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

Diego Clerissi, Maurizio Leotta, Gianna Reggio, Filippo Ricca

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.

Digital Object Identifier (DOI):

http://dx.doi.org/10.1109/REW.2017.39

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, Automatic Test Generation, Selenium WebDriver.**

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.

End-to-End testing is a relevant approach for improving the quality of complex web systems [8]: 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, and can also be used for the non-functional ones [11]. Moreover, the introduction of a glossary, to precisely describe the terminology referred by use cases, can 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 [11], 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 as navigational paths, basis for future test cases generation.

In this paper, we present a novel approach for generating End-to-End 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. Precision can also clarify how the functionalities of the web application have to be developed and tested, hence a specification compliant to such rules may work as a reference manual. To generate a specification, 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. Our proposal is currently tailored to the generation of test scripts for web applications, but with some adjustments it could be used to test also different kinds of systems (e.g. mobile apps).

Even though the generation of UML artefacts from use cases and consequent test cases extraction has been already investigated (for example, [14], [6], [12]), 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 and functionally complete to be exercised by the test scripts.

The proposed approach is intended to be supported by a prototype tool, to assist the final user in the definition of the requirements specifications, 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 into 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 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 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 behaviour of a system 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 [1]; in fact, they visually describe how the web application GUI should appear, and how it should react to users' interactions [11].

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[1], and a static view (i.e. a class diagram) defining the used data, the operations over them, and the interactions among the actors and the web application. In our proposal, each kind of specification is defined by means of a metamodel accompanied by a set of well-formedness constraints [2].

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, the screen mockups over which the test scripts instructions are performed, and the auxiliary classes coding the data, the operations over them and the occurring interactions.

[1] Among the variety of UML diagrams, we chose state machines to represent use cases, since they naturally describe the behaviour of a web application in terms of reactions to users interactions. Sequence diagrams may be employed as well, but usually require a larger number of constructs and may result more complex to understand, whereas activity diagrams are less suitable to represent the interactions between users and web application.

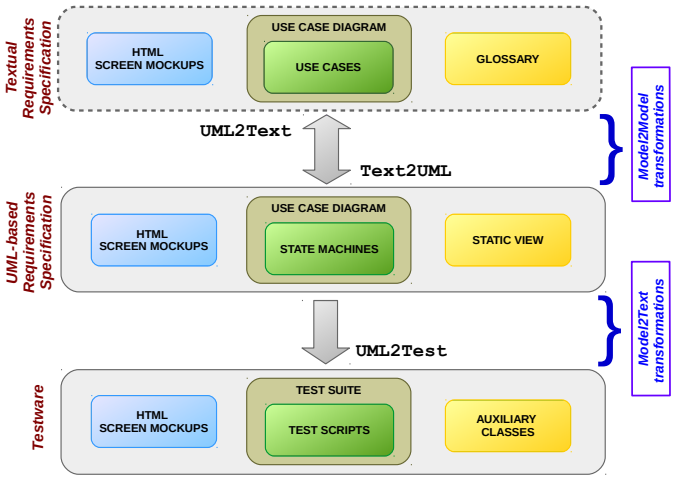


Fig. 1. The proposed approach: an overview.

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 view. Instead, **UML2Text** generates use cases from state machines, and the glossary from the static view. Notice that the screen mockups and the use case diagram 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 the 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 a precise form of 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 [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.

From now on, we use the term WebApp to denote a generic web application we want to test, after having specified its requirements. 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 [2].

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[2]) accompanied by a set of well-formedness constraints [2].

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.

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

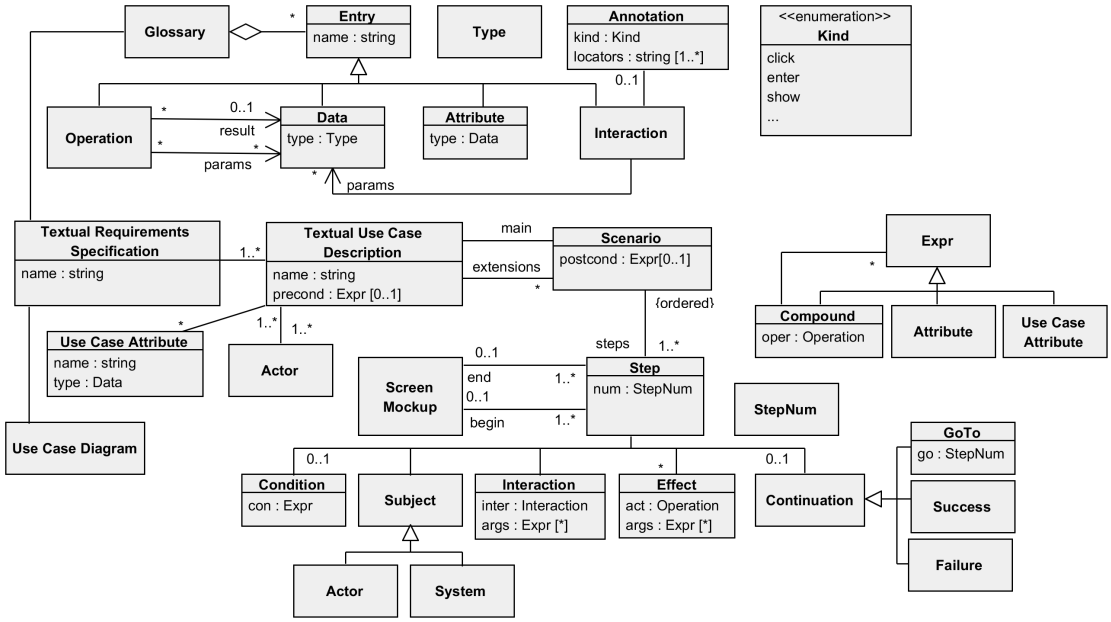


Fig. 2. The textual requirements specification metamodel.

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) 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) 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 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 set/get their content. They have form "$\underline{\text{namepart}_1}$ Data_1 ... $\underline{\text{namepart}_n}$ Data_n [returns Data]", where each Data is a previously defined data and the operation may have or not a return type. The semantics of the operations is given in a structured textual form. For example, in Fig. 3 we have "N: Name is well formed returns bool", which checks that the size of a Name N is less than 32, where "size of", "is less than" and "32" are predefined functions over strings and integers.

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
...

Attributes
LoggedUser has type Username
RegisteredEntries has type sequence(Entry)
...

Operations
N: Name is well-formed returns bool
  means size of N is less than 32
exists entry N: Name returns bool
  means RegisteredEntries includes <N, -, - >
...

User → PhoneBook Interactions
requests to add a new entry
  [click: "addNewEntry"]
enters entry details Name and Phone
  [enter: "name", "phone"]
...

Fig. 3. A portion of PhoneBook glossary.

4) Interactions: the atomic interactions between the actors and the WebApp and vice versa. They have form "$\underline{\text{namepart}_1}$ Data_1 ... $\underline{\text{namepart}_n}$ Data_n", where each Data has been previously defined. For example, in Fig. 3 we have requests to add a new entry, and enters entry details Name and Phone. The parts enclosed by square brackets are explained in Section III-D.

C. Use Cases Description

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 data introduced in the glossary. In Fig. 4, two attributes are declared: Name N and Phone P, where Name and Phone are data defined in the glossary (see Fig. 3).

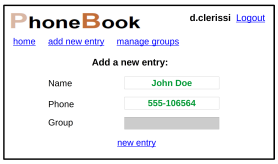
The *preconditions* state what we assume about the current state of the WebApp before the execution of the associated use case (optional). They are expressions built using the data, the attributes and the operations defined in the glossary, and are shared among all the scenarios composing the use case description. 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. 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 preconditions; for example,

Use Case: Add Entry
Actors: User
Attributes: Name **N**, Phone **P**
Preconditions: LoggedUser is defined
Main Success Scenario:
1. User requests to add a 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 **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 **N**, then PhoneBook shows entry already exists. InvalidEntryPage

Fig. 4. PhoneBook Add Entry use case.

in Fig. 4 a condition is associated with step 5 and checks if the entry name is well-formed and not already present in the registered entries (the definitions of 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 operations (having a side effect) 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.

Successful scenarios can optionally be associated with *postconditions*, as shown in Fig. 2, which are expressions built similarly to preconditions, but specific for a completed scenario.

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 [11] (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 some 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 ambiguities in the requirements specifications, instead of improving their quality [11]. The consistency between mockups and use cases steps is granted by a set of well-formedness constraints to be satisfied while creating the mockups and writing the steps [2]. An example of a simple screen mockup associated with a use case step is shown in Fig. 4; the

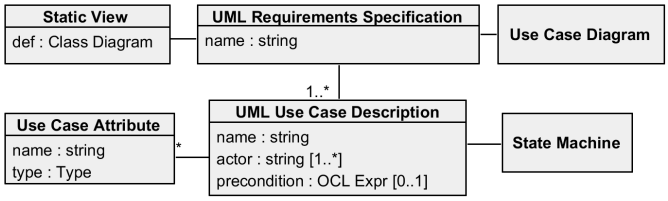


Fig. 5. The UML-based requirements specification metamodel.

AddEntryPage mockup is linked to step 3 and contains all the web elements and all entered data to perform the step interaction.

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[3] of the involved web elements. Annotations have form [kind: $\text{locator}_1, \ldots, \text{locator}_n$], where kind represents the kind of interaction performed over the web elements (e.g. enter, click), and $\text{locator}_1, \ldots, \text{locator}_n$ are strings identifying them (see the Annotation class in Fig. 2). Different types of locators exist [9], 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 the step 3 of the use case shown 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 a static view (i.e. a class diagram) defining the used data, the attributes, the related operations, and the interactions among the actors and the WebApp. A UML-based requirements specification is again precisely defined by a metamodel (see Fig. 5) and a set of well-formedness constraints [2].

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

A. Static View

The *static view* is a class diagram basically equivalent to the glossary part of the textual requirements specification. It contains:

1) Datatypes: the UML datatypes defining the data needed to express the requirements. For example, in Fig. 6 we have Name and Phone, each one containing string values.

2) Web App: the UML class modelling the WebApp. It is stereotyped by ≪webapp≫ and contains the attributes describing its updatable state and the interactions performed by the actors towards the WebApp. For example, in Fig. 6, PhoneBook class has RegisteredEntries attribute, storing all

[3] 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) [8]

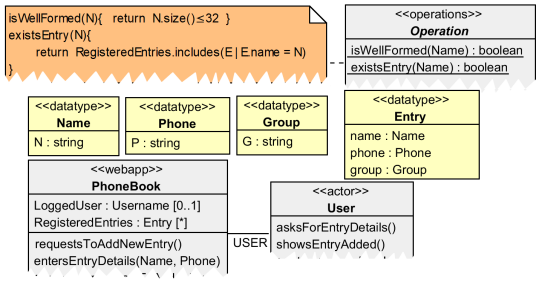


Fig. 6. A portion of PhoneBook static view.

the entries in the WebApp, and entersEntryDetails interaction, called by actors whenever a new entry is added.

3) Actors: the UML classes modelling the actors, hence containing the interactions performed by the WebApp towards them. Each actor class is stereotyped by ≪actor≫ and must be connected to the WebApp class by an association, named as the actor itself. For example, in Fig. 6 User class has showsEntryAdded interaction, used by the WebApp to inform the user about an entry added successfully.

4) Operations: the UML operations performed over data and attributes. They are static, can have or not a return type, and are grouped in the abstract class *Operation* (stereotyped by ≪operations≫). The definitions of the operations are given in attached notes and expressed using UML action language[4]; in the UML terminology, the definitions in the notes are the methods associated with such operations. For example, in Fig. 6 the note attached to isWellFormed operation of the *Operation* class describes whenever a Name N can be considered well-formed, in a similar way to the glossary part shown in Fig. 3.

B. Use Cases Description

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 static view (see Fig. 7).

The *preconditions* are OCL expressions put in notes attached to the starting states of the state machine. In Fig. 7, the precondition refers to oclIsDefined predefined OCL operation and is equivalent to the one given in Fig. 4.

The transitions of the state machine, representing the interactions among the actors and the WebApp, have one of the following forms:

– *Interaction* [*Condition*] / *Effect**, if the transition corresponds to an interactions from an actor towards the WebApp. *Interaction* is an event built by an interaction of the WebApp class, *Condition* is a boolean OCL expression, and *Effect** are either calls of operations of the *Operation* class or basic UML actions, in any case updating the WebApp state. In Fig. 7, the third transition is an event built by entersEntryDetails interaction of the WebApp, which uses the declared use case attributes to add a new entry. The transition is semantically equivalent to step 3 of Fig. 4.

– [*Condition*] / ACTOR.*Interaction* ; *Effect**, if the transition corresponds to an interactions from the WebApp towards an actor. *Interaction* is an event built by 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 User class, and the effect of updating the current WebApp state with the entered entry, by calling a basic UML action. The transition is semantically equivalent to step 5 of Fig. 4.

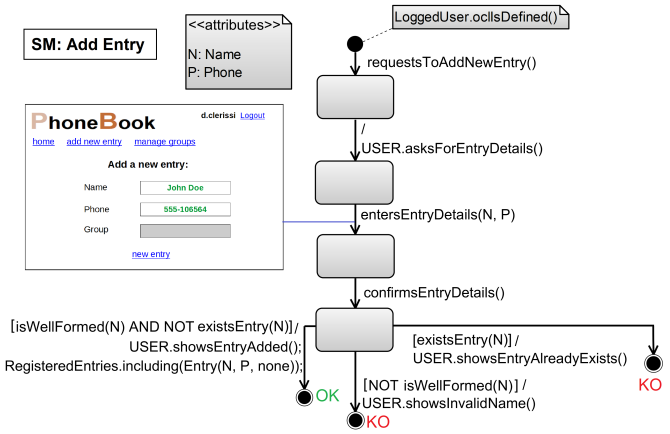


Fig. 7. PhoneBook Add Entry state machine.

Similarly to use cases, *postconditions* can optionally be associated with successful paths (i.e. those ending in a state labelled by OK, see Fig. 7), as OCL expressions put in notes attached to the ending states.

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 target 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 source 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 well as in the textual perspective, screen mockups must be consistent with the transitions they are linked to [11]. Moreover, the web elements of the screen mockups that are affected by the interactions in the state machines (e.g. entersEntryDetails of 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 tagged values. A tagged value encapsulates the kind of interaction performed and strings representing the locators of the web elements. The form adopted by tagged values for an interaction is {kind = locator_1, ..., locator_n}. For example, entersEntryDetails interaction of the WebApp class is associated with the tagged value {enter =

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

"name", "phone"}, indicating that the data about name and phone will be entered in the textfields identified by "name" and "phone" strings.

V. Transformations between Textual and UML-based Requirements Specifications

In our proposal, the transformations are initially described from a 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 source to target universes has been inspired by Tiso *et al.* [13].

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 [2].

A. Text2UML and UML2Text

`Text2UML` and **`UML2Text`** (shown as the bi-directional grey arrow in Fig. 1) are the Model2Model transformations between textual and UML-based requirements specifications. For space reasons, we just sketched **`UML2Text`** and omitted **`Text2UML`**, which is basically the inverse of the former.

`UML2Text` transforms a UML-based requirements specification into a textual one and is composed of several sub-transformations, each one handling a different part of it. The main transformation is defined 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, **`UCD`** transforms a UML-based use case description into a textual one, while **`Glossary`** transforms the datatypes, the attributes, the operations and the interactions defined in the static view into glossary entries. The use case diagram is instead kept unaltered. Again, further sub-transformations compose **`UCD`** (Fig. 8, below), generating the actors, the use case attributes, the preconditions, 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.

Given a state machine and its corresponding use case, the number of individual paths and scenarios is the same, thus use cases and state machines can be considered *isomorphic* in terms of transformations. 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.

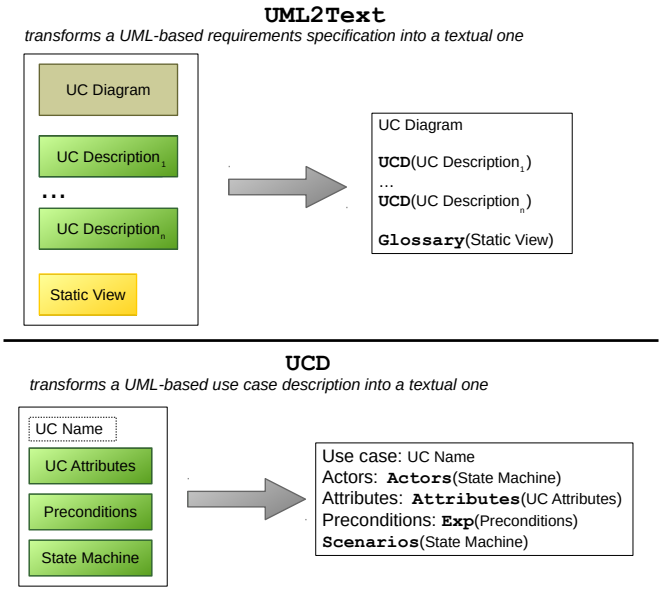


Fig. 8. (Above) **`UML2Text`**: from a UML-based to a textual requirements specification. (Below) **`UCD`**: from a UML-based to a textual use case description.

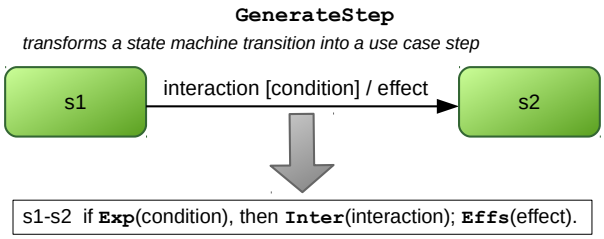


Fig. 9. **`GenerateStep`**: from a state machine transition to a use case step (an interaction from an actor towards the WebApp).

Among the activities we omitted, **`Scenarios`** calls **`GenerateStep`**, sketched in Fig. 9, which transforms a transition into a step; **`Exp`**, **`Inter`**, and **`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 both 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[5], by now the standard for Model2Model transformations and also highly supported, well-documented, and integrated in Eclipse IDE.

VI. Transformation from a 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.

[5] https://www.eclipse.org/atl/

- *Auxiliary Classes*: all the code corresponding to the entities defined in the static view, i.e. data, attributes, operations and interactions, used in *Test Scripts* instructions.
- *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[6], which is a popular solution for web applications testing, providing APIs to control the browser and interact with the web elements [8].

The complete set of (sub-)transformations moving from a UML-based requirements specification to a testware can be found in [2].

A. UML2Test

UML2Test is the Model2Text transformation responsible for the testware generation from a 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 a UML-based use case description (i.e. a state machine) into several test scripts, each one covering a specific path, while **AuxClasses** generates the code representing the content of the static view, needed to formulate 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. [7]). 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 are naturally treated as assertions (notice that, since postconditions are specific for successful paths, they are taken directly from the current path, if any), and the transitions composing the path are translated into test instructions by **Instructions**. 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[7], which is an OMG standard for Model2Text transformations and, again, is well-documented and embodied in Eclipse IDE.

[6] http://www.seleniumhq.org/projects/webdriver/
[7] https://www.eclipse.org/acceleo/

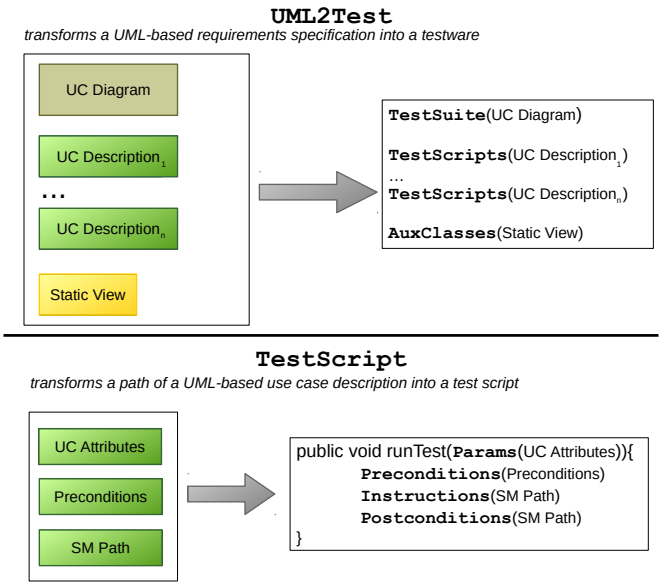


Fig. 10. (Above) **UML2Test**: from a UML-based requirements specification to a testware. (Below) **TestScript**: from a path of a UML-based use case description to a Java test script.

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.LoggedUser != null); //precond
  PhoneBook.requestsToAddNewEntry(); //transition 1
  assertTrue(User.asksForEntryDetails());  //transition 2
  PhoneBook.entersEntryDetails(N, P);  //transition 3
  PhoneBook.confirmsEntryDetails();  //transition 4
  //transition 5
  assertTrue(Operation.isWellFormed(N) &&
       !Operation.existsEntry(N));
  assertTrue(User.showsEntryAdded());
  PhoneBook.RegisteredEntries.add(new Entry(N, P, null));
}
```

It represents the scenario of adding a valid entry to PhoneBook. All instructions are calls to attributes or methods of the auxiliary classes, generated by **AuxClasses** (i.e. PhoneBook, User, and *Operation*), representing the operations over the data and the interactions between the user and the WebApp. Such interactions encapsulate 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 textfields.

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[8] to transform

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

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.* [10] 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 such interactions are essential. Cucumber[9] is a behaviour-driven development software tool to describe requirements of applications in a structured way and use them as a guidance for the development and testing phases. Functionalities follow restrained scenarios adhering to a *given, when, then* template, from which test cases instructions are generated.

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 will not require much effort or a long training, since in the future it is intended to be tool-assisted, and could be helpful in the implementation of the web application, given that the flow of interactions occurring among the users and the WebApp are made explicit and precise.

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, thus 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 currently tailored for web applications, whose requirements should be precisely specified (e.g. banking, e-commerce/payments systems, government services), having functionalities clearly described in terms of GUI and interactions - hence using the requirements specification as a sort of user manual - 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. Making requirements specifications precise, even if tool-assisted, is an onerous task, but the preliminary effort is rewarded in the last stage of the approach, where the testing artefacts are automatically derived and kept aligned with the requirements. 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) and on larger/more complex systems. 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.

[9] https://cucumber.io/

References

[1] 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. IEEE, 2016.

[2] D. Clerissi, G. Reggio, M. Leotta, and F. Ricca. Additional material: reference manuals for precise textual and UML-based requirements specifications methods, PhoneBook specifications, and transformations. http://sepl.dibris.unige.it/2017-RET.php.

[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 *Proceedings of 4th International Conference on Interaction Sciences (ICIS 2011)*, pages 102–107. IEEE, 2011.

[6] S. Kansomkeat and W. Rivepiboon. Automated-generating test case using UML statechart diagrams. In *Proceedings of SAICSIT 2003*, pages 296–300. South African Institute for Computer Scientists and Information Technologists, 2003.

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

[8] 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.

[9] M. Leotta, A. Stocco, F. Ricca, and P. Tonella. ROBULA+: An algorithm for generating robust XPath locators for web testing. *Journal of Software: Evolution and Process*, 28(3):177–204, 2016.

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

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

[12] P. Samuel, R. Mall, and A. K. Bothra. Automatic test case generation using UML state diagrams. *IET software*, 2(2):79–93, 2008.

[13] A. Tiso, G. Reggio, and M. Leotta. Unit testing of model to text transformations. In *Proceedings of 3rd Workshop on the Analysis of Model Transformations (AMT 2014)*, pages 14–23. CEUR, 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 ECMFA 2011*, pages 115–131. Springer, 2011.