From UML models to automatic generated tests for the dotLRN e-learning platform

Ana Cavalli [1] Stéphane Maag [2] Sofia Papagiannaki [3]
Georgios Verigakis [4]

GET / Institut National des Télécommunications
9 rue Charles Fourier
F-91011 Evry Cedex, France

Abstract

This paper presents a method for testing an e-learning, web based system. System specifications are provided using the UML modelling language and specifically the Sequence, Activity and Class diagrams of UML. These specifications are exported in XMI format which is parsed in order to produce the test cases. The system under consideration in this paper is dotLRN, an open source enterprise-class suite of web applications and portal framework for supporting course management, online communities and collaboration.

Key words: UML, dotLRN, XMI, Component Testing, Validation

1 Introduction

During the last few years more and more organizations and companies exploit the services they provide by making them available through their web sites. In this context, educational organizations can adopt an e-learning application to manage and support remote distance courses. Among the advantages of such a course over a conventional one is its capability to overcome the geographical barriers and, as a consequence, to address to a larger audience. Furthermore, it is very important to make the cost of such a system is minimal. The organization only needs to deploy a powerful web server with the goal that the users may access these services using a minimum computing equipment connected to the internet.

[1] Email: Ana.Cavalli@int-evry.fr
[2] Email: Stephane.Maag@int-evry.fr
[3] Email: Sofia.Papagiannaki@int-evry.fr
[4] Email: Georgios.Verigakis@int-evry.fr

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Since software systems get more and more sophisticated and complicated, software testing is very important not only to uncover bugs but also to ensure that the system conforms with the specifications and really does what is supposed to do. The fact that big systems are decomposed in smaller sub-systems developed by different teams brings new challenges to testing, since it has to verify that these sub-systems are integrated together smoothly.

Furthermore, testing an e-learning system is essential because by its nature, such an application is exposed in many threats: a web server has to respond to request from each node in the network. Therefore, testing should guarantee that a user cannot gain permissions over the system he is not supposed to have. For instance it would be undesirable if a student got the permission to change his grades. In addition to this, modern web applications are becoming increasingly complex and mission critical. This situation becomes even worse because of the lack of standardization in the web clients (browsers). Testing has to assert the system usability. For instance a user is unable to complete a process because the content of the web page does not appear correctly or due to a slow network connection he may be unable to login because of not realistic timeout value. Our approach takes into account how these functional aspects have been implemented, checking that they conform the specification.

This paper is part of the work being developed in the framework of the E-LANE project, an European and Latin American collaboration for the creation of an advanced integrated system for e-learning in which GET/INT is a partner[5]. We propose a testing method for dotLRN, that is composed by a suite of web applications and a portal framework of an e-learning system, and is used as the base platform for the development of E-LANE.

Our methodology is mainly influenced by [1,2], which describes how UML diagrams developed in the Analysis phase are analyzed in an automated way in order to produce test cases. Indeed, while [2] focuses on integrating testing conducted in an incremental way, [1] describes how to use class invariants and a detailed formal description in UML diagrams. Both of them are used to derive test requirements and the test suites. In this paper, the purpose and the challenge is that instead of an object-oriented software we have to use UML to model and then to validate a web application tool. The methodology presented in these two previous papers are related to big and complex real-world systems and are not applicable to web-based applications. Indeed, here we need to check the graphical user interfaces but also the content of the generated pages. Our contribution is to provide an answer to these difficulties. There is an extension of UML for Web Applications but that is more focused on the Design View. Therefore, in order to produce diagrams which convey the information we need to feed our tests, we got inspired from [5] and [4]. Another contribution of this paper is to present the techniques and the tools which are used to component and conformance testing.

[5] http://www.e-lane.org

The article is organized as the following. Section 2 describes the dotLRN platform, section 3 presents how to model dotLRN for testing, in section 4 we describe the tools to automate the test generation out of the models and finally section 5 concludes the article.

2 The dotLRN e-learning platform

dotLRN [6] is a web based e-learning platform and portal framework, designed to support course management, online communities and collaboration. It is open source and built over the OpenACS [7], a toolkit for building scalable, community-oriented web applications.

In dotLRN there are three main portal types: *user*, *class* and *community* portals. The user portal is the private space that each user owns, while the class and community portals contain all the pages related to a specific class or community. Each of these three portal types is divided into four sections: the main space, the calendar, the files and the control panel. The pages in dotLRN are composed of *portlets*. Portlets are small portals that have some specific functionalities, like the forums, the news or the calendar.

A user portal is created automatically whenever a new user is registered in dotLRN, but the class and community portals are created by the *site-wide administrator* according to the needs of the users. When a new class or community portal is created, the site-wide administrator assigns one ore more users as administrators of this portal. For example, for computer science class portal, the administrators can be the professor and his teaching assistants, while for a photography group community portal, the administrator will be one or more students. The responsibilities of a portal administrator are to add content, customize the layout and decide the policy of the portal.

When a class or community has an *open* policy, any user can join, while when the policy is *closed* only the administrator can add users. A third policy exists, the *wait* policy, where any user can ask to join and then the administrator will decide to accept or deny this user.

3 UML models for testing

In order to derive our test cases, we need to grasp and describe the functionality of the system in a formal way. Therefore, we choose to model the system using UML which has become a standard. At the beginning, we discover the use cases of the system under consideration and document them in a *Requirements Document* which contains the scope of the system, the main actors, the use case diagram and textual description of each use case [3]. The output of this step is not directly used as an input to our test suite but it is important in

[6] http://www.dotlrn.org
[7] http://openacs.org

order to design the UML diagrams which actually constitute our input data. The UML diagrams we use in our approach are:

- Activity Diagram for each actor: displays dependencies among the use cases.
- Sequence Diagram for each use case: describes the main and the alternative scenarios of the use case.
- Class Diagram: introduces the main classes of the system.
- Navigation Map: a Class Diagram which provides information about the dynamic content of the web pages.

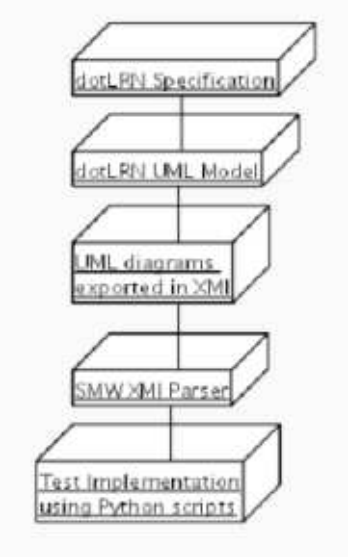

Figure 1. Outline of our methodology

The next step is to export these diagrams in an XMI format. This activity is supported by all the modern CASE tools like ArgoUML and Rational Rose. Afterwards we parse the XMI and we produce a program which connects to the web server and makes requests according to the given scenario in the Sequence Diagram. Finally, the web page of the response is examined to verify if it conforms with the specification. Figure 1 presents briefly the different steps of our study.

3.1 Modelling Use Case Dependencies

The use cases of a system are not independent. Apart from the *include* and *extend* relationships among them, there are also sequential dependencies. In order one use case to be executed, another should have taken place before. For instance, in dotLRN the user should login before being able to do anything else. Since the automation of the testing procedure is also of concern, we have to describe somehow these dependencies. We achieve this by introducing an activity diagram where the vertices represent use cases and edges are sequential dependencies between the use cases.

An edge in such a diagram denotes that the use case in the tail has to be executed before the use case in the head. *fork* and *join* are used when some use cases should be executed independently in order another one to take place. For instance, in dotLRN "Add subject" and "Add term" are two independent use cases which should be synchronized so as the "Add class" can be tested.

Furthermore, in this diagram we have included also the parameters of the use cases. The reason is that sometimes it is easier to realize the dependencies between the parameters of the use cases. For instance, in the above example, in order to add a new class, the administrator should provide information about the term(Term.name) and the subject of the class(Subject.name). As a consequence, there is a dependency between the "Add class" use case and the "Add Term" and "Add Subject" use cases.

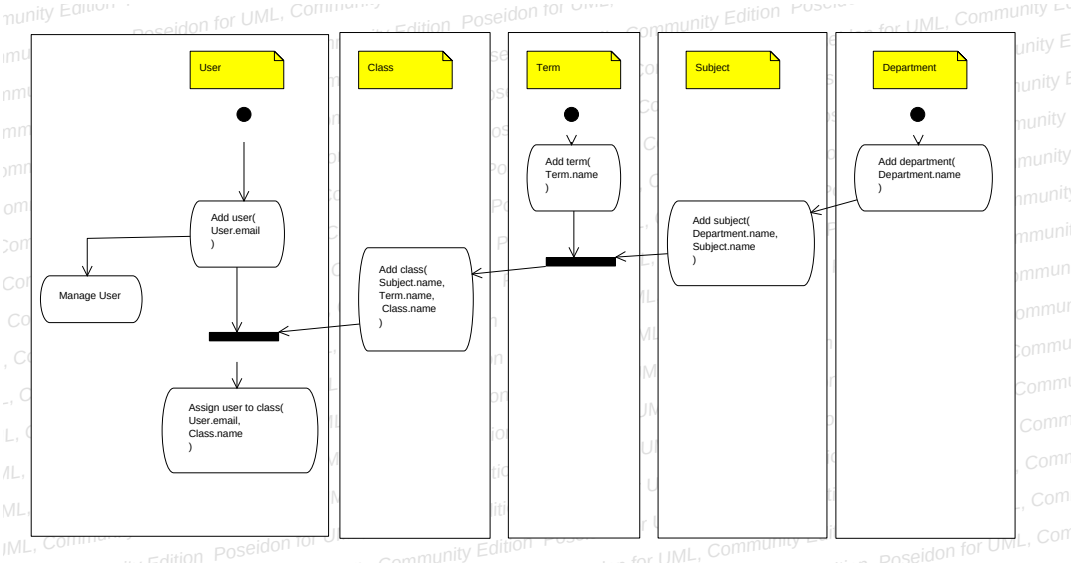

Figure 2. Activity diagram dependencies for "Assign user to class" use case

Finally, in the diagram the use cases are organized in group according to the object are associated with. These objects are instances of the classes in the Class diagram. Figure 2 shows the respective activity diagram for the Administrator. According to this latter, "Add department" should precede "Add subject". Also, "Add term" and "Add subject" should occur before "Add class", and "Add user" and "Add class" should take place before the execution of "Assign user to class". Finally, "Manage User" depends on "Add User" since first the user should be added to the system and then the administrator can edit his profile and modify his permissions.

In the testing phase, before simulating the scenarios in the Sequence diagrams these activity diagrams should be scanned to obtain the sequence by which the use cases will be tested.

3.2 Sequence Diagram

In UML, a Sequence diagram realizes the interaction of objects via the interchange of messages in time. Similarly, as in activity diagrams the objects are instances of a class described in the Class diagram. Usually the sequence diagrams describe a single scenario.

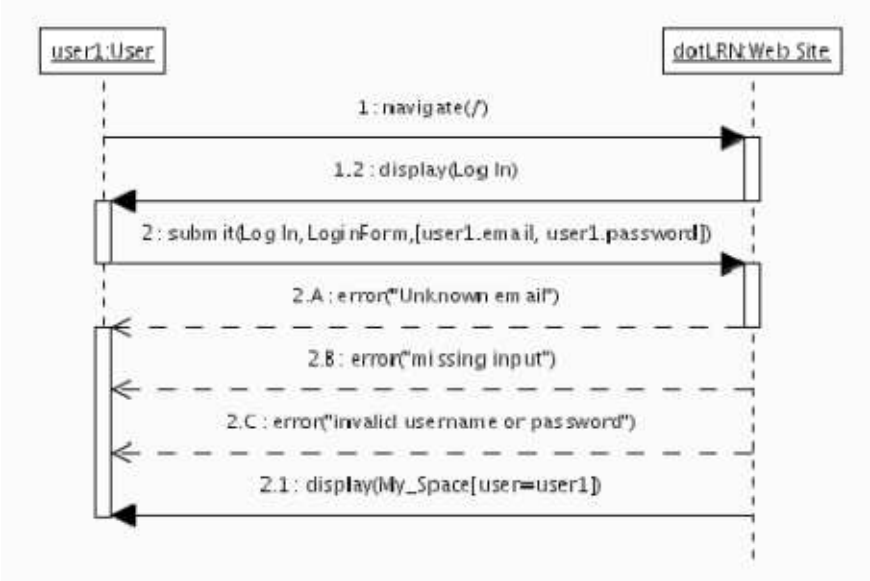

Figure 3. Sequence diagram for "Login" use case

We enumerate the messages as described in [2] so we can illustrate a number of alternative scenarios in the same diagram. According to this convention, capital letters denote alternatives (error messages). By adopting this tactic we can derive easily the different *Message_Sequences* [1] related to the same use case. Figure 3 shows the respective sequence diagram for the "Login" use case.

Our Sequence diagrams are also parameterized since input parameters can influence the execution and constitute separate *Choices* [1]. Such a parameter can be the email of a User. Whether this email belongs to a registered user (exists in the database) or belongs to a new user (does not exist in the database) determines what is going to happen later. In the former case the dotLRN page is displayed otherwise a warning appears in the Log In page. During the testing procedure, if there are such branches and parameters then the produced program has to fork to test all the different possibilities.

Table 1 summarizes the actions we use in the Sequence diagram organized as HTTP requests of the user and possible HTTP responses returned to the user by the server, since the success or the failure of our tests depends upon these requests and the respective responses. Since the system under testing is a web application there are three possibilities for the user: either navigates to a URL or requests a web page through another one (clicks on a link to the

HTTP request	**HTTP possible responses**
navigate(url:String): User makes an HTTP request for a url	**display(page:WebPage)**: Web server returns the requested web page
link(target:String): User clicks in a HTTP link	**display()ege:WebPage**: Web server returns the target web page
submit(page:WebPage, form:Form, data:List): User submits an HTTP form	**display(page:WebPage)**: in case of legitimate input the web server responses with a new web page **error(msg:String)**: in case of wrong input the web server responses with the previous page displaying a warning message

Table 1
Actions of the Sequence Diagram

wanted page) or submits information by filling an HTML form. The system answers either directly by returning the requested page (display) or by giving an error message.

3.3 Navigation Map

The Navigation Map of a web application introduced in [5] is useful because it provides information about the dynamic content of each web page which is part of the system as well as the links between the different web pages. This information is essential during the parsing of the HTML pages (section 4.3).

The Navigation Map is a Class diagram where each web page is a class and a link between two pages is an association between the two respective classes. This extention of UML for web applications introduces a number of *stereotypes*, *tagged values* and *constraints*.

The Table 2 summarizes the mapping we use between web entities and metamodels in a class diagram.

4 Automating the test generation

To automate the test generation, our goal is first to parse the UML diagrams obtained from the previous steps. Therefore, based on these diagrams, we generate the necessary requests to the dotLRN server and then check if the server's replies are as expected by the previous models. Since dotLRN is a

Web Entity	UML Metamodel	Tagged Values
web page	«web page» class	**TitleTag:** the title of the page **RelativeURL:** the relative URL of the page **BodyTag:** the set of attributes for the <body> tag
page scripts	«web page» class operations	
page variables	«web page» class attributes	
link	«link» association	
form	«form» class	**Method:** GET or POST
form types	«form» class attributes stereotyped by «form input»	
portal page	«portal page» class	**TitleTag:** the title of the page **RelativeURL:** the relative URL of the page **BodyTag:** the set of attributes for the <body> tag
portal element	«portal element» class	**BelongTo:** the «portal page» it belongs

Table 2
Mapping Web Applications to UML Class diagrams. Stereotypes are presented inside « ».

web application, the requests are HTTP requests that simulate a user navigating the site through a web browser. The possible actions are to fill and submit a form, to click on a link or to navigate to a given URL. Similarly the server's replies are HTTP Responses that can either contain an HTML page, a redirection to another URL or an error message. Assuming the first case, the HTML page of the response has to be parsed to see if its contents are the expected ones.

Based on these requirements, we had to choose the components that were required to build our test suite.

4.1 The programming language

Since we are dealing with UML, it would be more efficient to choose an object-oriented language. We also wanted this language to provide easy string handling and a high level of abstraction for network communications. The programming language that we found as the most suitable was Python[6]. Python is a modern object-oriented language which combines remarkable power with very clear syntax. Its built-in modules provide numerous functions that facilitate string handling and networking.

4.2 Parsing and executing the UML

To parse the UML diagrams we could either use the API of a UML tool, or export the diagrams in an XMI format that would allow to parse them using an XML parser. XMI is a standard created by the Object Management Group (OMG) to represent UML diagrams in XML. Exporting to XMI was the solution we preferred since it does not tie us to a specific tool.

Although we could use any XML parser to parse the XMI, due to the high complexity of the standard we decided to use a specialized XMI parser. The one we used was the parser included in the System Modeling Workbench tool [8]. It is free, open-source and also is written in Python, making it easier to integrate with our code. Being open-source it also enabled us to fix some incompatibility issues that appeared when used with XMI produced by the Poseidon tool.

4.3 Parsing the HTML pages

Since HTML mixes presentation and content data, the HTML output of dotLRN does not allow us to extract the information we want without first looking the implementation details. To avoid this we need to change the page templates of dotLRN in order to provide the data in a more formal way. We achieve this by adding *id* attributes to the tags we want to query. For example, to the td tag that contains the user's name in the user pages will have an attribute *id="username"*. That way we can query any page independently of the implementation of the layout of the page.

4.4 Example implementation

In this section we give the skeleton of a possible solution. In the code that follows we have left out some code (mainly some functions) in order to reduce the size and increase the clarity.

```
1 from urllib import urlopen
2 from smw.io import XMIStreamer
3 from smw.metamodel import UML14
```

[8] http://www.abo.fi/~iporres/html/smw.html

```
4
5  class TestSuite:
6          returnedPage = None
7
8          def validateLink(self, link):
9                  operation = self.getOperation(link)
10
11                 if operation.name == "navigate":
12                         url = self.getOperationParameters(operation)
13                         fd = urlopen(serverbase + url)
14                         self.returnedPage = fd.read()
15                         fd.close()
16
17                 elif operation.name == "display":
18                         params = self.getOperationParameters(operation)
19                         pageTemplate = generatePageTemplate(params)
20
21                         parser = dotHTMLParser(pageTemplate)
22                         parser.feed(returnedPage)
23
24         def execute(self, source):
25                 xmi = XMIStreamer(UML14)
26                 fd = open(source, "r")
27                 model = xmi.loadFromStream(fd)
28                 fd.close()
29
30                 sequenceDiagram = self.getSequenceDiagram(model)
31                 for link in self.getLinks(sequenceDiagram):
32                         self.validateLink(link)
```

Function *execute* (line 24) is the main function of the class and it reads the XMI code from the file defined in the variable *source* that is given as a parameter. It then isolates the Sequence Diagram (for this example we assume that only one exists) and then validate one by one all its messages (links). All the getX functions (like getSequenceDiagram and getLinks) are trivial to write and they consist of navigating through the structures generated by SMW to get a specific data. They are assumed to be defined inside the class.

The validation of each link depends on the operation. If the operation is *navigate* (lines 11-15) then we have to extract the destination URL from the parameters and then get the requested page. We assume that destination is a relative URL, so we use the *serverbase* variable (line 13) to make it absolute. The page is then kept in the *returnedPage* variable to be used by the following commands.

In the case of a *display* operation (lines 17-22), we create a template of the page based on the operations parameters and then use our HTML parser to compare the returnedPage with the template. The skeleton of the parser is as follows.

```
1  from HTMLParser import HTMLParser
2
```

```
class dotHTMLParser(HTMLParser):
        pageTemplate = None

        def __init__(self, pageTemplate):
                self.pageTemplate = pageTemplate

        def handle_starttag(self, tag, attrs):
                for attr in attrs:
                        if "id" in attr:
                                validateElement(tag, attr)
```

The *dotHTMLParser* class inherits the HTMLParser class and overides the *handle_starttag* function to search for elements that have an *id* attribute. Every such element will be validated according to the pageTemplate that was given during the instatiation (the details of the validation are not shown).

Similarly to the two example operations we can write the code to handle the rest of the supported operations.

5 Conclusion

We have presented in this paper a new approach to test the functionality of an e-learning, web based system, the dotLRN platform. This platform presents the advantage to be an open source toolkit for building scalable, community-oriented web applications.

The method and the software tool we propose has been applied to dotLRN platform but they are generic enough and can be applied to other e-learning and web based systems.

The test method is based on the test of objectives, which are selected taking into account the experts and designers advice. These tests are based on user cases and cover all relevant aspects of the system behaviour. Even if we cannot guarantee a total error coverage, it can be guaranteed for the selected tests.

The software tool presented perform the parsing of the UML specification, the generation and translation of the tests to the XMI form and their execution on the dotLRN platform. In a first step, we started with the test of communication interfaces and user requirements. Next steps will be the test of authentication mechanisms and application contents (for instance, to check the content of the required web page).

Experimentation results are very promising. The automation of the testing procedure reduces the time and costs to produce e-learning reliable software tools. In addition, the test of users requirements contributes to the design of tools with easy and convivial interfaces.

References

[1] Basanieri, F., A. Bertolino and E. Marchetti, *The Cow_ suit Approach to Planning and Deriving Test Suites in UML Projects*, Proc. Fifth International Conference on the Unified Modeling Language - the Language and its applications UML 2002, LNCS 2460, Dresden, Germany (2002), pp. 383–397.

[2] Briand, L. C. and Y. Labiche, *A UML-Based Approach to System Testing*, Software and Systems Modeling **1** (2002), pp. 10–42.

[3] Cockburn, A., "Writing Effective Use Cases," Addison-Wesley, 2000, 1st edition.

[4] Conallen, J., *Modelling Web Application Architectures with UML*, Communications of the ACM **42** (1999), pp. 63–70.

[5] Conallen, J., "Building Web Applications With UML," Addison-Wesley, 2002, 2nd edition.

[6] Jones, C. and F. Drake, "Python & XML," O'Reilly & Associates, 2001, 1st edition.