UCM-Driven Testing of Web Applications

Daniel Amyot[1], Jean-François Roy[1], and Michael Weiss[2]

[1] SITE, University of Ottawa, Ottawa, Canada
{damyot, jroy}@site.uottawa.ca
[2] School of Computer Science, Carleton University, Ottawa, Canada
weiss@scs.carleton.ca

Abstract. Despite their apparent simplicity, Web applications are surprisingly difficult to develop, if our aim is to build applications that behave correctly under regular conditions as well as adverse circumstances like out-of-order requests and race conditions. In this paper, we describe our experiences deriving customer-oriented acceptance tests for Web applications by modeling the essential capabilities of such applications with Use Case Maps (UCMs). Abstract test purposes are generated from a UCM model using scenario definitions and scenario extraction tools. These test purposes are then converted interactively to test cases in the FitNesse acceptance testing framework, which is popular in the Extreme Programming (XP) community. The test cases are used to validate a Web application where several typical but non-trivial bugs were planted. Challenges in the automation of such process are also discussed.

1 Introduction

Web applications can be surprisingly difficult to develop because they need to support a wide variety of expected usage scenarios. Moreover, we want these applications to be robust, that is, we want them to behave correctly under unexpected or averse circumstances. Due to the strict time-to-market requirements imposed on Web development projects, modeling and testing are often considered too time-consuming and lacking significant payoff [11]. More significantly, though, their general lack of robustness is due to the "openness" of Web applications and to concurrency issues.

Openness is most often associated with security concerns. In a recent survey [14], more than 90% of Web applications were found to be vulnerable to common hacking attacks such as cross-site scripting, parameter tampering, and cookie poisoning. These problems are usually caused by a failure of the Web application to properly validate user input. However, this is but one way that Web applications are "wide open" to unexpected use, or malicious exploitation.

Unlike in a "closed" desktop application, or even a distributed application with a tightly controlled API, requests can be sent in any order to a Web application. Expecting the input pattern to follow the designed navigational structure may lead to subtle design errors in developing the Web application. For example, a user could bookmark a page, and resubmit the request associated with it

later. Furthermore, Web applications can be accessed by different types of clients (browsers, robots, etc.), not just those they were designed for.

Other common errors are caused by race conditions. Web applications have been plagued by these from the beginning. A famous example is the multiple order problem caused by repeatedly clicking "Place Order" on a checkout page. The culprit here is the delay caused by processing the payment information, which may cause the user to think that their request has not been received. The actual error is that the developer forgot to check (or properly model) the state of the Web application before processing the additional order confirmations.

In this paper we describe our experiences deriving customer-oriented acceptance tests for Web applications by modeling the essential capabilities of the application as Use Case Maps (UCMs). Our hypothesis is that this scenario-based, lightweight level of modeling is more accessible to Web developers than heavyweight formal methods. We also try to tie in the existing methodologies and tools used in the Web application development community. In particular, we use the notion of *acceptance tests* as customer-driven tests, as defined in the Test-Driven Development (TDD) approach [4] in Extreme Programming (XP). Our tests run on the popular FitNesse acceptance testing framework [18].

Section 2 presents related work. In Section 3 we describe the Web application for an online store used as a case study to demonstrate our approach. A UCM model capturing the essence of this application is presented in Section 4. In Section 5, we introduce the FitNesse-oriented testing environment used for our experiment. Test generation and results are discussed in Section 6, followed by our conclusions and an outlook on future work.

2 Related Work

Although there are many commercial tools available for testing Web applications, their scope is often severely limited. Most of these tools are designed to assess the compatibility of a Web application with different browsers and operating systems, its ability to deal with large numbers of concurrent users (stress testing), and that the application is free of dead links (link testers) [8]. However, these tools do not provide facilities for structural and functional testing.

Recently, several approaches have been developed that do not only consider the externally visible behavior of the Web application, but also its internals. Kung [17] models the state-dependent behavior of interacting components in a Web application (client pages, server pages, and software components) as hierarchical, communicating state machines. In another paper [9], Di Lucca models the behavior of a Web browser as a statechart to generate test cases which can account for out-of-order messages caused by interactions with the browser buttons. Wittevrongel [24] outlines a scenario-based approach for testing Web applications in which test cases are automatically derived from sequence diagrams. Probert [19] suggests the use of an object-oriented extension to TTCN for (manually) defining various types of tests targeting Web applications.

Use Case Maps have been used to model the dynamics of complex systems in such domains as telecommunications and e-business process modeling [1,6,23]. They are being considered for standardization as part of ITU-T's User Requirements Notation [13,22]. From the perspective of modeling Web applications, two prior approaches are of particular relevance. Kaewkasi [16] uses Use Case Maps to model object-oriented Web applications in his Web Application Modeling (WWM) approach. Gordijn [10] models business processes with Use Case Maps as part of his e^3-value approach for analyzing value creation and exchange in e-business models.

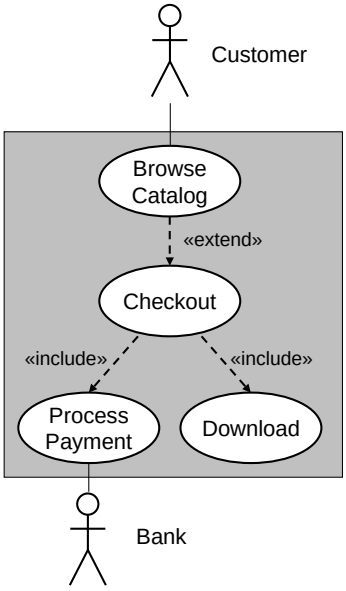
Several UCM-based testing approaches have been developed in recent years. Three main families have been compared in [3]: testing patterns (manual), scenario definitions (semi-automated), and translations to formal specifications (more automated). Among these three, scenario definitions [2] proved to be practical in many contexts as this approach is scalable, it prevents the generation of incorrect test purposes, it is supported by tools, and it generates test purposes in a XML format amenable to further transformations (e.g., to sequence diagrams or TTCN test skeletons). However, it requires human intervention for the (simple) definition of the scenarios leading to test purposes, and to add data information to test purposes in order to convert them to real test cases. In this paper, this approach will be explored in a new context where we attempt to generate concrete test cases for a Web application.

3 Web Application Case Study

The system under test (SUT) is a Web application for an online store (named `widgets.com`) where users can purchase license keys for software components (or *widgets*). It has enough non-trivial behavior to be interesting from a testing perspective, but is no more complex than necessary to exhibit several planted bugs used to validate our approach. It implements the four use cases shown in Figure 1: *Browse Catalog*, *Checkout*, *Process Payment*, and *Download*.

Browse Catalog comprises selecting categories, selecting products to request product detail, adding products to a shopping cart, and editing the cart. *Checkout* includes signing in for an account, building an order summary, and confirming the order. *Process Payment* involves asking the bank to process the payment information associated with the account. *Download* comprises going to a download area, and downloading the purchased licenses. Actors include customers and a bank. Figure 1 also shows the normal flows for the two key use cases, *Browse Catalog* and *Checkout*. The full model also includes their variants.

The application follows a standard Model-View-Controller Model 2 architecture with a main controller, and subcontrollers (request handlers) and associated views (renderers) for each use case that do the actual work [7]. The application is implemented as a Java servlet, and executes in a servlet container (in our case the Tomcat servlet engine [20]). In the case study we did not make use of a specific Web application framework such as Struts or J2EE. While such frame-



Browse Catalog
1. Customer navigates to the *widgets.com* home page.
2. System responds with a listing of categories.
3. Customer selects a category.
4. System displays a listing of all widgets in this category.
5. Customer selects a widget.
6. System responds with a product detail page for the widget.
7. Customer adds the widget to the cart.
8. System displays the updated cart.
9. Customer proceeds to checkout (see *Checkout*).

Checkout
1. Customer requests checkout.
2. System prompts the customer for his account number.
3. Customer enters account information.
4. System builds a summary of the order with totals.
5. Customer confirms the order.
6. System processes the payment (see *Process Payment*).
7. System displays invoice.
8. Customer proceeds to download area (see *Download*).

...

Fig. 1. Use case diagram for the `widgets.com` online store with two use cases

works can simplify the development of large Web applications, their use might also make our approach framework-specific, and thus less general.

Figures 2 to 4 show three typical screenshot of the online store application. In the first one, the left column of the page contains a list of available widget categories, and its center shows the contents of the shopping cart after a number of items have been added by following the *Browse Catalog* use case. Selecting "Proceed to Checkout" will terminate *Browse Catalog*, and initiate the *Checkout* use case. In that use case, the first screen (Figure 3) requires the customer to input a valid account number. Once the order is confirmed, the payment done, and the invoice displayed (not shown), the customer proceeds to the *Download* use case, where the bought widgets and license keys are available for download (Figure 4).

4 UCM Model and Scenarios

Use Case Maps (UCM) are a notation for modeling scenarios. Unlike use cases, UCMs allow us to model the dynamic behavior of an application. UCMs also allow us to model concurrency within a scenario. A single UCM can furthermore show multiple scenarios at once, and therefore allows us to study the interaction between scenarios, or multiple instances of the same scenario. However, unlike other scenario-modeling notations such as UML sequence diagrams or Message Sequence Charts (MSCs) [12], UCMs do not require an early commitment of scenarios to messages or to components.

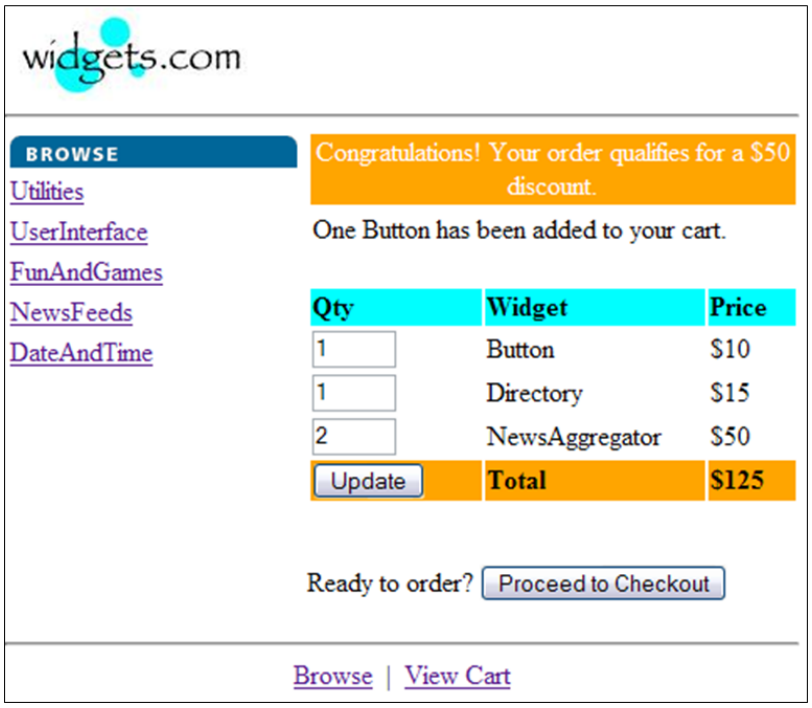


Fig. 2. Web application screenshot: Cart content while browsing

Fig. 3. Web application screenshot: Signing in with an account number

Fig. 4. Web application screenshot: Downloadable widgets with license keys

4.1 Use Case Map Model

A *scenario* is a partially-ordered set of *responsibilities* (activities, tasks, functions) that a system performs to transform inputs to outputs while satisfying certain pre- and postconditions [1]. The basic notational elements for modeling scenarios with UCMs are responsibilities (X's), paths (curved lines), start points (black dots), and end points (bars). Scenarios progress along *paths* from start to end points. Paths can fork to represent alternatives and concurrency, and also join. Responsibilities can be allocated to *components* by placing them within the boundaries of that component (rectangle). Figure 5 shows the root (top level) map for the `widgets.com` applications introduced in the case study.

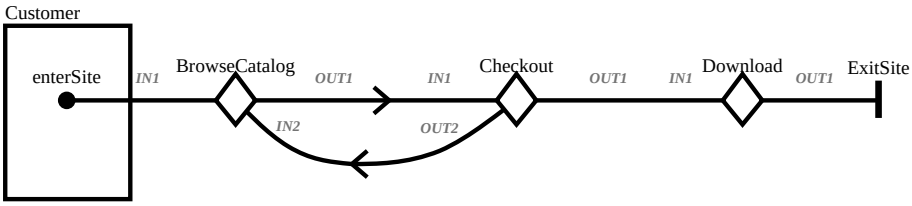


Fig. 5. Root map for the `widgets.com` online store

The UCM notation also provides a hierarchical abstraction mechanism in the form of *stubs* (diamonds) and *plug-ins* (sub-maps). Each hierarchy of maps has a *root map* that contains stubs where lower-level maps can be plugged in. Figure 5 shows the root map for the `widgets.com` applications introduced in

the case study. It contains three stubs, one for the *Browse Catalog*, one for the *Checkout*, and one for the *Download* use case. Figures 6 to 8 show the plug-ins for the BrowseCatalog, Checkout, and Download stubs from Figure 5.

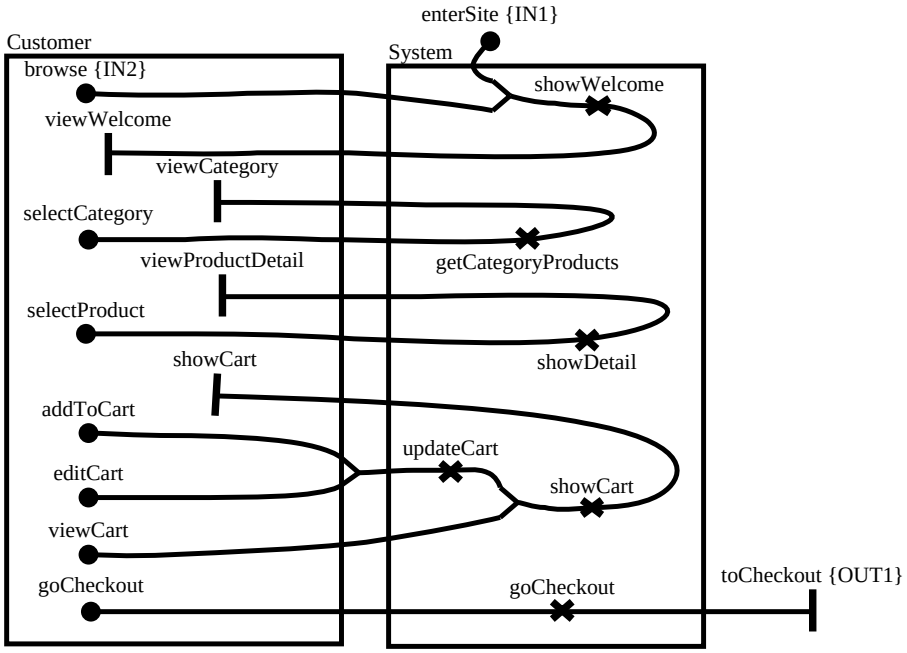


Fig. 6. Plug-in for BrowseCatalog stub in the root map of Figure 5

These stubs correspond approximately to the initial use cases and to major phases of the online store application under study. Several start and end points of the plug-ins are connected to their parent stub's input and output segments, as indicated by the corresponding labels between curly braces (e.g., IN1, OUT2). This binding relationship ensures the continuity of scenarios across different levels of maps. Note that UCM plug-ins can be nested at many levels. The Checkout map contains another stub where a ProcessPayment plug-in (corresponding to the *Process Payment* use case, but is not shown here) needs to be bound.

In this model, the start points in the Customer component correspond to events (e.g., hyperlinks and buttons) that customers can trigger. The end points correspond to page updates visible to the customers. Several responsibilities have been identified for the system, but none is assigned to its actors.

4.2 Scenario Definitions

In order to support scenario definitions, as defined in [2,22], the basic UCM path model needs to be augmented with a simple data model. Several Boolean vari-

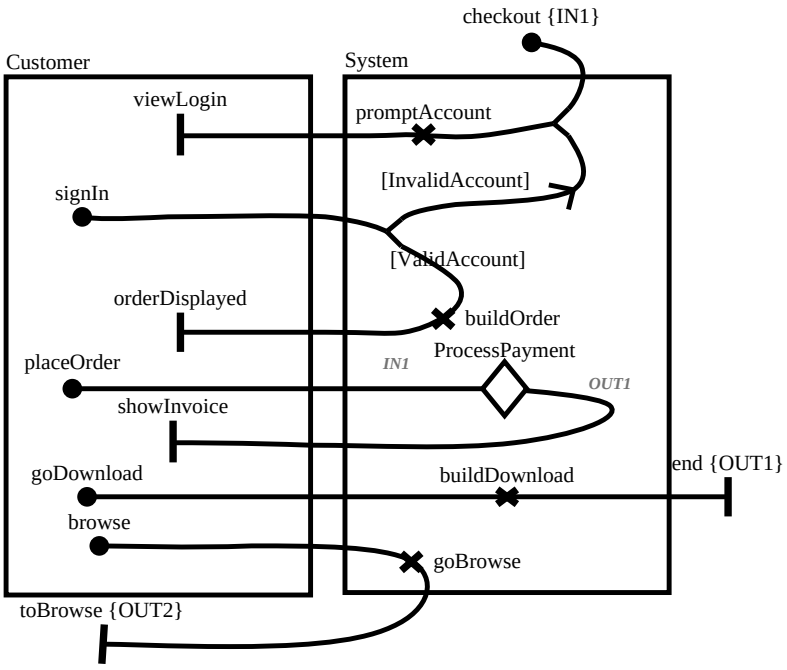


Fig. 7. Plug-in for Checkout stub in the root map of Figure 5

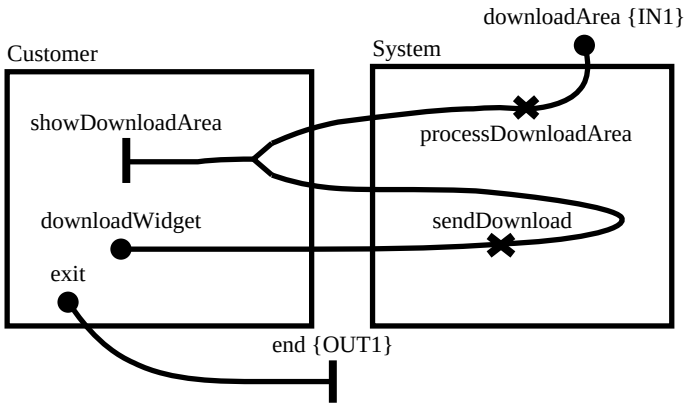


Fig. 8. Plug-in for Download stub in the root map of Figure 5

ables, described in Table 1 were created in order to formalize guarding conditions (e.g., at branching points), preconditions (in start points), and initial contexts and post-conditions in scenario definitions themselves.

Table 1. Global Boolean variables used in the UCM model

Variable	Description
CanAddProd	Products can be added on this page.
CanGoDownload	Can go to the download area.
CanPlaceOrder	An order can be placed on this page.
CanSignIn	The customer can sign in.
CartAvailable	The cart is visible.
CategoryAvailable	Categories can be selected on this page.
InBrowser	In the browser page.
InCheckout	In the checkout page.
InDownloadArea	In download area.
ProductsDisplayed	Products are displayed.
SuccessfulDownload	The download was successful.
ValidAccount	The customer account is valid.

Preconditions were added to many start points to reflect the situations under which they can be triggered. For instance, the preconditions for the start points in the BrowseCatalog plug-in map are described in Table 2. Several responsibilities in this UCM also modify the content of these variables. Table 3 shows, for the same map, how these variables are updated by the responsibilities.

Table 2. Start point preconditions and parameters for the BrowseCatalog map

Start Point	Precondition	Parameter
enterSite	–	–
browse	InBrowser	–
selectCategory	InBrowser $\wedge$ CategoryAvailable	Category
selectProduct	InBrowser $\wedge$ ProductsDisplayed	Product
addToCart	InBrowser $\wedge$ CanAddProd	–
editCart	InBrowser $\wedge$ CartAvailable	Product
viewCart	InBrowser	–
goCheckout	InBrowser $\wedge$ CartAvailable	–

Several *scenario definitions* were then added to our UCM model. Each such definition consists of a name, initial values for the variables, a list of start points to be triggered, and an optional post-condition expected to be satisfied at the end of the execution of the scenario. Scenario definitions can be combined to a path traversal algorithm in order to highlight specific scenarios in a complex UCM model, or to transform them to other representations. Details of the var-

Table 3. Variables modified by responsibilities in BrowseCatalog map

Responsibility	Modifications (T for True, F for False)
getCategoryProducts	ProductsDisplayed ← T
goCheckout	InBrowser ← F, CartAvailable ← F, CategoryAvailable ← F
showCart	CartAvailable ← T, CanAddProd ← F, ProductsDisplayed ← F
showDetail	CanAddProf ← T, ProductsDisplayed ← F
showWelcome	InBrowser ← T, CartAvailable ← F, CategoryAvailable ← T

ious algorithms used here can be found in [2,22]. In a nutshell, the algorithm uses a depth-first traversal of the graph that captures the UCMs' structure and generates scenarios where sequences and concurrency are preserved, but where alternatives are resolved using the Boolean variables. If conditions cannot be satisfied or evaluated during the traversal (e.g., in a precondition or in guarded branches), then the algorithm stops and reports an error.

In our model, we created a non-exhaustive collection of scenario definitions to cover the interesting functionalities offered by the system, as well as all the UCM path segments in the model. Although scenarios extracted from a UCM can be used for many reasons (model understanding, scenario highlight, generation of MSCs, etc.), our goal was to explore the generation of several typical test purposes for testing our Web application. In Table 4, the first four scenarios represent four normal and expected usages of the Web site whereas the last three target specific types of faults in the implementation.

Table 4. Scenarios for the `widget.com` UCM model

Scenario name	Description
BaseCase	Primary scenario where customer buys one widget and everything works.
SecondThoughts	The customer goes back to the browsing mode while checking out, in order to review the cart.
ManyProducts	Several widget products are bought by the customer.
InvalidAccount	Checks that the customer cannot download widgets without a valid account.
RemoveWidgetOnCart	The customer edits the shopping cart where a widget has been added, and sets its quantity to 0.
DiscountOnOrders	Checks whether the discount is correctly applied when widgets are removed.
MultipleOrdersTwoCustomers	Checks that two customers with the same login (from the same company) can order widgets at the same time.

As an example, the scenario definition of BaseCase is presented in Table 5. All the other scenarios are constructed in a similar way.

Table 5. BaseCase scenario definition

Initialization	ValidAccount ← T
Start points	enterSite, browse, selectCategory, selectProduct, addToCart, goCheckout, placeOrder, goDownload, downloadWidget, exit
Post-condition	SuccessfulDownload = T

The UCMNAV tool [21] was used to model this UCM and define and explore these scenarios. Each produced scenario was also automatically exported to an XML file, which uses the format described in [2]. Although these files are too verbose to be included in this paper, the result of the BaseCase scenario is shown as a MSC in Figure 10(a). Note that such MSCs were not used in this study; this one is included here to better visualize the scenario generated.

5 Test Environment

As explained in the previous sections, UCMNAV was used to create a UCM model, with variables and scenario definitions, for the target Web application. This tool was also used to generate an XML file for each scenario, hence providing the desired test purposes. Figure 9 illustrates the remaining steps of our approach. We created a small conversion application (in Perl) called UCM2FIT (Section 5.2), which converts the XML test purposes to FitNesse test cases, provided some additional information (a configuration file and user-selected values). The tests are automatically installed in the FitNesse test environment (described in Section 5.1), which requires an adaptation layer composed of *fixtures* to run them on the SUT and produce test results.

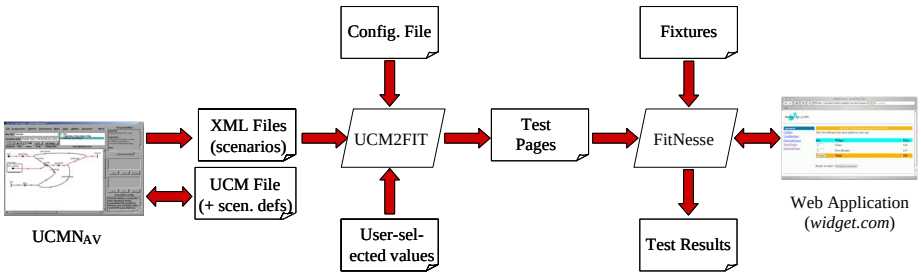


Fig. 9. Overview of the testing process used in the case study

5.1 FitNesse Framework

FitNesse is a tool popular in the Extreme Programming community [18]. It provides an environment for authoring and executing acceptance tests from within

a Web browser. It is itself based on two subsystems: a Wiki clone and FIT (Framework for Integrated Test). Wiki can best be described as a Web-based, collaborative editor. FIT is the core framework for executing acceptance tests.[3] FitNesse provides a Java-based implementation of Wiki that incorporates FIT, and can be run without a Web server. Partial support for .NET is also available.

The combination of FIT and Wiki can best be described as a "literate programming" environment for tests. Not only are documentation and tests kept in the same place, but tests can be defined in a very simple way, e.g., by creating a table. They can even be edited in Excel and copy-pasted into FitNesse. The first row of each table defines the type of *fixture* to use for the test, and the remaining rows specify the test data to be interpreted by the fixture. A fixture is the Java or C# class that FitNesse calls to process the contents of the table.

FitNesse provides a set of standard fixtures. Most relevant in our context is the *action fixture*, which allows one to emulate a user interface. It provides three types of actions to interact with an application: `press`, `enter`, and `check`. `Press` simulates pressing a button, which is mapped to invoking a method on the fixture. `Enter` is used to set a value in the fixture, and `check` tests if invoking a method of the fixture results in a given expected result. The rows of an action fixtures contain a "script" for the class specified in the second row.

For testing Web applications we created a special type of action fixture. The methods supported by the `WebFixture` are shown in Table 6. This fixture was implemented with the help of the jWebUnit framework [15]. This framework provides a high-level API for navigating Web applications. It includes navigation via links, form entry and submission, validation of table contents, etc. Behind the scenes it uses the well-known HttpUnit unit testing framework.

`WebFixture` can be used on any Web application that uses HTTP. However, in order to support the testing of a specific application (`widget.com` in our case), one can extend `WebFixture` and add methods that provide an adaptation layer which can interpret the abstract events in the test purposes to check the SUT. We hence created `WidgetFixture`, which contains a short Java methods for each start point in our UCM model. They can be more or less complex depending on what information need to be provided on a given Web page. For instance, selecting a category of widgets (start point selectCategory) on the Web page and providing an account number (start point signIn) are implemented as follows:

```
public void selectCategory(String category)
{
 linkWithText(category);
}

public void signIn(String accountNumber)
{
 form("sign-in");
```

[3] Both Wiki and FIT have, incidentally, been developed by Ward Cunningham, who is also known as the father of CRC cards, and a well-known pattern and XP guru.

Table 6. Operations supported by `WebFixture`

Loading Web pages	
base(url)	Set the base URL for relative URLs
begin(path)	Set a relative URL
Setting bookmarks	
getLocation()	Get the current URL
setLocation(url)	Request the page with the given URL
Checking page attributes and contents	
title()	Get the page title
contents()	Get the content of a page
contains(text)	Check if text is present on a given page
contains(id,text)	Check if text is present in a page element with given id
matches(pattern)	Check if the page content matches a given regular expression
Clicking links	
link(id)	Click a link with a given id
linkWithText(text)	Click a link with a given anchor text
linkWithImage(path)	Click a link in an image given its file path
button(id)	Click a button with given id
Submitting form data	
form(id)	Set the working form given its id
formElement(name,value)	Set the value of a field given its name
submit()	Submit a form
submit(button)	Submit a form by pressing the given button
reset()	Reset a form

```
  formElement("account", accountNumber);
  submit();
 }
```

5.2 UCM2FIT

As introduced in Figure 9, the goal of UCM2FIT is to convert a XML scenario file generated by UCMNav to a FitNesse test case. However, this cannot be done entirely automatically. Part of the information that needs to be added to the test purposes can be provided in advance, for instance in a configuration file, but the information related to the selection of values is currently provided interactively by the tester, during the transformation.

The configuration file (also in XML) contains the following information:

- The target directory of the test suite, in a place where FitNesse can find it automatically.
- Test setup information (e.g., path to FitNesse and fixture classes)
- Data types (e.g., categories, products, and account numbers), together with sample values. These correspond to items that the tester can select interactively when requested by UCM2FIT.
- For each end point in the UCM, a list of text items (information) to be checked on the Web page.

As an example of information associated to an end point, what is specified for viewWelcome is a page title and a well-known text pattern that does not appear on the other pages:

```
<endpoint name="viewWelcome">
   <check type="title" value="widgets.com"></check>
   <check type="pattern" value="Welcome to widgets.com"></check>
</endpoint>
```

This information is used to generate appropriate verification code in the FitNesse tests each time an end point is mentioned in the test purposes.

When UCM2FIT gets to a start point with parameters in a test purpose (see Table 2), the human tester is interactively asked to provide a value of that type among those proposed in the configuration file. For instance, for start point selectCategory, the value NewsFeeds could be selected. Care must be taken to select appropriate values so that the test purpose can progress. For instance, if the test purpose presupposes that the account number to be provided will be valid, then the selected value must be consistent with this assumption.

Similarly, several end points have been supplemented with output parameters in the UCM model (something new for this experiment). For example, the end point showCart has a parameter that corresponds to the cost of the selected product, which needs to be input by the tester. This cost is checked against the one displayed on the Web page.

This process is performed once for each test purpose, and it is usually a matter of seconds to produce one test. The output is a collection of executable tests coded as (textual) Wiki pages and understandable by FitNesse, together with an index that corresponds to a test suite. This test suite enables FitNesse to check all the test cases, in batch, and to provide a summary of the results. Note that multiple test cases could be generated from one test purpose by selecting different combinations of values (however, this was not done for this experiment).

Figure 10(b) presents the test case generated from the BaseCase test purpose (see Table 5), in a tabular form displayed by the Web browser. Note how the `press` actions correspond to the start points and how the `enter` and `check` operations are used to verify that the output from the SUT is correct. A recurring pattern is that an `enter` method specifies an expected value (e.g., the `patternToMatch`), and a subsequent `check` asserts whether the actual value should (or should not) correspond to (e.g., `matches`) the expected value.

UCM2FIT is currently implemented with standalone, command-line Perl scripts and a `XML::DOM` parser. However, this functionality could be integrated into FitNesse itself in the future.

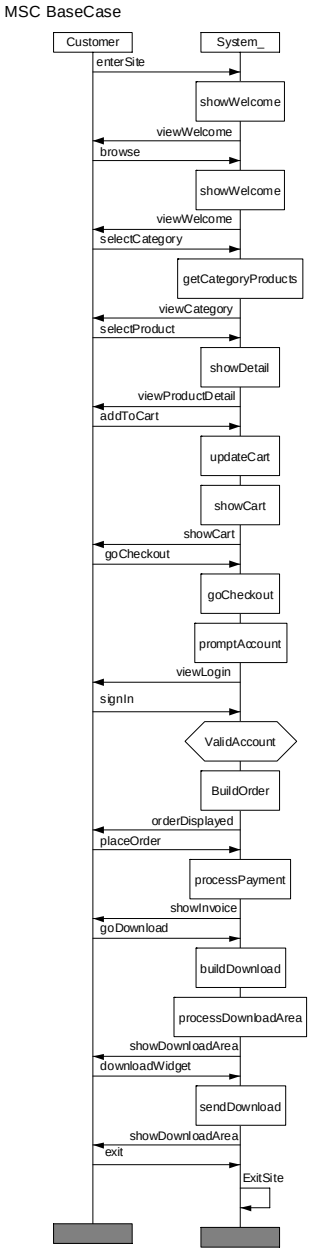
6 Test Execution and Results

As a measure of the effectiveness of our tests, a number of bugs were "planted" into the Web application. These are all major functional errors that affect the function of the application in a significant manner. For example, if a user account was charged for a widget that they did not order, we consider this a major error. Minor errors are errors that lead to locally incorrect behavior, but do not significantly impact the execution of the application. For example, entering a non-digit as an account number may cause an exception to be thrown, but it does not lead to an inconsistent state. Below is the list of planted errors:

1. Edits to the shopping cart are not properly updated.
2. Discount ($50 when the total is greater than $100) is incorrectly applied.
3. Race condition: Only the most recent order is shown in the download section.
4. State-/Timing-related: Can add widgets to the cart for free.

Two of these bugs (#3 and #4) require a fault model where many customers, or many sessions (of the same customer) are active at the same time. In UCM terms, this means that the same start point (e.g., enterSite) can be triggered multiple times in order to simulate multiple widget buying sessions. This is also supported by our approach based on scenario definitions.

The results generated by our test suite are reported in Figure 11, which shows the summary produced by FitNesse. In the summary table, tests that completed without failures are shown in green / light grey (pass), while those with failures are shown in red / black (fail). As expected, the first four tests from Table 4 did not reveal any problem in the SUT. These tests validate the intended functionalities of the application, without looking for subtle types of errors.



(a) MSC view

fit.ActionFixture		
start	ucm2fit.WidgetFixture	
check	title	widgets.com
enter	patternToMatch	Welcome to widgets.com.
check	matches	true
press	browse	
check	title	widgets.com
enter	patternToMatch	Welcome to widgets.com.
check	matches	true
enter	selectCategory	Utilities
enter	patternToMatch	Select from these sumptuous
check	matches	true
enter	selectProduct	Directory
enter	patternToMatch	discount on orders over
check	matches	true
press	addToCart	
enter	patternToMatch	Ready to order?
check	matches	true
enter	patternToMatch	15
check	matches	true
press	goCheckout	
enter	patternToMatch	Enter your 4-digit account number
check	matches	true
enter	signIn	1234
enter	patternToMatch	Please review and submit your order.
check	matches	true
enter	patternToMatch	Directory
check	matches	true
press	placeOrder	
enter	patternToMatch	Please proceed to the
check	matches	true
press	goDownload	
enter	patternToMatch	Download the purchased widgets
check	matches	true
enter	patternToMatch	Directory
check	matches	true
enter	downloadWidget	Directory
enter	patternToMatch	Download the purchased widgets
check	matches	true

(b) Test case

Fig. 10. MSC for the BaseCase scenario and its corresponding FitNesse test.

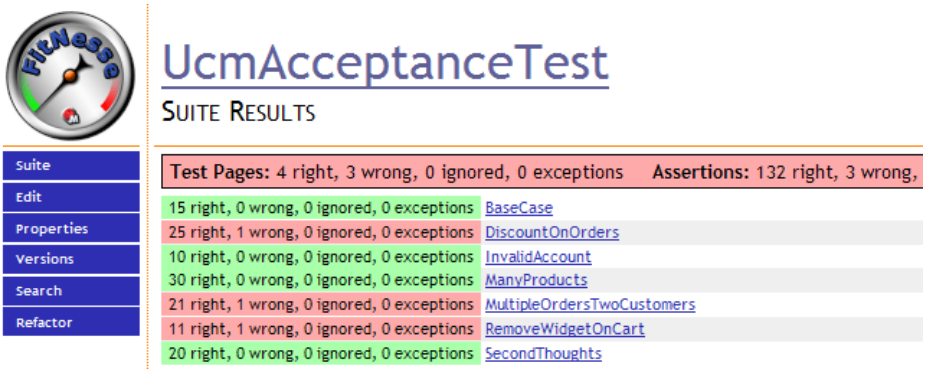


Fig. 11. Summary of FitNesse test results

However, the last three tests of Table 4 revealed three of the four planted bugs. RemoveWidgetOnCart showed that when the quantity of a product is set to 0 while editing a cart, then this quantity is not updated correctly (bug #1). In FitNesse, such an error is reported as a violation of an assertion. Figure 12 shows how FitNesse displays the test detailed test results for RemoveWidgetOnCart. Correct assertions are shown in light grey while incorrect ones are displayed in black together with the expected and actual values.

	...	
enter	selectProduct	CpuMeter
enter	patternToMatch	discount on orders over
check	matches	true
press	addToCart	
enter	patternToMatch	Ready to order?
check	matches	true
enter	editCart	CpuMeter
enter	patternToMatch	Ready to order?
check	matches	true
enter	patternToMatch	CpuMeter
check	matches	**false *expected*** / **true *actual***

Actual result is different from expected one! The quantity of CpuMeter was set to 0 while editing the cart, yet this product is still listed (a false match was expected).

Fig. 12. Extract of the RemoveWidgetOnCart test result

The test DiscountOnOrder adds items to the cart in excess of $100, and then removes an item. It revealed that when the total cost of the products in the cart gets over $100, the $50 discount is correctly applied but, as widgets are subsequently removed, the discount is (incorrectly) not recomputed. As a result,

a discount may be applied, although the actual total may no longer be above $100. This state-related *stuck at fault* corresponds to bug #2.

The last test, MultipleOrdersTwoCustomers, checks the situation where two customers sharing the same account number try to order widgets simultaneously. Unfortunately, the implementation suffers from a race condition problem (bug #3) and the content of only one of the carts is shown for download to the customers. This scenario is interesting because it required the multiple triggering of the initial start point, enabling two orders to evolve concurrently.

Bug #4 was not revealed by our test suite. In fact, we could not produce a UCM-driven test case for it with the current environment. A more sophisticated fixture supporting forking seems required, and our UCM model needs to be able to simulate out of order messages more easily (for example, "add to cart" after "place order"). Such improvements are left for future work.

7 Discussion and Conclusions

In this paper we have described an innovative and lightweight approach for generating acceptance tests for Web applications from a Use Case Maps (UCM) model. Through a case study, we demonstrated that this approach is capable of detecting subtle design errors that often go unnoticed using conventional acceptance testing techniques such as *extended use cases* [5]. We believe that this is the first paper to introduce UCM-based testing in this emerging domain. Another contribution of the paper is to show how abstract test sequences generated from UCMs can be transformed into test cases in the FitNesse testing framework.

The UCM model presented here uses an unconventional style where many start points, capturing almost one-to-one the possible user events (hyperlink, form, or button), are at the source of disjoint paths. Preconditions insure that they can only be triggered when they the application should allow it. This is a benefit of the emphasis on the UML data model, but this is also a drawback because, from a testing point of view, we would also like to test scenarios where such events are provided in an *incorrect* or unanticipated order, as one would do using the back or forward buttons on a browser, bookmarks, or direct URLs. This is also a limitation of other Web testing approaches, including [8]. A tool like UCMNav could be extended to allow some flexibility in the checking or bypassing of these preconditions during scenario generation.

One of the main issues to be addressed in the future relates to how best to provide suitable data values during the transformation from abstract test purpose to concrete test case. At the moment, one would need to input such values each time a scenario is modified. Some of these values could be inferred from the scenario preconditions (e.g., a valid account number when ValidAccount is true), others could come from predefined equivalence classes where the values correspond to the ones the SUT would expect (e.g., using some shared database for the test setup). Previous choices of values could also be stored independently and reused whenever possible.

This experiment also raised a few interesting points related to the UCM notation. It seems that being able to provide formal parameters to start points and end points is very beneficial in a testing context. Also, current scenario definitions focus solely on start points, whereas there might be a need for intermediate assertions of end point values in the middle of a scenario, and not just at its end.

Another area for future research is a closer integration of UCMs with Extreme Programming (XP), given their relatively lightweight nature compared to "heavier" approaches such as MSCs, which require and early commitment to messages or components. Also the extension of the current fixtures for testing more generic types of Web applications should be investigated in the future.

Acknowledgments

This research was supported by the Natural Sciences and Engineering Research Council of Canada, through its programs of Strategic Grants and Discovery Grants.

References

1. Amyot, D.: Introduction to the User Requirements Notation: Learning by Example. Computer Networks, 42(3), 285–301, June 2003.
2. Amyot, D., Cho, D.Y., He, X., and He, Y.: Generating Scenarios from Use Case Map Specifications. Third Int. Conf. on Quality Software (QSIC'03), Dallas, USA, Nov. 2003, 108–115.
3. Amyot, D., Logrippo, L., and Weiss, M.: UCM-Based Generation of Test Purposes. To appear in Computer Networks, 2005.
4. Beck, K.: Test-Driven Development By Example. Addison-Wesley, 2004.
5. Binder, R.: Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley, 2000.
6. Buhr, R.J.A., and Casselman, R.S.: Use Case Maps for Object-Oriented Systems. Prentice Hall, 1996. `http://www.usecasemaps.org/pub/UCM_book95.pdf`
7. Burke, E.: Java and XSLT: Embedding XML Processing into Java Applications. O'Reilly, 2001.
8. Di Lucca, G., Fasolino, A., Faralli, F., and de Carlini, U.: Testing Web Applications. 18th Int. Conf. on Software Maintenance (ICSM), 2002, 310–309.
9. Di Lucca, G., and Di Penta, M.: Considering Browser Interaction in Web Application Testing. 5th Int. Work. on Web Site Evolution, 2003, 74–81.
10. Gordijn, J., and Akkermans, H.: Designing and Evaluating E-Business Models. IEEE Software, July/August 2001, 11–17.
11. Hieatt, E., and Mee, R.: Going Faster: Testing the Web Application. IEEE Software, March/April 2002, 60–65.
12. ITU-T – International Telecommunications Union: Recommendation Z.120 (04/04) Message Sequence Chart (MSC). Geneva, Switzerland, 2004.
13. ITU-T – International Telecommunications Union: Recommendation Z.150 (02/03), User Requirements Notation (URN) – Language Requirements and Framework. Geneva, Switzerland, 2003.
14. Jacques, R.: Web Applications Wide Open to Hackers. vnunet.com news, `http://www.vnunet.com/News/1152521`, Feb 5, 2004.

15. jWebUnit: http://jwebunit.sourceforge.net
16. Kaewkasi, C., and Rivepiboon, W.: WWM: A Practical Methodology for Web Application Modeling. 26th Int. Computer Software and Applications Conf. (COMPSAC), 2002, 603–609.
17. Kung, D., Liu, C., and Hsia, P.: An Object-Oriented Web Test Model for Testing Web Applications. 24th Int. Computer Software and Application Conf. (COMPSAC), 2000, 537–542.
18. Martin, R., and Martin, M.: FitNesse Web Site, http://www.fitnesse.org
19. Probert, R.L., Xiong, P., Stepien, B.: Life-Cycle E-commerce Testing with OO-TTCN-3. M. Núñez, Z. Maamar, F.L. Pelayo, K. Pousttchi, F. Rubio (Eds.): Applying Formal Methods: Testing, Performance and M/ECommerce, FORTE 2004 Workshops The FormEMC, EPEW, ITM, Toledo, Spain, October 1-2, 2004. Lecture Notes in Computer Science 3236, Springer 2004, 16–29.
20. Tomcat: http://jakarta.apache.org/tomcat
21. UCM User Group: UCMNav 2, http://www.usecasemaps.org/tools/ucmnav/
22. URN Focus Group: Draft Rec. Z.152 – Use Case Map Notation (UCM). Geneva, Switzerland, September 2003. http://www.UseCaseMaps.org/urn/
23. Weiss, M. and Amyot, D.: Business Process Modeling with URN. International Journal of E-Business Research, 1(3) 63-90, July-September 2005.
24. Wittevrongel, J., and Maurer, F.: SCENTOR: Scenario-Based Testing of E-Business Applications. Int. Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2001, 41–46.