. 4, a screen mockup is linked at the end of each step.

Any screen mockup associated with a step must be consistent with it, i.e. it should present the same informative content, otherwise the introduction of the mockup would be the cause of further ambiguities in the Requirements Specifications, instead of improving their quality [9]. The consistency between

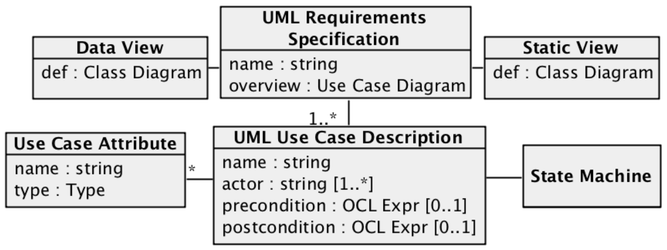


Fig. 5. UML-based Requirements Specification Metamodel.

mockups and use cases steps is granted by a set of well-formed constraints proposed by our method [1] to be followed while creating the mockups and writing the steps [9]. An example of a simple screen mockup associated with a use case step is shown in Fig. 4; the AddEntryPage mockup is linked to step 3 and contains all the web elements and all entered data used to fill a form.

The web elements of the screen mockups affected by the interactions in use cases steps must be made explicit. This is achieved by annotating the interactions in the glossary with the locators[2] of the involved web elements. Annotations have form [kind: locator$_1$, . . . , locator$_n$], where kind represents the kind of interaction performed over the web element(s), each one identified by a locator (see the Locator class in Fig. 2). Different types of locators exist, e.g. identifier, class name, or link text; in this work, for the sake of simplicity, locators are limited to identifiers. In Fig. 3, the interaction enters entry details Name and Phone used in Fig. 4 is annotated by [enter: "name", "phone"], stating which web elements of AddEntryPage mockup allow to perform it (i.e. the textfields localized by "name" and "phone").

IV. UML-based Requirements Specifications

An UML-based Requirements Specification has a similar structure of a Textual one, even though the parts are expressed using the UML constructs instead of plain text. Then, it consists of a use case diagram, a description of each use case, given by a state machine with associated screen mockups, plus data and static views (both class diagrams) defining the data and the interactions between the actors and the WebApp. An UML-based Requirements Specification is again precisely defined by a metamodel (see Fig. 5) and a set of constraints [1].

In the following, the use case diagram is omitted, since it is already discussed in Section III-A.

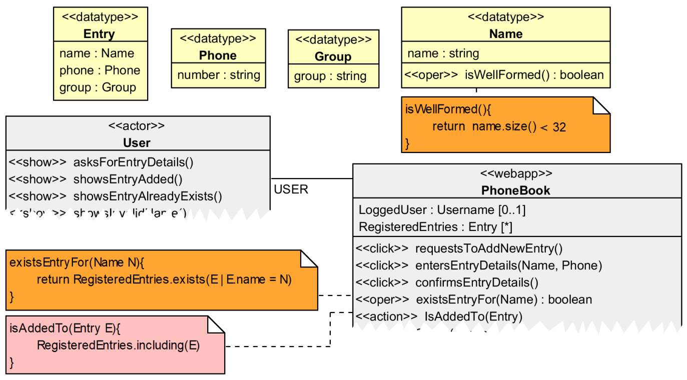
A. Data and Static Views

The *data view* is a class diagram containing the UML datatypes defining the data needed to express the requirements; it is basically equivalent to the definitions of WebApp data and operations in the glossary part of the Textual Requirements Specification. Any operation over a given data must be added to the corresponding UML datatype and stereotyped by ≪oper≫. The definitions for the operations are given in attached notes

[2] a locator is a hook pointing to a specific web element inside the DOM of an HTML page; it is used to retrieve the web elements the test script interacts with (e.g. find the link that must be clicked) [7]

and expressed using UML action language[3]; in the UML terminology, the definitions in the notes are the methods associated to such operations. See on top of Fig. 6 for a portion of PhoneBook data view. For example, Name includes an operation named isWellFormed, which checks whether it has length less than 32.

The *static view* is a class diagram containing the UML classes modelling the interactions between the actors and the WebApp, using all the data introduced by the data view; it is basically equivalent to the definitions of WebApp attributes, actions and interactions in the glossary part of the Textual Requirements Specification. The class modelling the WebApp must be stereotyped by ≪webapp≫, while those modelling the actors must be stereotyped by ≪actor≫. Each actor class must be connected to the WebApp class by an association, named as the actor itself. See on the bottom of Fig. 6 a portion of PhoneBook static view, including the WebApp and the actor classes. The attributes of the WebApp class model its updatable state (e.g. RegisteredEntries in Fig. 6); any operation applied over them is added as an UML operation to the WebApp class and stereotyped by ≪oper≫, as well as the actions performed over such attributes, which are stereotyped by ≪action≫. Definitions for operations and actions are given in notes attached to the WebApp class using the UML action language; for example, the pink note in Fig. 6 defines isAddedTo action, which takes an Entry E and adds it to RegisteredEntries. Finally, the interactions of the actors towards the WebApp are modelled by operations of the WebApp class, while those performed by the WebApp towards an actor are modelled by operations of the actor class; their stereotypes, such as ≪show≫ and ≪click≫ in Fig. 6, are explained in Section IV-C.


<<datatype>>
Entry
name : Name
phone : Phone
group : Group
<<datatype>>
Phone
number : string
<<datatype>>
Group
group : string
<<datatype>>
Name
name : string
<<oper>> isWellFormed() : boolean
isWellFormed(){
return name.size() < 32
}
<<actor>>
User
<<show>> asksForEntryDetails()
<<show>> showsEntryAdded()
<<show>> showsEntryAlreadyExists()
USER
<<webapp>>
PhoneBook
LoggedUser : Username [0..1]
RegisteredEntries : Entry [*]
<<click>> requestsToAddNewEntry()
<<click>> entersEntryDetails(Name, Phone)
<<click>> confirmsEntryDetails()
<<oper>> existsEntryFor(Name) : boolean
<<action>> IsAddedTo(Entry)
existsEntryFor(Name N){
return RegisteredEntries.exists(E | E.name = N)
}
isAddedTo(Entry E){
RegisteredEntries.including(E)
}
Fig. 6. A portion of PhoneBook data and static views.

B. Use Cases Descriptions

In the UML perspective, the description of a use case includes some info analogous to those of the Textual use cases, but the behaviour of the WebApp is here described by a state machine instead of a set of scenarios.

The *use case attributes* are declared in a note stereotyped by ≪attributes≫, and have form "name : *type*", where the *type* is a datatype defined in the data view (see Fig. 7).

[3] http://www.omg.org/spec/ALF/1.0.1/PDF/

The *pre/post conditions* are OCL expressions put in notes attached to the starting/ending states of the state machine. In Fig. 7, the precondition attached to the starting state refers to isLoggedUserDefined operation of the WebApp class (see Fig. 6) and is equivalent to the one given in Fig. 4.

The transitions of the state machine representing the behavioural aspect of the use case have one of the following forms:

- *Interaction* [*Condition*] / *Effect**, if the transition corresponds to an interactions from an actor towards the WebApp. *Interaction* is a call to an interaction of the WebApp class, *Condition* is a boolean OCL expression, and *Effect** are UML actions changing the WebApp state. In Fig. 7, the third transition is a call of the entersEntryDetails interaction of the WebApp, which uses the declared use case attributes to add a new entry.
- [*Condition*] / ACTOR.*Interaction* ; *Effect**, if the transition corresponds to an interactions from the WebApp towards an actor. *Interaction* is a call to an interaction of the ACTOR class, while the other parts are the same as before. In Fig. 7, the last transition on the left includes: a condition calling some operations over the entered Name N, *showsEntryAdded* interaction of the actor class, and the effect of updating the current WebApp state with the entered data, by calling the proper WebApp action.

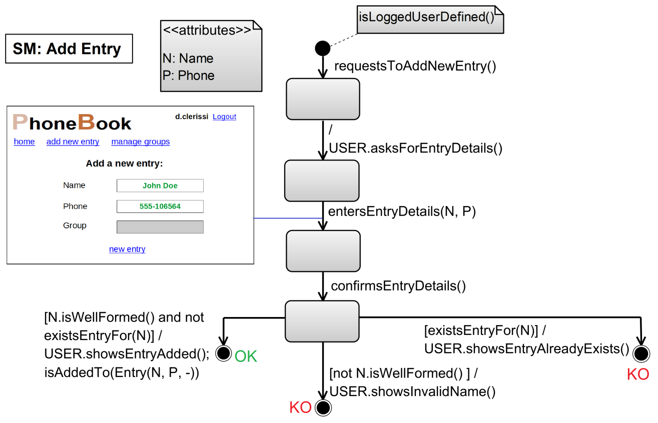


Fig. 7. PhoneBook Add Entry state machine.

C. Screen Mockups

In the UML perspective, screen mockups are associated with the transitions and the states of the state machines modelling the use cases behaviours. More specifically, if the transition corresponds to an interaction of the WebApp towards an actor, a single screen mockup is linked to the transition ending state. Instead, if the transition corresponds to an interaction of an actor towards the WebApp, then at most two mockups can be linked: one to the transition starting state, and one to the transition arrow head. At least one mockup is needed for each transition. An example of a screen mockup attached to a transition arrow head is given in Fig. 7.

As for use cases, screen mockups must be consistent with the transitions they are linked to [9]. Moreover, the web elements of the screen mockups that are affected by the interactions in the state machines (e.g. entersEntryDetails from Fig. 7) must be explicitly referred in the static view, where such interactions are defined. In the UML perspective, this connection is achieved by employing stereotypes and tagged values. A stereotype identifies the kind of interaction performed, while the tagged value encapsulates strings (thus, even multiple values) representing the locators of the web elements. The form adopted by tagged values for an interaction is {locators = "locator$_1$", . . . , "locator$_n$"}. For example, entersEntryDetails of the WebApp class is stereotyped by ≪enter≫, and associated with the tagged value {locators = "name", "phone"}, indicating that the data about name and phone will be entered in the text fields identified by "name" and "phone" strings.

V. Transformations between Textual and UML-based Requirements Specifications

In our proposal, the transformations are initially described from an high level perspective, and then refined in details by a decomposition stage where additional sub-transformations are involved.

Each (sub-)transformation shows how a source entity (e.g. a use case step) is transformed into a target entity (e.g. a state machine transition). The procedure of abstractly describing transformations from a source to a target universe has been inspired by Tiso *et al.* [12].

A (sub-)transformation is characterized by a name, an informal description in natural language declaring its goal, and a graphical representation of how source entities are transformed into target ones, including additional calls to further sub-transformations, in case a decomposition stage is needed.

The complete set of (sub-)transformations between Specifications can be found in [1].

A. Text2UML and UML2Text

Text2UML and **UML2Text** (shown as the bi-directional grey arrow in Fig. 1) are the Model2Model transformations applied between Textual and UML-based Requirements Specifications, having the dual goal to automatically generate one from the other. For space reasons, we limited the description of **UML2Text** to a sketched idea and we omitted **Text2UML**, since is basically the inverse of the former.

UML2Text transforms an UML-based Requirements Specification into a Textual one and is composed of several sub-transformations, handling the different parts composing it. The main transformation is sketched on top of Fig. 8: on the left, the source UML-based Requirements Specification; on the right, the target Textual Requirements Specification, where each part is generated by sub-transformations calls.

More specifically, **T-UCD** transforms an UML-based use case description into a Textual one, while **WA-Data**, **WA-Attributes** and **Interactions** respectively transform the information of the data and static views into the glossary entries (i.e. WebApp data and attributes with associated actions, operations, and interactions). The use case diagram

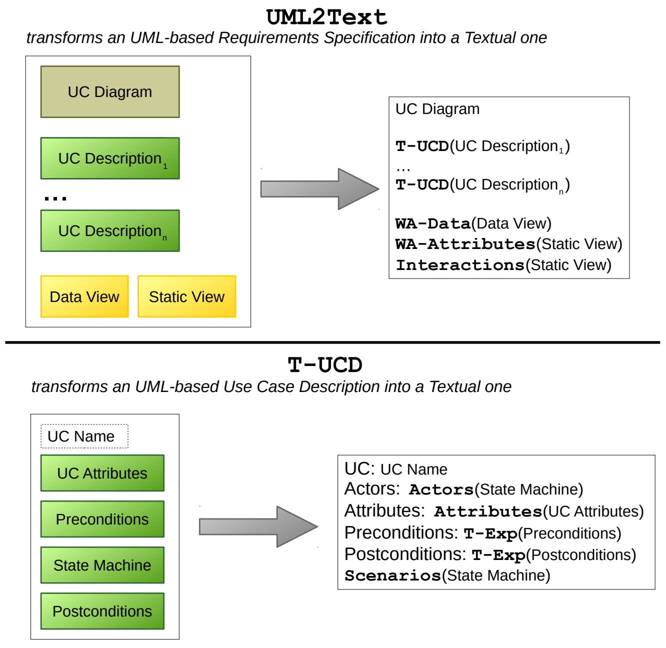

UML2Text
transforms an UML-based Requirements Specification into a Textual one
UC Diagram
UC Description$_1$
...
UC Description$_n$
Data View
Static View
UC Diagram
T-UCD(UC Description$_1$)
...
T-UCD(UC Description$_n$)
WA-Data(Data View)
WA-Attributes(Static View)
Interactions(Static View)
T-UCD
transforms an UML-based Use Case Description into a Textual one
UC Name
UC Attributes
Preconditions
State Machine
Postconditions
UC: UC Name
Actors: Actors(State Machine)
Attributes: Attributes(UC Attributes)
Preconditions: T-Exp(Preconditions)
Postconditions: T-Exp(Postconditions)
Scenarios(State Machine)

Fig. 8. (Above) **UML2Text**: from an UML-based to a Textual Requirements Specification. (Below) **T-UCD**: from an UML-based to a Textual use case description.

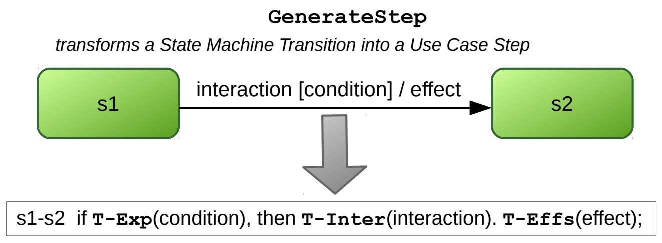

GenerateStep
transforms a State Machine Transition into a Use Case Step
s1
interaction [condition] / effect
s2
s1-s2 if T-Exp(condition), then T-Inter(interaction). T-Effs(effect);

Fig. 9. **GenerateStep**: from a state machine transition to a use case step (an interaction from an actor towards the WebApp).

is instead kept unaltered by the process. Again, further sub-transformations compose **T-UCD** (Fig. 8, below), generating the actors, the use case attributes, the pre/post conditions, and the scenarios respectively. **Scenarios** takes a state machine in input and generates the main and the alternative scenarios from its paths, each one characterized by a sequence of transitions from the starting to an ending state. The states of a state machine having multiple leaving transitions are the extensions points determining the various scenarios in the corresponding use case description; e.g. the last state in the state machine in Fig. 7 is the extension point for **5**, **5a.1**, and **5b.1** use case steps in Fig. 4. Among the activities we omitted, **Scenarios** includes **GenerateStep**, sketched in Fig. 9, which transforms a transition into a step; **T-Exp**, **T-Inter**, and **T-Effs** handle with conditions, interactions, and effects respectively. As shown in Fig. 1, screen mockups are not affected by the process; however, since they are part of the Requirements Specifications, transformations will attach them to the corresponding step (in the Textual perspective) or transition/state (in the UML perspective).

UML2Text and **Text2UML** will be implemented using the ATL language of the Eclipse Modeling Project[4], by now the standard for model to model transformations and also highly supported, well-documented, and integrated in Eclipse IDE.

VI. Transformation from an UML-based Requirements Specification to a Testware

In our proposal, the Testware is generated from the UML models, and is characterized by:

- *Test Scripts*: composed of instructions coding the transitions of state machines paths.
- *Test Suite*: the collection of all the *Test Scripts* and the general settings needed for their execution.
- *Auxiliary Classes*: all the code corresponding to the entities defined in the static and in the data views, hence representing the data and the instructions used by the *Test Scripts*.
- *Screen Mockups*: the HTML pages describing the WebApp GUI over which the *Test Scripts* instructions are performed.

For the aim of this paper, we decided to code the Testware components in Java, relying on the state-of-the-practice Selenium WebDriver testing framework[5], which is a popular solution for web applications testing, providing APIs to control the browser and interact with the web elements [7].

The complete set of (sub-)transformations moving from an UML-based Requirements Specification to the Testware can be found in [1].

A. UML2Test

UML2Test is the Model2Text transformation responsible for the Testware generation from an UML-based Requirements Specification (last grey arrow in Fig. 1), and is based again on several sub-transformations. The various UML constructs are separately transformed into code, as shown on top of Fig. 10. More specifically, **TestSuite** gives the structure of the test suite, hence grouping the test scripts together as driven by the use case diagram, **TestScripts** transforms an UML-based use case description (i.e. a state machine) into several test scripts, each one covering a specific path, while **DataClasses** and **StaticClasses** generate the auxiliary classes representing the WebApp data, the WebApp and the actors themselves, all needed to compose the test scripts instructions.

TestScripts has to handle the various paths of the state machine representing the behaviours of a use case; different algorithms solving minimum-cost flow problems may be used to extract paths from the state machine (e.g. [6]). Thus, it creates a class for all the tests scripts separately covering the state machine paths and calls, for each path, **TestScript** (Fig. 10, below). **TestScript** transforms a path into a test method of the previous class: the use case attributes become the parameters of the method, since they represent the entered data, pre/post conditions naturally become assertions, and the transitions composing the path are translated into test instructions by **Instructions**, which works basically as **GenerateStep**

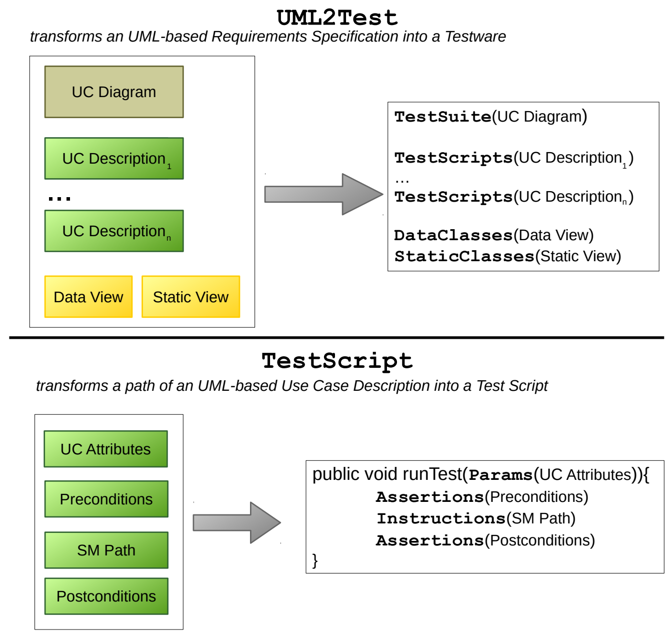

UML2Test
transforms an UML-based Requirements Specification into a Testware
UC Diagram
UC Description$_1$
...
UC Description$_n$
Data View
Static View
TestSuite(UC Diagram)
TestScripts(UC Description$_1$)
...
TestScripts(UC Description$_n$)
DataClasses(Data View)
StaticClasses(Static View)
TestScript
transforms a path of an UML-based Use Case Description into a Test Script
UC Attributes
Preconditions
SM Path
Postconditions
public void runTest(Params(UC Attributes)){
Assertions(Preconditions)
Instructions(SM Path)
Assertions(Postconditions)
}

Fig. 10. (Above) **UML2Test**: from an UML-based Requirements Specification to a Testware. (Below) **TestScript**: from a path of an UML-based use case description to a Java test script.

of Fig. 9. For each transition, **Instructions** differentiates the ones having an actor as subject from those having the WebApp; the latter are treated as assertions, since the WebApp may have to notify/show to the user the details about the content of a web page. To make test scripts directly executable, a last instrumentation step for feeding the code with the proper input data is necessary; in this work, we have chosen to make test scripts parametric in terms of the declared use case attributes, but our goal is to investigate towards smarter solutions (e.g. search-based testing).

UML2Test will rely on Acceleo[6], which is an OMG standard for model to text transformations and, again, is well-documented and embodied in Eclipse IDE.

Here follows a simplified Selenium WebDriver test script (i.e. a method) generated from a path of the state machine shown in Fig. 7;

```
public void runTest(Name N, Phone P){
  assertTrue(PhoneBook.isLoggedUserDefined()); //precond
  PhoneBook.requestsToAddNewEntry();//transition 1
  assertTrue(USER.asksForEntryDetails());  //transition 2
  PhoneBook.entersEntryDetails(N, P);  //transition 3
  PhoneBook.confirmsEntryDetails();  //transition 4
  //transition 5
  assertTrue(N.isWellFormed() &&
       !PhoneBook.existsEntryFor(N));
  assertTrue(USER.showsEntryAdded());
  PhoneBook.isAddedTo(new Entry(N, P, null));
}
```

It represents the scenario of adding a valid entry to PhoneBook. All instructions are calls to methods of the auxiliary classes, generated by **DataClasses** and **StaticClasses**, representing the operations over the data and the interactions between the USER and the WebApp. Such interactions encapsu-

[4] https://www.eclipse.org/atl/ [5] http://www.seleniumhq.org/projects/webdriver/

[6] https://www.eclipse.org/acceleo/

late Selenium WebDriver APIs; for example, entersEntryDetails is a method of the WebApp class and represents a `sendKeys` command, i.e. it enters Name N and Phone P in the corresponding text fields.

VII. Related Works

Many works investigate in the relationships between use cases (and, more generally, requirements) and testing artefacts and how to get the latter from the former. Yue *et al.* [14] proposed an automated approach to generate state machines from restrained use cases, according to a set of transformation rules; by means of a model-based testing technique applied over the state machines representing the system, test cases are extracted. Somé implemented the UCEd tool[7] to transform simplified use cases into a state model, from which abstract test cases representing use cases scenarios are extracted. Jiang *et al.* [5] proposed an approach to automatically generate test cases from use cases, whose descriptions are constrained to predefined sentences. Use cases lead in the generation of Extended Finite State Machines (EFSM), where paths corresponding to test cases can be drawn. Moreover, changes in the use cases are reflected to EFSMs, hence providing the alignment between requirements and tests. Olek *et al.* [8] introduced a Test Description Language to record manual interactions occurring on web GUI sketches, attached to use cases steps, and code them into test cases instructions. In this work, the aid of a specific tool to capture the interactions and of a language to represent them are essential.

Our approach differentiates from the aforementioned ones since, in our case, the final output are executable test cases directly runnable over a web application. In our proposal, the Requirements Specifications, both Textual and UML-based, are made precise and are integrated with the screen mockups, which empower the overall comprehension and also help in the subsequent testing process. Moreover, the vocabulary of usable terms to formulate use cases sentences is not restrained, thus provides more freedom and customization. Finally, the definition of a Requirements Specification compliant to our proposal does not require much effort or a long training, since in the future is intended to be tool-assisted.

VIII. Conclusions and Future Work

In this paper, we sketched a novel approach for web applications aimed at defining a precise Requirements Specification, either Textual or expressed using the UML, and at generating from it a functionally complete set of test scripts. Textual and UML-based Requirements Specifications are semantically equivalent and the analyst is left free to choose from which one to start. Two transformations are employed to automatically move between the Textual and the UML perspectives, and an additional transformation generates the Testware from UML. The approach is tailored for web applications, whose requirements should be precisely specified (e.g. banking, e-commerce/payments systems) and with the need for an intensive testing process to enforce their reliability.

We are currently working on implementing the various transformations and including them in a prototype tool, aimed also at reducing the manual effort needed to apply our proposal (e.g. by supporting the specification of use cases satisfying the well-formedness constraints) as much as possible. As future work, we plan to empirically evaluate such effort w.r.t. different approaches based on manual or semi-automatic test generation, and to investigate its applicability on different technologies and domains (e.g. mobile). We are also oriented in studying the maintainability cost of the Requirements Specifications and the Testware during the web application natural evolution, and in improving the proposal by generating also portions of the web application itself. Finally, we intend to investigate the generation of input data for the test scripts, for example employing search-based testing techniques.

[7] http://www.site.uottawa.ca/~ssome/Use_Case_Editor_UCEd.html

References

[1] D. Clerissi, M. Leotta, G. Reggio, and F. Ricca. Two Methods for precise Textual and UML-based Requirements Specifications. http://sepl.dibris.unige.it/2017-RET.php.

[2] D. Clerissi, M. Leotta, G. Reggio, and F. Ricca. Test driven development of web applications: A lightweight approach. In *Proceedings of 10th International Conference on the Quality of Information and Communications Technology, (QUATIC 2016)*, pages 25–34, 2016.

[3] A. Cockburn. *Writing Effective Use Cases*. Addison Wesley, 2000.

[4] S. Ghosh, D. Elenius, W. Li, P. Lincoln, N. Shankar, and W. Steiner. Arsenal: Automatic requirements specification extraction from natural language. In *NASA Formal Methods Symposium*, pages 41–46. Springer, 2016.

[5] M. Jiang and Z. Ding. Automation of test case generation from textual use cases. In *Interaction Sciences (ICIS), 2011 4th International Conference on*, pages 102–107. IEEE, 2011.

[6] J. M. Kleinberg and É. Tardos. *Algorithm design*. Addison-Wesley, 2006.

[7] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella. Approaches and tools for automated end-to-end web testing. *Advances in Computers*, 101:193–237, 2016.

[8] Ł. Olek, B. Alchimowicz, and J. Nawrocki. Acceptance testing of web applications with test description language. *Computer Science*, 15(4):459, 2014.

[9] G. Reggio, M. Leotta, and F. Ricca. A method for requirements capture and specification based on disciplined use cases and screen mockups. In P. Abrahamsson, L. Corral, M. Oivo, and B. Russo, editors, *Proceedings of 16th International Conference on Product-Focused Software Process Improvement (PROFES 2015)*, volume 9459 of *LNCS*, pages 105–113. Springer, 2015.

[10] G. Reggio, F. Ricca, and M. Leotta. Improving the quality and the comprehension of requirements: Disciplined use cases and mockups. In *Proceedings of 40th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2014)*, pages 262–266. IEEE, 2014.

[11] F. Ricca, G. Scanniello, M. Torchiano, G. Reggio, and E. Astesiano. Assessing the effect of screen mockups on the comprehension of functional requirements. *ACM Transactions on Software Engineering and Methodology*, 24(1):1:1–1:38, Oct. 2014.

[12] A. Tiso, G. Reggio, and M. Leotta. Unit testing of model to text transformations. In *Proceedings of the Workshop on Analysis of Model Transformations co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2014), Valencia, Spain, September 29, 2014.*, pages 14–23, 2014.

[13] P. Tonella, F. Ricca, and A. Marchetto. Recent advances in web testing. *Advances in Computers*, 93:1–51, 2014.

[14] T. Yue, S. Ali, and L. Briand. Automated transition from use cases to uml state machines to support state-based testing. In *Proceedings of 7th European Conference on Modelling Foundations and Applications (ECMFA 2011)*, pages 115–131. Springer, 2011.