Model-Based Testing of Web Applications using NModel

Juhan Ernits[1], Rivo Roo[2], Jonathan Jacky[3], and Margus Veanes[4]

[1] University of Birmingham, UK
j.ernits@cs.bham.ac.uk
[2] Reach-U Ltd,Tartu, Estonia
rivo.roo@reach-u.com
[3] University of Washington, Seattle, WA, USA
jon@u.washington.edu
[4] Microsoft Research, Redmond, WA, USA
margus@microsoft.com

Abstract. We show how model-based on-the-fly testing can be applied in the context of web applications using the NModel toolkit. The concrete case study is a commercial web-based positioning system called WorkForce Management (WFM) which interacts with a number of other services, such as billing and positioning, through a mobile operator. We describe the application and the testing, and discuss the test results.

1 Introduction

In model-based testing, test cases are generated automatically from a model that describes the intended behavior of the system under test [1, 4, 2]. This contrasts with conventional unit testing, where the test engineer must code each test case. Therefore model-based testing is recommended where so many test cases are needed to achieve adequate coverage that it would be infeasible to code them all by hand.

Model-based testing is especially indicated where an implementation must support ongoing data- and history- dependent behaviors that exhibit nondeterminism, so that many variations (different data values, interleavings etc.) should be tested for each scenario (or use case). Web applications are examples. A web application is a program that communicates with a client using the HTTP protocol, and possibly other web protocols such as SOAP, WebDAV etc. Here nondeterminism arises from the multiple valid orderings of observable messages that may be returned by the system under test (SUT) in response to some set of stimuli sent by the tester. Such orderings depend on many factors such as for example load of certain components in the system or utilisation of the network links.

We present a nontrivial case study of applying on-the-fly model-based testing on a component of a distributed web application - a web-based positioning system developed by Reach-U Ltd called WorkForce Management (WFM). The purpose of the system is to allow subscribers to track the geographical position of their employees by mobile phones for the purpose of, for example, improving the practice of a courier service. In addition to the web interface, the system interfaces with a number of software components of the mobile operator, for example billing and positioning systems.

There are many model-based testing tools, see [4] for an overview. For this study we chose the open-source NModel testing framework, where the models are expressed in C# and web services are accessed with .NET libraries [2]. NModel supports on-the-fly testing (section 5) and uses composition of models for both structuring and scenario control (section 2).

2 Model programs

In model-based testing the model acts as the test case generator and oracle. In the NModel framework, the model is a program called a *model program*. Test engineers must write a model program for each different implementation they wish to test.

Model programs represent behavior (ongoing activities). A *trace* (or run) is a sample of behavior consisting of a sequence of *actions*. An *action* is a unit of behavior viewed at some level of abstraction. Actions have names and arguments. The names of all of a system's actions are its *vocabulary*. For example some of the actions in the system we tested are `Initialize`, `WebLogin_Start` and `WebLogin_Finish`.

Traces are central; the purpose of a model program is to generate traces. In order to do this, a model program usually must contain some stored information called its *state*. The model program state is the source of values for the action arguments, and also determines which actions are enabled at any time.

There are two kinds of model programs in NModel:

Contract model program is a complete specification (the "contract") of the system it models. It can generate every trace (at the level of abstraction of the model) that the system might execute including known faults and appropriate responses to those.

In the NModel framework, contract model programs are usually written in C#, augmented by a library of modeling data types and attributes. The valuations of variables of the model program are its state. The actions of the model program are the methods labeled with the `[Action]` attribute (in C#, an *attribute* is an annotation that is available at runtime). For each action method, there is an *enabling condition*: a boolean method that returns true in states where the action is enabled (the enabling condition also depends on the action arguments).

Scenario model program constrains all possible runs to some interesting collection of runs (subset of all allowed behaviours) that are related in some way. Scenario model programs can be thought of as abstract test cases as it is possible to specify certain states and the sequence in which the states need to be traversed while leaving the intermediate states to be decided by the contract model program.

To enhance maintainability and keep a close match between the textual specification and the formalized model, we make heavy use of the composition facilities provided by NModel [6]. *Composition* of model programs enables one to build up larger models from smaller ones, and to restrict larger models to specific scenarios. Composition combines two or more model programs into a new model program called the product. Composition synchronizes shared actions (that appear in more than one of the composed programs) and interleaves unshared actions (that appear in only one). The composition facilities allow us to split separate actions but also separate functionality

performed by the same actions into separate classes annotated with the `[Feature]` attribute. We distinguish contract features and scenario features, where the former define specified behaviour and the latter constrain it in some interesting way determined by the test designer.

The specification was grouped into features which represent logically tightly related functionality. We modeled $Login$, $LogOff$, $Positioning$, $BillingAndHistory$ and $Restart$ as separate *features* in NModel [6] that when composed specify the *contract model program* of the positioning functionality of WFM.

To focus the test on some particular part of the system we used different compositions of components. For example for focusing the test just at the login functionality we instantiated a model by composing $Login \oplus Logoff$. For the purpose of testing the positioning functionality, we restricted the logging in and logging off by scenario features. The $OneLogin$ feature restricts new login attempts by a user when already logged in. This enabled us to keep the tests focused on the positioning functionality while letting the developers investigate a potential issue with the login functionality that the tests revealed. The $CorrectPassword$ feature is used to further constrain the login to try only correct passwords.

3 Test setup

To automatically test a web application, replace the usual client (for example, a user with a web browser) with a test tool. The tool generates test cases that correspond to client behaviors, including sending requests to the server, and checks the server's responses to determine whether it passes or fails each test.

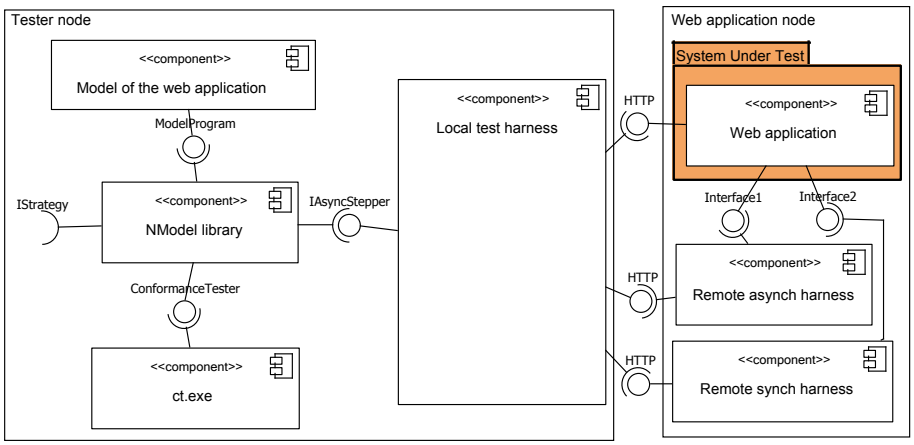

Fig. 1. A SysML component diagram of the setup of model-based testing of a web application.

The test setup that is used throughout the current paper is given in Fig. 1. The model resides in the component called the "Model of the web application".

The `IAsyncStepper` interface, which stands for "asynchronous stepper", supports mapping synchronous actions from the model to the actual implementation and vice versa and mapping asynchronous actions to action terms in the appropriate observer queue (see figure). Synchronous actions correspond to the behavior where the action blocks until the response is received. In the web application setting this corresponds to sending a GET or POST query to the web application from the client and waiting for the response that is received as the result.

Such actions can also be split into a pair of split actions defining action start and action finish separately. This enables to split the controllable part of invoking the action and the observable part of receiving a response to be split and allow other actions to take place during the time in between. This requires registering observers for asynchronous observable actions which are invoked by the web application. A concrete example of such an action is the event where the record of a completed transaction reaches the log database. It should not occur when no transaction has completed but can occur with a certain delay depending on the performance of the subsystems involved in completing the transaction. The implementation of the `IAsyncStepper` interface provides means for passing such messages on to the model after they have been converted to the abstraction level of the model in the harness.

The division of the components into tester node and web application node in Fig. 1 is based on how the test system is deployed. The tester can perform tests of the web application over the network, but it may be required to have some remote components of the test harness reside in the same node as the web application, as explained later.

4 Test harness

The connection of the model programs to the actual web application requires building a *test harness* (sometimes also called an *adapter*). The test harness defines parameterized HTTP-queries and expected answers to the queries, and sets them into accordance with transitions in the model using the `IAsyncStepper` interface. The intuition is that the test harness makes the model and the system work in lock step.

A concrete test harness is generally application specific. We were able to use the .NET libraries for producing HTTP queries. Thus building support of cookies, GET and POST queries into the test harness required only a modest amount of work. We used packet capturing methods for acquiring queries that were added to the harness in an appropriately parameterized form.

Some observations, which are important from the point of view of determining the verdict of test runs made on some internal ports of the system, are intercepted by a custom script which triggers callbacks to our test harness.

5 Running the tests

The tests were run with different compositions of features. For example, we used the following compositions for testing ($OneLogin$ and $CorrectPassword$ are abbreviated as OL and CP respectively):

1. $Login \oplus Logoff$
2. $Login \oplus Positioning \oplus OL \oplus CP$
3. $Login \oplus Positioning \oplus BillingAndHistory \oplus OL \oplus CP$
4. $Login \oplus Positioning \oplus BillingAndHistory \oplus Restart \oplus OL \oplus CP$

We carried the tests out in two main phases according to the progress of the development of the test harness. In the first phase we ran the tests using the web interface of the system and in the second phase we incorporated server side test adapter components to communicate additionally via the Billing and History ports of the system.

We used *on-the-fly* testing in this study. This contrasts with offline test generation, where test cases are generated in advance and stored until test execution. In that case, the generated test suites can be quite large, because they can include many similar sequences that are allowed by nondeterminism, but are never executed (because the system under test does not choose them at test execution time). On-the-fly testing dispenses with computing test cases in advance, and instead generates the test case from the model as the test executes, considering only the sequences that are actually executed. When testing in this way, conformance is based on establishing the alternating refinement relationship between the application and the model program, as discussed in [5].

6 Results

Table 1 contains a summary of the proportion of time and effort spent on building the model-based test system of WFM. Lines of code include comments. It shows that about 40% of time was spent on modeling and 60% on building various components of the test harness.

Several of the components developed in this case study are generic, independent of WFM, and can be reused for similar applications.

Table 1. Time and effort spent on model based testing the WFM system

Part of the test system	Lines of code	Time (%)
Model	1000	40
Asynchronous test harness local to the tester	850	38
Web server in the test harness	200	15
Restart module	125	4
Modifications in WFM code	125	3
Total	2300	100

These test failures occurred:

1. While running $Login \oplus Logoff$ tests an "Internal Server Error" message sometimes occurred after entering the correct user name and password.

2. Running $Login \oplus Positioning \oplus OL \oplus CP$ revealed a situation where the after receiving a positioning request the server kept replying that there are no new positioning results.
3. Running $Login \oplus Positioning \oplus OL \oplus CP$ causes the system sometimes to return an "Internal Server Error" message.
4. Running $Login \oplus Positioning \oplus BillingAndHistory \oplus Restart \oplus OL \oplus CP$ reconfirmed that all positioning requests that had not completed by the time of restart were lost. After the system came back up, the positioning requests remained unanswered. Deployed WFM systems work in a cluster and if one server goes offline the work is carried on by another server. Our test setup did not include the cluster.

The functionality of the WFM system had previously been tested by a 3rd party using JMeter [3]. Model-based testing found errors in that were not discovered using that tool. It also provided more flexibility for automated testing.

As the model based testing toolkit is also a piece of software, our experiment revealed two errors in NModel. Both were fixed in the toolkit.

7 Conclusions

The WFM case study showed that model-based tests exposed errors in the web application that were not exposed by conventional methods.

It is possible to build a domain-specific web service testing application with a modest effort, roughly a few man-months.

The main drawbacks were related to complexities in building the test harness and a relatively steep learning curve. The functionality that was tested is a fairly small fraction of the overall functionality of the WFM system.

References

1. M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems, Advanced Lectures [The volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004]*, volume 3472 of *Lecture Notes in Computer Science*. Springer, 2005.
2. J. Jacky, M. Veanes, C. Campbell, and W. Schulte. *Model-based Software Testing and Analysis with C#.* Cambridge University Press, 2008.
3. JMeter. http://jakarta.apache.org/jmeter/, accessed in May 2009.
4. M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan-Kaufmann, 2006.
5. M. Veanes, C. Campbell, W. Schulte, and N. Tillmann. Online testing with model programs. *SIGSOFT Softw. Eng. Notes*, 30(5):273–282, 2005.
6. M. Veanes and W. Schulte. Protocol modeling with model program composition. In K. Suzuki, T. Higashino, K. Yasumoto, and K. El-Fakih, editors, *FORTE*, volume 5048 of *Lecture Notes in Computer Science*, pages 324–339. Springer, 2008.