Model-Based Testing of Web Applications using NModel

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

[1] School of Computer Science,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 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). WFM allows subscribers to track the position of their employees for the purpose of, for example, improving the practice of a courier service. WFM interacts with other services, such as billing and positioning, through a mobile operator. NModel is a lightweight toolkit based on C# that can be used across different platforms for establishing such a domain-specific testing application. Models are written using C# and online tests are generated automatically from the models that are also used as oracles for determining the test verdicts. The main goal was to test functional correctness of WFM under normal circumstances and during startup and shutdown. We describe an overview of the architecture of this application and discuss concrete test results.

1 Introduction

In model-based testing, test cases are generated automatically from a model that describes the intended behavior of the implementation under test [3, 21, 10]. 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 services are examples, where the nondeterminism arises from user (client) behavior and vicissitudes in the network and servers.

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.

In offline test generation, test cases are generated in advance and stored until test execution. The generated test suites can be quite large, because they can include many similar sequences that are allowed by nondeterminism. In the case of online test generation, sometimes called on-the-fly testing, testing dispenses with computing test cases in advance, and instead generates the test case from the model as the test executes. On-the-fly testing can be random or can be guided by a programmed strategy that seeks to probe particular behaviors or maximize a coverage measure. By non-determinism we mean multiple valid orderings of observable messages 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.

In this paper we demonstrate these techniques with the open-source NModel testing framework, where the models are expressed in C# and web services are accessed with .NET libraries [10]. The example discussed here is a web based positioning application. We present an extensive overview of the functional requirements of the system to demonstrate how the narrative is converted into model programs and give the user some detailed intuition about the system.

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 called WorkForce Management (WFM). The purpose of the system is to allow subscribers to track the 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.

The goal of the application of model-based testing in this case study is to test the functional behaviour of the WFM under normal circumstances and during startup and shutdown. For conformance testing the normal behaviour of the system, it is accessed through the web interface, as a regular web browser client would. Some observations, which are important from the point of view of determining the verdict of test runs are made on some internal ports of the system which are intercepted by a custom script which triggers callbacks to our test harness – the adapter between the system and its model.

During startup and shutdown there are some permissible faults, like denying access to a user with a valid password. On the other hand, there are some faults which should not occur, like debiting an account of a user and not providing the service. Both kinds of errors are modeled in the model and can thus be detected and distinguished by model-based testing.

For modeling we use *model programs*. Model programs is a useful formalism for modeling of software and for design analysis. Model programs are used as the foundation in tools such as Spec Explorer [23], Spec Explorer 2007 [7] and NModel [10][5]. The models and the test harness are implemented in the C# language and are compiled into an intermediate language that runs on several platforms such as Mono $\geq$1.2.6 and Microsoft .NET $\geq$2.0.

[5] Available from http://nmodel.codeplex.com

2 Testing web applications with NModel

A web application is a program that communicates with a client using the HTTP protocol, and possibly other web protocols such as SOAP, WSDL, etc. Our sample is fragment of a Workforce Management System which enables the operator to position the employees using triangulation provided by mobile service providers. Before delving into the details of the concrete application, we provide a description of the general test setup that could be of interest for testers of other web applications.

2.1 Model programs

In model-based testing the model acts as the test case generator and oracle. In the NModel framework, the model is program called a *model program*. The user must write a model program for each different implementation he wishes to test.

```
enum ControlMode { Initalizing, Running }
enum User { user1, user2 } //user3 ...

class Common {
  ControlMode state = ControlMode.Initalizing;
  Set<User> usersLoggedIn = Set<User>.EmptySet;
  Map<User,Set<PersonNumber>> userNumbers=Map<User,Set<PersonNumber>>.EmptyMap;
  Map<User, int> userPersonCount = Map<User, int>.EmptyMap;

  [Action]
  void Initialize() { state = ControlMode.Running; }

  bool InitializeEnabled() { return state == ControlMode.Initalizing; }
}
```

Fig. 1. Shared variables and initialization action of the model of the application detailed in Sec. 3.

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`. For example, the `Initialize` action, the corresponding action guard and the specification of global variables common to different subcomponents of the model built for the case study detailed in Section 3 are given in Fig. 1[6].

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. For example there is a state variable called `state` that represents the control mode of the model program in Fig. 1.

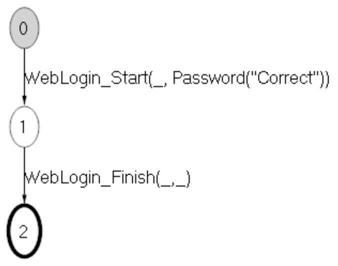
[6] In this and all further C# examples we have omitted the keywords `public` and `static` from variable, action and action guard declarations to improve readability. It should be noted that it is possible to give the model programs in a less verbose formalism, e.g. AsmL. We have chosen to keep all models in C# as they were used in the case study and we kindly ask the reader to bear with us and notice the abstract models behind the programming language syntax.

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.



```
FSM(0,Accepting(2),Transitions(
  t(0,WebLogin_Start(_,
      Password("Correct")),1),
  t(1,WebLogin_Finish(_,_),2)
),
Vocabulary("WebLogin_Start",
      "WebLogin_Finish"))
```

Fig. 2. A login scenario restricting the behaviour to only trying correct passwords.

In the NModel framework scenario model programs can be specified as Finite State Machines like the logging in scenario restricting the login to only trying the correct password in Fig. 2. The scenario model programs can also be specified as C# model programs.

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 [25]. The facilities allow 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 semantics of model programs can be given in terms of labelled transition systems (LTS) and their composition can be explained in terms of lazy automata theoretic composition of the underlying LTSs where actions are composed by unification [25].

The goal of modelling the web application using model programs is to eventually perform online testing of the system to establish the formal conformance relationship between the model and the part of the web application that we are interested in. The conformance is based on establishing the alternating refinement relationship between the application and the model program, as shown in [24].

2.2 Test setup

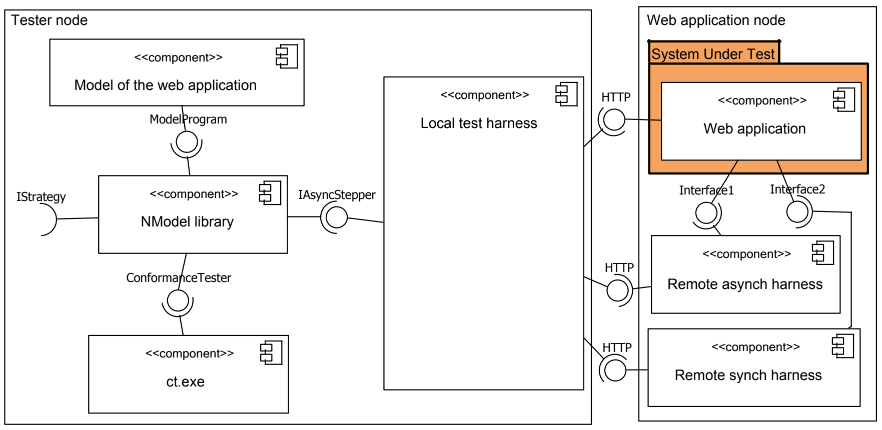


Fig. 3. A SysML component diagram of the setup of model-based testing of a web application.

The test setup that is used throughout the current paper is given in Fig. 3. The model resides in the component called the ”Model of the web application”.

The NModel library is central to the toolkit. It provides the tools for loading and stepping through (generating traces of) the specified model program according to the selected functionality. In the test setup in Fig. 3 the library’s `ConformanceTester` functionality is used which is invoked by the corresponding command line tool `ct.exe`. The conformance testing procedure loads the model and the test harness that is called via the `IAsyncStepper` interface defined in the NModel library.

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. 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 page 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 NModel library also provides an explicit interface `IStrategy` for defining various strategies to traverse the model. In the current case study we used the random strategy built into the conformance tester functionality of NModel and guided the tests using pruning of the state space by scenario component model programs.

The division of the components into tester node and web application node in Fig. 3 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.

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 used .Net libraries for producing HTTP queries. Thus building support of cookies, GET and POST queries into the test harness required fairly moderate amount of work. There is a WebApplication sample among other samples available from the NModel web page.

A typical web application involves several subcomponents like, for example, a database, or is indeed a composition of different services which are not directly visible from the web interface. Thus there may be a need to intercept messages on other interfaces of the web application than the HTTP frontend to test a well defined part of the system. Intercepting such messages is application specific, but are captured by the ”Remote asynchronous harness” component in Fig. 3. The component listens to some internal interface indicated by ”Interface1” in the figure and passes the actions, when detected, on to the ”Local test harness” by invoking a HTTP query into a web server embedded into the latter. We used an open source C# web server[7].

It may also be required to be able to change the operating mode of the web application in a way that is possible only locally to the web application. Such behavior, for example invoking a restart of the web application, is supported by the ”Remote synchronous harness” component in Fig. 3 and involves coding the component up as a web service that can be called from the test harness local to the tester.

Capturing and parameterization of the HTTP queries and regular expressions for the responses is a task that we found to be time consuming and tedious. As part of the future work we plan to use support provided by specialized web application testing tools line JMeter and Selenium that to semi-automate recording the queries and analyzing responses. Those tools themselves do not support model-based testing.

JMeter [11] is a web application testing toolkit that can be used for recording message sequences from a connection between the browser and the server, save them and then play them back later. The recording functionality of JMeter is implemented as an

[7] Available from http://webserver.codeplex.com/.

intercepting proxy. Selenium [18] is a toolkit that runs with Firefox web browser and enables to record sequences of actions like button presses and dropdown selections. Both of these toolkits are relevant in the future work to solve the problem of obtaining concrete instances of messages that correspond to actions in the specification. While JMeter can be used for obtaining instances of HTTP messages, Selenium can be used to refer to UI elements like `ui=loginPage::loginButton()`.

Creating and running model-based tests using NModel toolkit consists of the following steps:

1. Choosing the functionality to be tested and specifying the requirements.
2. Building a model in C#. The model describes the behaviour of system that we want to test.
3. Checking and validating the model visually, using the Model Program Viewer tool. It is also possible to perform some reachability analyses of the model program with a Model Checker tool.
4. Building the adapter. The adapter needs to implement either the `IStepper` or the `IAsyncStepper` interfaces of NModel.
5. If offline testing is used then generating offline tests with Offline Test Generator tool.
6. Specifying scenarios or pruning the state space of the model with scenario components to focus the test on some particular part of the model (optional).
7. Selecting strategies how to traverse the state space during testing (optional).
8. Performing testing with Conformance Tester tool. The conformance tester can be used for running both predefined tests and and for establishing the alternating refinement relationship between the selected model and the implementation.
9. Fixing and fine-tuning the model and adapter if during test runs, some problems occur that may be caused either by an error in the model or the adapter.
10. Analysing test results, fixing errors in system under test and re-running tests.

3 Positioning system WFM

Workforce Management (WFM) is a system developed by Reach-U Ltd that enables to track the geographical position of employees, send them messages and check the history of their movement. The system is based on mobile positioning service provided by a mobile service provider.

Application of model-based testing to WFM is motivated by the need to test various aspects of the system and to evaluate the usefulness of the current state of the art of model-based testing for industrial application.

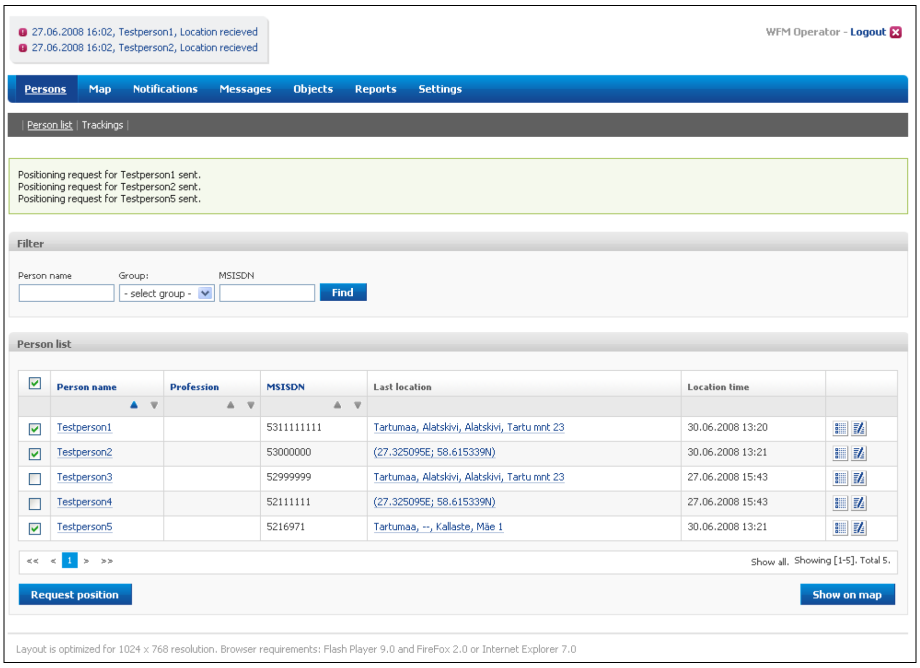
3.1 User roles in WFM

There are the following user roles in WFM.

– Super Administrator. The top user of WFM that manages companies. Works by the mobile operator that offers WFM service to companies.
– Company Administrator. The administrator user in a company.

– Company Operator. The user in a company. Works on logistics and work planning and needs WFM for organizing the work of company field workers: keeping track of their movements and sending messages to them.
– Company Worker. A company field worker that can be positioned and to whom company operator can send messages. Uses a mobile phone - he is positioned using mobile positioning techniques and the messages are sent to him by SMS. Company Worker does not use WFM web interface.

For the current exercise we study the situation where *Company Operator* uses WFM web interface and positions one or more *Company Workers*. *Super Administrator* and *Company Administrator* roles are only used for preparing the needed environment: creating test-companies and test-users in WFM.



27.06.2008 16:02, Testperson1, Location recieved
27.06.2008 16:02, Testperson2, Location recieved

Positioning request for Testperson1 sent.
Positioning request for Testperson2 sent.
Positioning request for Testperson5 sent.

✓	Person name	Profession	MSISDN	Last location	Location time
✓	Testperson1		5311111111	Tartumaa, Alatskivi, Alatskivi, Tartu mnt 23	30.06.2008 13:20
✓	Testperson2		53000000	(27.325095E; 58.615339N)	30.06.2008 13:21
☐	Testperson3		52999999	Tartumaa, Alatskivi, Alatskivi, Tartu mnt 23	27.06.2008 15:43
☐	Testperson4		52111111	(27.325095E; 58.615339N)	27.06.2008 15:43
✓	Testperson5		5216971	Tartumaa, --, Kallaste, Mäe 1	30.06.2008 13:21

<< < 1 > >> Show all. Showing [1-5]. Total 5.

Layout is optimized for 1024 x 768 resolution. Browser requirements: Flash Player 9.0 and FireFox 2.0 or Internet Explorer 7.0

Fig. 4. The page of the WFM system which displays the list of persons to be positioned (in the lower part of the page) and the notifications that positioning requests have been fulfilled (in the box above the menu).

3.2 WFM user interface

A screenshot of the page where the operator can after successfully logging in see the list of workers, choose the workers to position, and receive their locations in textual form is given in Fig. 4). This is the main interface for the part of the application we are testing. The list of numbers that can be positioned can be selected from the lower part of the page. When positioning results are ready, appropriate notifications are fetched by appropriate JavaScript function to the upper region of the page.

3.3 Requirements of the functionality of WFM

The functional specification was given in terms of causal relationships between different messages on different ports of the system. We modeled the actions that comprise a typical positioning scenario, where the operator logs in, selects some cell phone numbers to position and expects the results. The structure of the system and ports where certain events occur are given in Fig. 5. The test system sends the controllable inputs (marked as bold numbers) to the system from the web interface, *web* in Fig. 5. In addition we observe messages on the *web*, *billing*, and *history* interfaces.

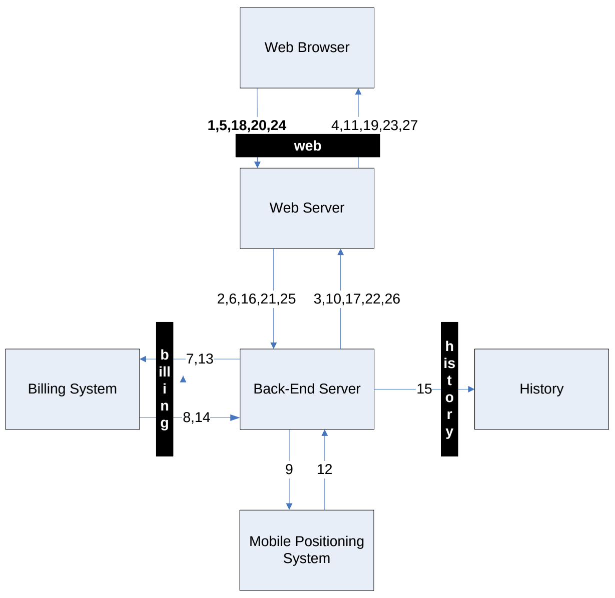


Fig. 5. Architecture of the WorkForce Management system.

The system works in the way that the web server talks to the backend, which in turn communicates with the billing system, the actual positioning system, and logs the procedures to the history subsystem. The events of the WFM with event number corresponding to the numbers in Fig. 5 is given in the Appendix.

3.4 Modeling

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 [25] that when composed specify the *contract*

```
[Feature("Login")]
class Login {
  enum LoginStatus { Success, Failure, Blocked }
  readonly Set<int> numbersPerPage = new Set<int>(0, 1, 2);
  Map<User, LoginStatus> activeLoginRequests = Map<User, LoginStatus>.EmptyMap;
  [Requirement("1.")]
  [Action]
  void WebLogin_Start(User user,Password password) {
    if (password==Password.Correct && Common.state==ControlMode.Running)
      activeLoginRequests = activeLoginRequests.Add(user, LoginStatus.Success);
    else if (password == Password.Incorrect)
      if (Common.usersLoggedIn.Contains(user) && Common.state==ControlMode.Running)
        activeLoginRequests = activeLoginRequests.Add(user, LoginStatus.Success);
      else
        activeLoginRequests = activeLoginRequests.Add(user, LoginStatus.Failure);
  }
  bool WebLogin_StartEnabled() { return Common.state == ControlMode.Running; }
  bool WebLogin_StartEnabled(User user)
  { return !(activeLoginRequests.ContainsKey(user)); }
  [Requirement("4.")]
  [Action]
  void WebLogin_Finish(User user, LoginStatus status,
                [Domain("numbersPerPage")] int numberCount)
  {
   activeLoginRequests = activeLoginRequests.RemoveKey(user);
   if (status == LoginStatus.Success) {
    if (!Common.usersLoggedIn.Contains(user))
     Common.usersLoggedIn = Common.usersLoggedIn.Add(user);
    if (Common.userNumbers.ContainsKey(user))
     Common.userNumbers = Common.userNumbers.RemoveKey(user);
    if (numberCount > 0) {
     Common.userNumbers=Common.userNumbers.Add(user,new Set<PersonNumber>("0"));
     for (int i = 1; i < numberCount; i++)
      Common.userNumbers = Common.userNumbers.Override(
         user,
         Common.userNumbers[user].Add(i.ToString()));
     }
     Common.userPersonCount = Common.userPersonCount.Override(user, numberCount);
   } //else status == LoginStatus.Failure
  }
  bool WebLogin_FinishEnabled() { return Common.state == ControlMode.Running; }
  bool WebLogin_FinishEnabled(User user,LoginStatus status) {
   return
    activeLoginRequests.ContainsKey(user) &&
    activeLoginRequests[user].Equals(status || status == LoginStatus.Blocked);
  }
}
```

Fig. 6. Model program of the login feature.

model program of the positioning functionality of WFM. For example, Fig. 1 represents the common part of the model that is shared between different components of the model. Fig. 6 represents the model program of the login feature. The numbers in the `[Requirement]` attributes correspond to the messages in Fig. 5 and in this way facilitate keeping track of the the requirements that a particular action formalizes. The model programs of the other features are omitted due to space limitations.

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

```
[Feature("OneLogin")]
class OneLogin {

  [Action]
  void WebLogin_Start(User user, Password password) { }
  bool WebLogin_StartEnabled(User user)
  { return !Contract.usersLoggedIn.Contains(user); }
}

[Feature("CorrectPassword")]
class CorrectPassword {

  [Action]
  void WebLogin_Start(User user, Password password) { }
  bool WebLogin_StartEnabled(User user, Password password)
  { return (password == Password.Correct); }
}
```

Fig. 7. Features adding constraints to the model for guiding tests.

the positioning functionality, we restricted the logging in and logging off by scenario features given in Fig. 7. The $OneLogin$ feature restricts new login attempts by a user when already logged in. This enabled us to 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. By the way, the effect of composing the contract model program with it is equivalent to composing it with the FSM given in Fig. 2.

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

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

4 Results

The functionality of the system had previous to applying model-based testing been tested by using JMeter. 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 (Fig. 5).

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.

As the model based testing toolkit is also a piece of software, our experiment revealed two errors in NModel. A minor error was related to a missing dependency between two command line arguments of the conformance tester utility of NModel and the other one was related to the behavior of state cache in NModel that behaved correctly when run on Mono but sometimes returned an unexpected value when run on .Net. Both errors were fixed in the toolkit.

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

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. Time spent is not always proportional to the number of lines of code because different components of the system were with different complexity. The table shows that about 40% of time was spent on modeling and 60% on building various components of the test harness.

The benefits of applying model-based testing to the WFM system are

1. Model-based testing helped to find errors that were not discovered before.
2. Model-based testing provides more flexibility for automated testing than previously used tools (JMeter).

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.

Another problem is that the system is modified slightly according to the needs of each client. This involves turning certain parts of the system on or off, and changing other parts. Such changes are currently not well supported and involve manual modification of the model and the test harness. One possible future direction is evaluation of using model-based testing for testing internal components of the system.

5 Related Work

There are several tools, both commercial and academic, that can be used for model based testing of distributed applications. Some of these tools also support online testing. In particular, QTronic [9] is a commercial tool that supports online testing and allows several input languages for writing models, including C#. Both Spec Explorer (SE) [23] and Spec Explorer 2007 (SE07) support online testing. SE07 is used in the Windows organization as an integral part of the protocol quality assurance process [7] for model based testing of public application-level network protocols. As in NModel the use of composition is of key importance in SE07 for scenario control and the notion of conformance is also alternating refinement [5]. It was shown recently [22] that, for model programs, alternating refinement *coincides* with ioco [19] for systems under test that are input-enabled, which is typically the case for web applications as a robustness criterion. Online testing based on ioco is also used in tools such as TorX [20] and TGV [6]. When timing aspects are critical, Uppaal-Tron [14] provides an environment for modeling, validating and testing real-time systems. Rather than using a programming language such as C# many tools use and support TTCN-3 as the test platform and TTCN based approaches have been applied for testing web services [17, 4]. The above list is not exclusive; see [21] for a comprehensive overview of model-based testing tools. We are not aware of if, besides NModel, these tools have been applied in the context of testing industrial web applications.

A distinguishing feature for the above tools is a formal notion of conformance that builds on ioco or alternating refinement and thus uses action traces and violations of enabling conditions of actions to detect bugs and to produce action traces as witnesses of bugs. There are several other tools and methodologies for testing web services, discussed below, where the emphasis is on scenario control and the conformance aspect is more ad-hoc. What also differs our study from several of these papers, is the fact that the test harness built in this study is asynchronous, and supports nondeterminism. Also, in several papers, just a methodology is described, and no working prototype is built. This paper is a case study where the NModel framework is used for testing a commercial web application.

In [16], a model-based testing method is discussed where statecharts are proposed for web application testing; the level of test execution automation is unclear in this method. In [1], a technique called FSMWeb is presented. The technique copes with the state space explosion problem by using a hierarchical collection of aggregated FSMs. The bottom level FSMs are formed from Web pages and parts of Web pages called logical Web pages, and the top level FSM represents the entire system under test. In our work, we divide the model into logical parts called features and use feature composition for scenario control. In [12], an automated approach for online model-based testing of thin-client web applications is presented, using nondeterministic extended finite state machines. The test system is implemented in the programming language Clean and the tests are executed using the G∀ST test tool. The systems tested are synchronous, G∀ST provides the inputs and checks the validity of the HTML code received as a response. Unlike G∀ST, we allow asynchronous tests where the system under test is proactive and the test tool observes actions initiated by the system under test. The paper [26] describes automated test oracle design for GUI-based software that may have a web based user interface. GUI events are required to be deterministic. In this study, six instances of test oracles were developed and compared in an experiment on four software systems. The study showed that the decisions in implementing test oracles play an important role in determining the effectiveness and cost of the testing process. In our case the model is used both for test generation and as an oracle. In [13] a methodology is presented that uses an object-oriented Web Test Model to support web application testing. From the test model, both structural and behavioral test cases can be derived automatically. Non-determinism does not seem to be allowed. In [15], a framework for model-driven testing for web applications MDWATP is presented as a plug-in for Eclipse. The framework is separated into two main parts: a tester and a modeler. The modeler enables creating, visualizing and storing models. The tester is used for test generation and execution; support for online testing is unclear. In [2] the focus is on model-based testing a web application generator and the web application code itself is generated automatically. In [8], a methodology is proposed for testing applications designed with a model-driven approach. Distributed applications can be tested by using common design patterns. Externally called components are simulated. The same test cases are used for local and distributed testing. In our tests, we also simulated external components like billing and history. Billing and history were simulated by a model-based test oracle. Also the real mobile positioning system was substituted with a simulator for test purposes, but this was outside the scope of test oracle - a separate simulator was used.

6 Summary

The WFM case study showed that it is possible to build a domain-specific web service testing application with a generic lightweight tool such as NModel, with a modest effort, roughly a few man-months. More advanced features, such as for example building support for application restart and test harness components that talk to custom ports take up more time, especially when implemented for the first time. Several of the components developed in this case study are generic, independent of WFM, and can be reused for similar applications.

The study shows how typical behavioral requirements of a web application can be described in terms of model programs that can be used for generating tests according to some scenarios of interest, and as oracles for determining test verdicts.

The study also showed how model-based tests helped to discover errors in the web application that would have been difficult to find with conventional methods.

Still, the case study also demonstrated that a significant amount of time was spent on learning the modeling formalism and last but not least building the test harness, i.e. the adapter between the model and the system under test.

To deploy model-based testing for web applications it would be useful to have in the future: 1) A test harness generator which can generate most of the tedious-to-write parts of the test harness; 2) A tool to pick messages, i.e. combinations of parameters of POST or GET queries from the web browser interface (possible integration with tools like JMeter or Selenium) 3) A scenario generating tool to help generate scenarios that correspond to abstract test cases.

References

1. A. A. Andrews, J. Offutt, and R. T. Alexander. Testing web applications by modeling with FSMs. *Software and Systems Modeling*, 4:326–345, 2005.
2. L. Baresi, P. Fraternali, M. Tisi, and S. Morasca. Towards model-driven testing of a web application generator. In *ICWE*, pages 75–86, 2005.
3. 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.
4. B.Stepien, L.Peyton, and P.Xiong. Framework testing of web applications using ttcn-3, 2009. to appear in STTT.
5. L. de Alfaro. Game models for open systems. In N. Dershowitz, editor, *Verification: Theory and Practice: Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, volume 2772 of *LNCS*, pages 269 – 289. Springer, 2004.
6. J. Fernandez, C. Jard, T. Jéron, and C. Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming - Special Issue on COST247, Verification and Validation Methods for Formal Descriptions*, 29(1-2):123–146, 1997.
7. W. Grieskamp, D. MacDonald, N. Kicillof, A. Nandan, K. Stobie, and F. Wurden. Model-based quality assurance of windows protocol documentation. In *First International Conference on Software Testing, Verification and Validation, ICST*, Lillehammer, Norway, April 2008.
8. R. Heckel and M. Lohmann. Towards model-driven testing. *Electr. Notes Theor. Comput. Sci.*, 82(6), 2003.
9. A. Huima. Implementing conformiq qtronic. In A. Petrenko, M. Veanes, J. Tretmans, and W. Grieskamp, editors, *TestCom/FATES*, volume 4581 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2007.
10. J. Jacky, M. Veanes, C. Campbell, and W. Schulte. *Model-based Software Testing and Analysis with C#.* Cambridge University Press, 2008.
11. JMeter. http://jakarta.apache.org/jmeter/, accessed in May 2009.
12. P. W. M. Koopman, R. Plasmeijer, and P. Achten. Model-based testing of thin-client web applications. In K. Havelund, M. Núñez, G. Rosu, and B. Wolff, editors, *FATES/RV*, volume 4262 of *Lecture Notes in Computer Science*, pages 115–132. Springer, 2006.
13. D. C. Kung, C.-H. Liu, and P. Hsia. An object-oriented web test model for testing web applications. In *COMPSAC '00: 24th International Computer Software and Applications Conference*, pages 537–542, Washington, DC, USA, 2000. IEEE Computer Society.
14. K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou. Testing real-time embedded software using uppaal-tron: an industrial case study. In *EMSOFT '05: Proc. of the 5th ACM international conference on Embedded software*, pages 299–306, New York, NY, USA, 2005. ACM.
15. N. Li, Q. qin Ma, J. Wu, M. zhong Jin, and C. Liu. A framework of model-driven web application testing. In *COMPSAC '06: Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, pages 157–162, Washington, DC, USA, 2006. IEEE Computer Society.

16. H. Reza, K. Ogaard, and A. Malge. A model based testing technique to test web applications using statecharts. In *ITNG '08*, pages 183–188, Washington, DC, USA, 2008. IEEE.
17. I. Schieferdecker, G. Din, and D. Apostolidis. Distributed functional and load tests for web services. *STTT*, 7(4):351–360, 2005.
18. Selenium. http://seleniumhq.org/, accessed in May 2009.
19. J. Tretmans. Model based testing with labelled transition systems. In R. Hierons, J. Bowen, and M. Harman, editors, *Formal Methods and Testing*, volume 4949 of *LNCS*, pages 1–38. Springer, 2008.
20. J. Tretmans and E. Brinksma. TorX: Automated model based testing. In *1st European Conference on Model Driven Software Engineering*, pages 31–43, Nuremberg, Germany, December 2003.
21. M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann, 2006.
22. M. Veanes and N. Bjørner. Input-output model programs. Technical Report MSR-TR-2009-56, Microsoft Research, Redmond, May 2009.
23. M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson. Model-based testing of object-oriented reactive systems with Spec Explorer. In R. Hierons, J. Bowen, and M. Harman, editors, *Formal Methods and Testing*, volume 4949 of *LNCS*, pages 39–76. Springer, 2008.
24. M. Veanes, C. Campbell, W. Schulte, and N. Tillmann. Online testing with model programs. *SIGSOFT Softw. Eng. Notes*, 30(5):273–282, 2005.
25. 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.
26. Q. Xie and A. M. Memon. Designing and comparing automated test oracles for gui-based software applications. *ACM Trans. Softw. Eng. Methodol.*, 16(1):4, 2007.

Appendix A: Textual specification of the behavior of WFM

1. Operator logs in to the system. Web browser sends web server a request containing user name and password.
2. Web server sends login request to back-end.
3. Back-end returns response (login successful/unsuccessful) to web server.
4. Web server returns login response to browser.
5. In web browser, the operator makes a HTTP request for new position of a person or persons.
6. Web server forwards the request to back-end.
7. Back-end server asks the billing system to check whether the operator has enough credit left to make the request.
8. Billing system returns the answer about credit status of the operator. The status can be OK, NOT OK, or error.
9. If the billing system returned positive answer, back-end sends the positioning request asynchronously to mobile positioning system.
10. Back-end returns the answer received from the billing system to web server.
11. Web server sends the answer to web browser. In positive case, the answer shown to the WFM end user is "Positioning request for person x sent" for every person that was positioned. The answer sent to web browser does not contain the person's (or persons') new location yet.
12. Back-end receives response from MPS, containing the new location of person(s).
13. Back-end charges the operator in billing system.
14. Billing system returns the result to back-end. The result can be OK, NOT OK, or error.
15. If the result from billing system is OK, the result is sent to History.
16. Web server polls the back-end to check whether the new location data is available.
17. Back-end returns the answer: 1 (new location data available) or 0 (no new data available).
18. Without user involvement, web browser calls web server to check whether the new location data for the persons in the company is available.
19. Webserver returns the answer. Returned value 0 (no new data available) means that the process goes back to step No 16 until new data is received. Returned value 1 means that there is new location data available for the persons in current company and the process goes on to step No 20.
20. Over HTTP, web browser requests last location of the positioned persons.
21. Web server calls back-end for last known location of person.
22. Back-end returns last known location.
23. Web server returns XML containing two last changes in the data of persons of the company and web browser displays the data on a banner on web page.
24. Operator logs out of the system. Web browser sends web server a logout request.
25. Web server sends logout request to back-end.
26. Back-end returns response (logout successful/unsuccessful) to web server.
27. Web server returns logout response to browser.