Article08/06/2015

December 2013

Volume 28 Number 12

Model Based Testing - An Introduction to Model-Based Testing and Spec Explorer

By Sergio Mera | December 2013

Producing high-quality software demands a significant effort in testing, which is probably one of the most expensive and intensive parts of the software development process. There have been many approaches for improving testing reliability and effectiveness, from the simplest functional black-box tests to heavyweight methods involving theorem provers and formal requirement specifications. Nevertheless, testing doesn't always include the necessary level of thoroughness, and discipline and methodology are often absent.

Microsoft has been successfully applying model-based testing (MBT) to its internal development process for more than a decade now. MBT has proven a successful technique for a variety of internal and external software products. Its adoption has steadily increased over the years. Relatively speaking, it has been well received in the testing community, particularly when compared with other methodologies living on the "formal" side of the testing spectrum.

Spec Explorer is a Microsoft MBT tool that extends Visual Studio, providing a highly integrated development environment for creating behavioral models, plus a graphical analysis tool for checking the validity of those models and generating test cases from them. We believe this tool was the tipping point that facilitated the application of MBT as an effective technique in the IT industry, mitigating the natural learning curve and providing a state-of-the-art authoring environment.

In this article we provide a general overview of the main concepts behind MBT and Spec Explorer, presenting Spec Explorer via a case study to showcase its main features. We also want this article to serve as a collection of practical rules of thumb for understanding when to consider MBT as a quality assurance methodology for a specific testing problem. You shouldn't blindly use MBT in every testing scenario. Many times another technique (like traditional testing) might be better a choice.

What Makes a Testable Model in Spec Explorer?

Even though different MBT tools offer different functionality and sometimes have slight conceptual discrepancies, there's general agreement about the meaning of "doing MBT." Model-based testing is about automatically generating test procedures from models.

Models are usually manually authored and include system requirements and expected behavior. In the case of Spec Explorer, test cases are automatically generated from a state-oriented model. They include both test sequences and the test oracle. Test sequences, inferred from the model, are responsible for driving the system under test (SUT) to reach different states. The test oracle tracks the evolution of the SUT and determines if it conforms to the behavior specified by the model, emitting a verdict.

The model is one of the main pieces in a Spec Explorer project. It's specified in a construct called model programs. You can write model programs in any .NET language (such as C#). They consist of a set of rules that interact with a defined state. Model programs are combined with a scripting language called Cord, the second key piece in a Spec Explorer project. This permits specifying behavioral descriptions that configure how the model is explored and tested. The combination of the model program and the Cord script creates a testable model for the SUT.

Of course, the third important piece in the Spec Explorer project is the SUT. It isn't mandatory to provide this to Spec Explorer to generate the test code (which is the Spec Explorer default mode), because the generated code is inferred directly from the testable model, without any interaction with the SUT. You can execute test cases "offline," decoupled from model evaluation and test-case generation stages. However, if the SUT is provided, Spec Explorer can validate that the bindings from the model to the implementation are well-defined.

Case Study: A Chat System

Let's take a look at one example to show how you can build a testable model in Spec Explorer. The SUT in this case is going to be a simple chat system with a single chat room where users can log on and log off. When a user is logged on, he can request the list of the logged-on users and send broadcast messages to all users. The chat server always acknowledges the requests. Requests and responses behave asynchronously, meaning they can be intermingled. As expected in a chat system, though, multiple messages sent from one user are received in order.

One of the advantages of using MBT is that, by enforcing the need to formalize the behavioral model, you can get a lot of feedback for the requirements. Ambiguity, contradictions and lack of context can surface at early stages. So it's important to be precise and formalize the system requirements, like so:

```
R1. Users must receive a response for a logon request.
R2. Users must receive a response for a logoff request.
R3. Users must receive a response for a list request.
R4. List response must contain the list of logged-on users.
R5. All logged-on users must receive a broadcast message.
R6. Messages from one sender must be received in order.
```

Spec Explorer projects use actions to describe interaction with the SUT from the test-system standpoint. These actions can be call actions, representing a stimulus from the test system to the SUT; return actions, capturing the response from the SUT (if any); and event actions, representing autonomous messages sent from the SUT. Call/return actions are blocking operations, so they're represented by a single method in the SUT. These are the default action declarations, whereas the "event" keyword is used to declare an event action. **Figure 1** shows what this looks like in the chat system.

Figure 1 Action Declarations

XML

```
// Cord code
config ChatConfig
{
  action void LogonRequest(int user);
  action event void LogonResponse(int user);
  action void LogoffRequest(int user);
  action event void LogoffResponse(int user);
  action void ListRequest(int user);
  action event void ListResponse(int user, Set<int> userList);
  action void BroadcastRequest(int senderUser, string message);
  action void BroadcastAck(int receiverUser,
    int senderUser, string message);
  // ...
}
```

With the actions declared, the next step is to define the system behavior. For this example, the model is described using C#. System state is modeled with class fields, and state

transitions are modeled with rule methods. Rule methods determine the steps you can take from the current state in the model program, and how state is updated for each step.

Because this chat system essentially consists of the interaction between users and the system, the model's state is just a collection of users with their states (see **Figure 2**).

Figure 2 The Model's State

XML

```
/// <summary>
/// A model of the MS-CHAT sample.
/// </summary>
public static class Model
{
  /// <summary>
  /// State of the user.
  /// </summary>
  enum UserState
  {
    WaitingForLogon,
    LoggedOn,
    WaitingForList,
    WatingForLogoff,
  }
  /// <summary>
  /// A class representing a user
  /// </summary>
  partial class User
  {
    /// <summary>
    /// The state in which the user currently is.
    /// </summary>
    internal UserState state;
    /// <summary>
    /// The broadcast messages that are waiting for delivery to this user.
    /// This is a map indexed by the user who broadcasted the message,
    /// mapping into a sequence of broadcast messages from this same user.
    /// </summary>
    internal MapContainer<int, Sequence<string>> waitingForDelivery =
      new MapContainer<int,Sequence<string>>();
  }
    /// <summary>
    /// A mapping from logged-on users to their associated data.
    /// </summary>
    static MapContainer<int, User> users = new MapContainer<int,User>();
      // ...
  }
```

As you can see, defining a model's state isn't much different from defining a normal C# class. Rule methods are C# methods for describing in what state an action can be activated. When that happens, it also describes what kind of update is applied to the model's state. Here, a "LogonRequest" serves as an example to illustrate how to write a rule method:

XML

```
[Rule]
static void LogonRequest(int userId)
{
  Condition.IsTrue(!users.ContainsKey(userId));
  User user = new User();
  user.state = UserState.WaitingForLogon;
  user.waitingForDelivery = new MapContainer<int, Sequence<string>>();
  users[userId] = user;
}
```

This method describes the activation condition and update rule for the action "LogonRequest," which was previously declared in Cord code. This rule essentially says:

- The LogonRequest action can be performed when the input userId doesn't yet exist in the current user set. "Condition.IsTrue" is an API provided by Spec Explorer for specifying an enabling condition.
- When this condition is met, a new user object is created with its state properly initialized. It's then added to the global users collection. This is the "update" part of the rule.

At this point, the majority of the modeling work is finished. Now let's define some "machines" so we can explore the system's behavior and get some visualization. In Spec Explorer, machines are units of exploration. A machine has a name and an associated behavior defined in the Cord language. You can also compose one machine with others to form more complex behavior. Let's look at a few example machines for the chat model:

XML

```
machine ModelProgram() : Actions
{
  construct model program from Actions where scope = "Chat.Model"
}
```

The first machine we define is a so-called "model program" machine. It uses the "construct model program" directive to tell Spec Explorer to explore the entire behavior of the model

based on rule methods found in the Chat.Model namespace:

```
XML
```

```
machine BroadcastOrderedScenario() : Actions
{
  (LogonRequest({1..2}); LogonResponse){2};
  BroadcastRequest(1, "1a");
  BroadcastRequest(1, "1b");
  (BroadcastAck)*
}
```

The second machine is a “scenario,” a pattern of actions defined in a regular-expression-like way. Scenarios are usually composed with a “model program” machine in order to slice the full behavior, as in the following:

```
XML
```

```
machine BroadcastOrderedSlice() : Actions
{
  BroadcastOrderedScenario || ModelProgram
}
```

The “||” operator creates a “synchronized parallel composition” between the two participating machines. The resulting behavior will contain only the steps that can be synchronized on both machines (by “synchronized” we mean have the same action with the same argument list). Exploring this machine results in the graph shown in **Figure 3**.

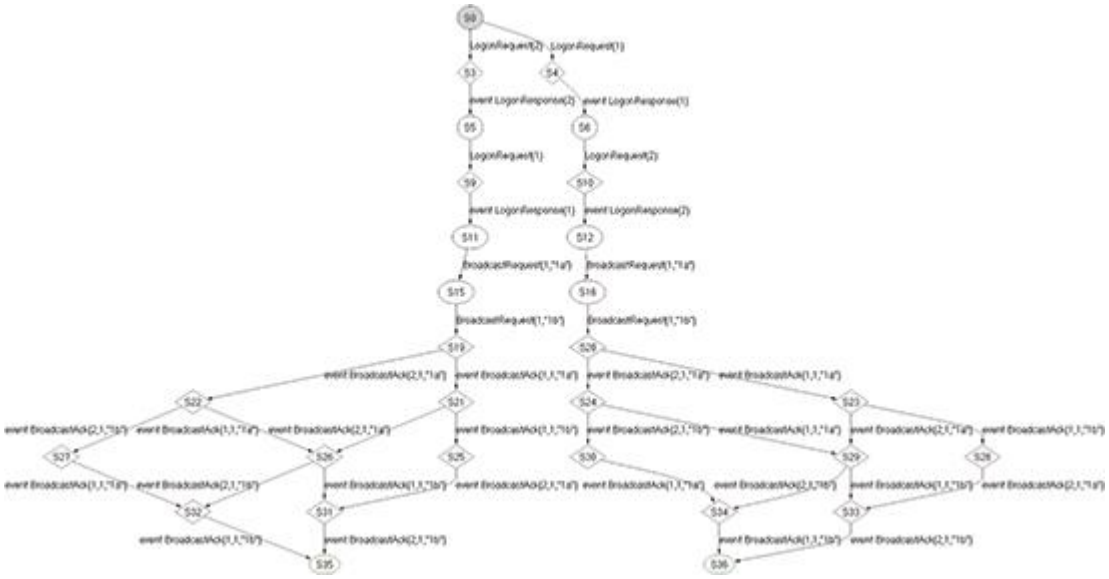

Figure 3 Composing Two Machines

As you can see from the graph in **Figure 3**, the composed behavior complies with both the scenario machine and the model program. This is a powerful technique for getting a simpler subset of a complex behavior. Also, when your system has infinite state space (as in the case of the chat system), slicing the full behavior can generate a finite subset more suitable for testing purposes.

Let's analyze the different entities in this graph. Circle states are controllable states. They're states where stimuli are provided to the SUT. Diamond states are observable states. They're states where one or more events are expected from the SUT. The test oracle (the expected result of testing) is already encoded in the graph with event steps and their arguments. States with multiple outgoing event steps are called non-deterministic states, because the event the SUT provides at execution time isn't determined at modeling time. Observe that the exploration graph in **Figure 3**contains several non-deterministic states: S19, S20, S22 and so forth.

The explored graph is useful for understanding the system, but it's not yet suitable for testing because it isn't in "test normal" form. We say a behavior is in test normal form if it doesn't contain any state that has more than one outgoing call-return step. In the graph in **Figure 3**, you can see that S0 obviously violates this rule. To convert such behavior into test normal form, you can simply create a new machine using the test cases construction:

XML

```
machine TestSuite() : Actions where TestEnabled = true
{
  construct test cases where AllowUndeterminedCoverage = true
  for BroadcastOrderedSlice
}
```

This construct generates a new behavior by traversing the original behavior and generating traces in test normal form. The traversal criterion is edge coverage. Each step in the original behavior is covered at least once. The graph in **Figure 4** shows the behavior after such traversal.

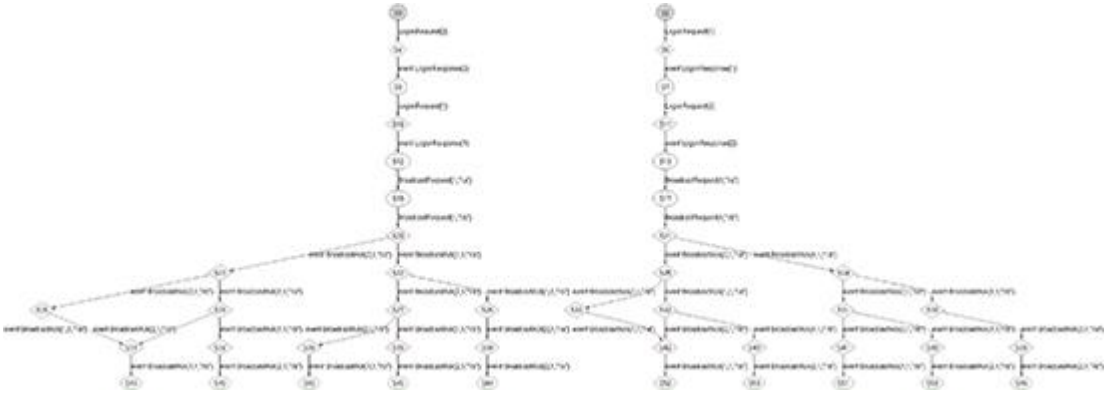

Figure 4 Generating New Behavior

To achieve test normal form, states with multiple call-return steps are split into one per step. Event steps are never split and are always fully preserved, because events are the choices that the SUT can make at runtime. Test cases must be prepared to deal with any possible choice.

Spec Explorer can generate test suite code from a test normal form behavior. The default form of generated test code is a Visual Studio unit test. You can directly execute such a test suite with the Visual Studio test tools, or with the mstest.exe command-line tool. The generated test code is human readable and can be easily debugged:

XML

```
#region Test Starting in S0
[Microsoft.VisualStudio.TestTools.UnitTesting.TestMethodAttribute()]
public void TestSuiteS0() {
  this.Manager.BeginTest("TestSuiteS0");
  this.Manager.Comment("reaching state \'S0\'");
  this.Manager.Comment("executing step \'call LogonRequest(2)\'");
  Chat.Adapter.ChatAdapter.LogonRequest(2);
  this.Manager.Comment("reaching state \'S1\'");
  this.Manager.Comment("checking step \'return LogonRequest\'");
  this.Manager.Comment("reaching state \'S4\'");
  // ...
    }
```

The test-code generator is highly customizable and can be configured to generate test cases that target different test frameworks, such as NUnit.

The full Chat model is included in the Spec Explorer installer.

When Does MBT Pay Off?

There are pros and cons when using model-based testing. The most obvious advantage is that after the testable model is completed, you can generate test cases by pushing a button. Moreover, the fact that a model has to be formalized up front enables early detection of requirement inconsistencies and helps teams to be in accord in terms of expected behavior. Note that when writing manual test cases, the "model" is also there, but it's not formalized and lives in the head of the tester. MBT forces the test team to clearly communicate its expectations in terms of system behavior and write them down using a well-defined structure.

Another clear advantage is that project maintenance is lower. Changes in system behavior or newly added features can be reflected by updating the model, which is usually much simpler than changing manual test cases, one by one. Identifying only the test cases that need to be changed is sometimes a time-consuming task. Consider as well that model authoring is independent of the implementation or the actual testing. That means that different members of a team can work on different tasks concurrently.

On the downside, a mindset adjustment is often required. This is probably one of the major challenges of this technique. On top of the well-known problem that people in the IT industry don't have time to try out new tools, the learning curve for using this technique isn't negligible. Depending on the team, applying MBT might require some process changes as well, which can also generate some push back.

The other disadvantage is that you have to do more work in advance, so it takes more time to see the first test case being generated, compared with using traditional, manually written test cases. Additionally, the complexity of the testing project needs to be great enough to justify the investment.

Luckily, there are some rules of thumb we believe help identify when MBT really pays off. Having an infinite set of system states with requirements you can cover in different ways is a first sign. A reactive or distributed system, or a system with asynchronous or non-deterministic interactions is another. Also, methods that have many complex parameters can point in the MBT direction.

When these conditions are met, MBT can make a big difference and save significant testing effort. An example of this is Microsoft Blueline, a project where hundreds of protocols were verified as part of the Windows protocol compliance initiative. In this project, we used Spec Explorer to verify the technical accuracy of protocol documentation with respect to the actual protocol behavior. This was a gigantic effort and Microsoft spent around 250 person-years in testing. Microsoft Research validated a statistical study that showed using

MBT saved Microsoft 50 person-years of tester work, or around 40 percent of the effort compared with a traditional testing approach.

Model-based testing is a powerful technique that adds a systematic methodology to traditional techniques. Spec Explorer is a mature tool that leverages the MBT concepts in a highly integrated, state-of-the-art development environment as a free Visual Studio Power Tool.

Yiming Cao *is a senior development lead for the Microsoft Interop and Tools team and works on the Protocol Engineering Framework (including Microsoft Message Analyzer) and Spec Explorer. Before joining Microsoft he worked for IBM Corp. on its enterprise collaboration suite, and then joined a startup company working on media-streaming technologies.*

Sergio Mera *is a senior program manager for the Microsoft Interop and Tools team and works on the Protocol Engineering Framework (including Microsoft Message Analyzer) and Spec Explorer. Before joining Microsoft he was a researcher and lecturer for the Computer Science Department at the University of Buenos Aires and worked on modal logics and automated theorem proving.*

Thanks to the following technical expert for reviewing this article: Nico Kicillof (Microsoft)
Nico Kicillof (nicok@microsoft.com) is the Lead Program Manager for Windows Phone Build Architecture and Development Tools. His work consists in creating tools and methods which enable engineers to build, test, and maintain software products. Before joining Microsoft, he was a Professor and Deputy Chair of the Computer Science Department at University of Buenos Aires.