Building Executable Software Test Specifications

Michael Corning
Microsoft Corporation

Software tends to get more powerful. Hardware, and to some extent software, technologies evolve at non-linear rates with Moore's Law at the limit. Software *test*, however, has not kept pace, so even as systems have more and more services and features, relatively fewer of these new services and features are adequately tested. While there is no panacea for this predicament, there are definite steps test organizations can take to begin to close the gap, or at least slow down the speed of the spread.

The key to success for testing advanced software systems is to lower the time it takes to design, implement, and execute software tests and to raise the amount of time machines spend executing these tests; i.e., the goal should be to have 100% of the machines running interesting tests 100% of the time. Executable software test specifications help with the first goal, and generating and running an arbitrarily large number of test cases continuously helps with the second. Indeed, there is even some prospect that the software test infrastructure, itself, can decide what's "interesting."

There are two obstacles to success, two specific problems, keeping us from reaching our first goal. First, the bandwidth of the systems we test far outstrips the bandwidth of testers. This means human comprehension of the artifacts of human ingenuity always lags. But if we can apply the very power of the systems we test against the systems we test then we can leverage the power of the machine to understand machines.

As we make progress using this leverage we introduce a reciprocal problem: if we massively test our systems we might expect to find massive numbers of bugs (but we should not forget that those bugs were always in there). If this happens, it's three steps forward one step back. We need to devise test systems that, by design, do a better job of keeping bugs out of the system, altogether.

Testers on Microsoft's Server Management team face these issues everyday. There's a very good reason that there are no widely effective server management systems on the market today: server management (among thousands of machines) is difficult, and server management is not built into server technology. So while Microsoft developers rectify this deficiency, Microsoft testers are devising clever systems to enable them to test these vastly more powerful server technologies. This report documents some of their success.

> This document augments the Power Point presentation by focusing far more on the technical details of the executable software test specification system. See the Power Point slides for a higher level view of the system.

A Model of a Test

Over the years Microsoft test engineers have developed a model of a software test. These testers have simultaneously developed a strategy to cope with the issues noted above, a strategy especially well-suited to an environment like server management where all kinds and combinations of services and topologies is arrayed before them.

In the early stages of the design phase of test development, testers partition tests into a handful of categories. These categories help guide the testers' thinking about what to test. Sometimes these priorities radically change, as they did last year at Microsoft when the entire company did nothing but run security tests. The categories chosen to guide test development are up to test management to sanction, but once done, those categories become a permanent part of the test org's model of a test.

The second strategy used by testers is to stratify or prioritize tests. This permits some level of control over which tests run when. There are two general parameters that distinguish one stratum from the next. Time to run and functional coverage, and they often interrelate to each other. That is, (normally) functional constraint implies short run times. As with test partitions, test strata are dictated by test management who has considerable discretion when setting standards.

The third strategy for dealing with complex test engagements is to define a test matrix that maps values of variables describing aspects of the overall system. Variable for server management include operating system (including versions and service packs) and services such as Exchange Server or Active Directory or Internet Information Services. Again, specifying these combinations enables test management to decide what configurations are most important to test.

The Microsoft Server Management Test organization has given specific names to each of these strategic components. The "Set" is the construct that enables groups of tests to be executed in different configurations. The "Level" construct is a flexible data container that includes both an ordinal number (each representing one of eight strata) and a category name. The final construct is the "Var." The Var contains a description of each individual test (or class of tests). Vars can optionally include runtime data, in the spec, these runtime data are rendered in so-called "data tables."

Data tables are an important productivity feature. One of the most powerful ways to use the machine to test the machine is to enable the machine to generate test cases. In a data-driven spec the tester is responsible for either defining individual tests (by the specific data passed to the test execution runtime) or by declaring what values a test's arguments can take (and let the machine actually generate the individual tests, each with its own combination of possible values for the test's arguments).

These three constructs (Set, Level, and Var) form a natural hierarchy, and each Var can be uniquely identified at any moment in time by the Set.Level.Var triplet. Figure 1 is the object model for Sets, Level, and Vars.

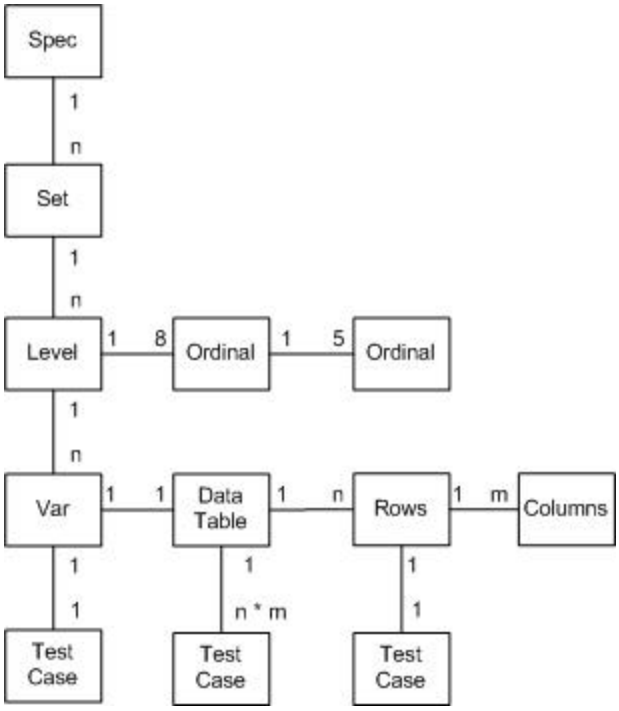

Figure 1: A Model of Tests

Since it might not be clear from the object model diagram, I should note in passing that a Set may contain any number of Level numbers, and any of those Level numbers can be the same value; what makes Levels

of the same number different is the category assigned to each. So a more precise way of stating it is that any Set can contain any number of Level Number/Category combinations. For example you could have up to six Level 3 (Functional) tests in any Set, each Level using a different category.

This object model is more closely related to an interface, for it merely specifies the constituents of a model of a test. The SLV model is silent with respect to implementation. For example, early implementations of this test model used text files to list the Vars to execute (and testers were responsible for maintaining the SLV identity of tests which got them into trouble when they copied and pasted lines in the file for new tests but forgot to renumber the new lines); the current implementation of this model on the Server Management team uses XML. As we will see later, XML provides some very distinct advantages, especially when it comes to generating (automatically numbered) test cases.

> It is important to note that the object model in Figure 1 has information only used by humans. That human-friendly information is the Level Category. Each spec reports the number of tests designed for each of the five categories defined by test management, but other than reporting and guiding a tester's test design, the category is not used at runtime.

Modeling a Test

The executable software test specification described above is manually (or semi-automatically) generated. For all intents and purposes data-driven tests are static in that the combinations of data values that define a test are fixed by the tester explicitly. Granted, by merely specifying the names of a test's arguments and the possible values of each argument, a data-driven test will generate n * m number of test cases, but that Cartesian product changes only if the number of arguments or number of possible values change.

There are varieties of executable software test specifications that are far more dynamic than this. One variety generates finite state machines directly from tester input. The FSM tool used by Server Management testers was developed by a different team inside Microsoft, and that tool is not available outside the company. Developing a similar tool is not outside the scope of skill of most competent test organizations, and until a third party product appears on the market, that's the only option available for testers who want to use models to dynamically generate test cases.

A second variety of model-based tool is, however, available to readers of this paper. The Abstract State Machine Language (AsmL) is available from Microsoft Research. At the time of this writing, AsmL was still in beta, but was capable of not only generating a finite state machine and test cases from the FSM, it was also capable of generating a state graph of the FSM. See my "Confessions of a Modeling Bigot" column on ASPToday.com for more details on this invaluable modeling language.

While important, most of the balance of this paper will not discuss model-based techniques and will instead focus on technologies and tools widely available today to develop executable software test specifications.

The Role of Test Specifications

Specifications are an important part of the software development process, especially when many people contribute to the process. The essential function of a specification is to describe what a program does and why the program needs to do it. This information is crucial when submitting a test design to peer review and failure investigation (by someone other than the tester).

How the software test specification fulfills its function requires thought and care in the design of the data the specification conveys. The data model used by the test specification is a blueprint for the design and execution of developed tests. The spec is a model of the test that describes what things are important to test. In other words, a test spec tells testers how to think about the software they will test. It's the tester's job to decide what to think, and it's the tester's lead and peers that pass judgment during peer review on the quality

of that thinking. If consistently used and reviewed, reports of the data all software test specifications include will be reliable and useful to readers such as test leads and managers.

Software test specifications can have one more important function: if you serialize the data in the software test specification you can execute it by sending the serialized data to a properly constructed test execution runtime.

Executable software test specifications can contribute to the consistency of the system in several ways. First, you can validate serialized data so the test execution runtime always processes input of consistent quality. Second, a well-designed schema enables testers to design and implement tests that consistently cover the application programming interface or the user interface under test. Third, since the test execution runtime can't start without the serialized test specification and since each test variation is bound to a class in the code run by the test execution runtime, the specification and implementation are always consistent with respect to each other.

Executable software test specifications can also streamline the test execution process. Part of the data emitted from the specification includes references to classes and optional data that start up a test run. These test run contexts can install software or configure the application under test before any test variations run. Different contexts can bind to different test variations at runtime, and all this because the executable software test specification document is serialized as xml.

An Executable Software Test Specification Workflow

Figure 2 shows specific steps and files that make up the executable software test specification workflow. In this workflow, a tester uses Microsoft Word to statically generate test cases; i.e., the tester specifies what each test case will be, what steps it will take and what data it will use at runtime. Word was chosen because an effective test spec has two parts, an informal, free-form section that tells the reader something about the system under test and the rationale for the tests outlined in the subsequent structured section of the spec containing the actual tests to run. A word processor like Word is the natural choice for writing free form content. Writing a user interface for Word enabled testers to create the structured document section, as well.

Each step in the workflow generates a very specific form of XML. The Intermediate Markup Language (IML) is the XML schema generated first. Its role is to faithfully represent the contents of the Word document so that the system can generate a web form version of the spec. The Failure Investigator component of the system uses this machine-readable version of the test spec to display the section of the spec related to a failed test. This ability of infrastructure to reference the spec speeds up the bug resolution process.

The transformation that generates the second XML schema, XIML, is the process that generates test cases from the description given by the tester in the spec. Depending on the data provided by the tester, a spec with only a few Vars in it can expand to hundreds of test cases. The key thing to remember is that a test consists of two things: actions to take and data to use. A single var is defined by the actions it will take; and the multiplicity of generated test cases is a function of the diversity of data these actions consume. Our executable software test specification system fully exploits this duality of executable test specs.

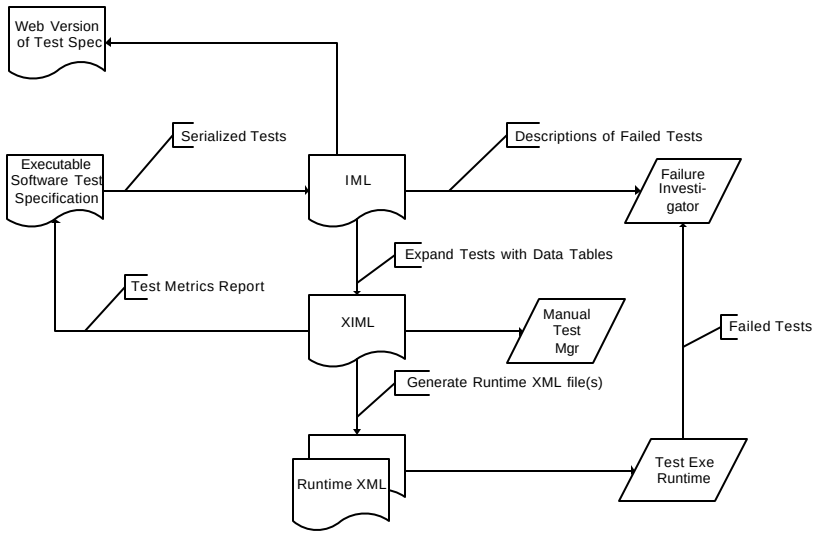

Figure 2: Executable Software Test Specification Workflow

> When a model is the executable software test specification Figure 2 is slightly modified. The IML and XIML files don't exist and the Runtime XML Data provides the data for the web based version of the spec.

The ultimate goal of all this xml processing is to get the data in a shape that the test execution runtime can consume. The final step in the workflow transforms the XIML into an XML format defined by the developer of the test execution runtime. Since the original data was entered into our system in a Word document, there's the distinct possibility that the XIML data may be malformed or incomplete and the serialization process (documented in my MSDN.Online article, "Using Visual Basic .NET from VBA to Serialize Word Documents as XML") didn't catch the problem or couldn't resolve it without the tester's input. And since reports of the number of tests specified should reflect the number of tests that the runtime could actually execute, it makes sense to base the spec's reports on data validated by the XML Schema that defines the Runtime XML file consumed by the test execution runtime. That's what our system does.

So once the spec's data is validated it's ready for the tester to send it to the test execution runtime. Testers use a command line to call on the runtime, and they can pass arguments in the call that specifies which tests to run. Testers can run one Set or a range of Sets; a single Level or a range of Levels; a single Var or a range of Vars. It all depends on what the tester needs at runtime.

Our test execution runtime reports failures to the results repository by the Set.Level.Var triplet. This identification can assist an investigator when resolving the bug, especially when the person investigating the failure is not the author of the test.

The Schema is the API

In this section we'll take a closer look at the xml files depicted in Figure 2. Think of these XML files as program interfaces. If you want to devise your own executable software test specification system, use these XML "interfaces" to guide your development work.

The IML Schema

Figure 3 shows an example IML file (some nodes without data have been removed for clarity). The IML schema captures the basic information for its test specification, including the name of the spec document

(see `doc` attribute in the `testAreas` root element). Also noteworthy are the `templateFolder` and `specTransform` attributes; these attributes provide a level of indirection that enables me to change the XSLT transform that renders the IML data into a web version of the spec without changing the source code in the Word template. This was a lesson I learned the hard way, so my general guideline is ensure all xml files have enough data to transform themselves without undue reliance on binary augmentation (such as VBA code in a Word template).

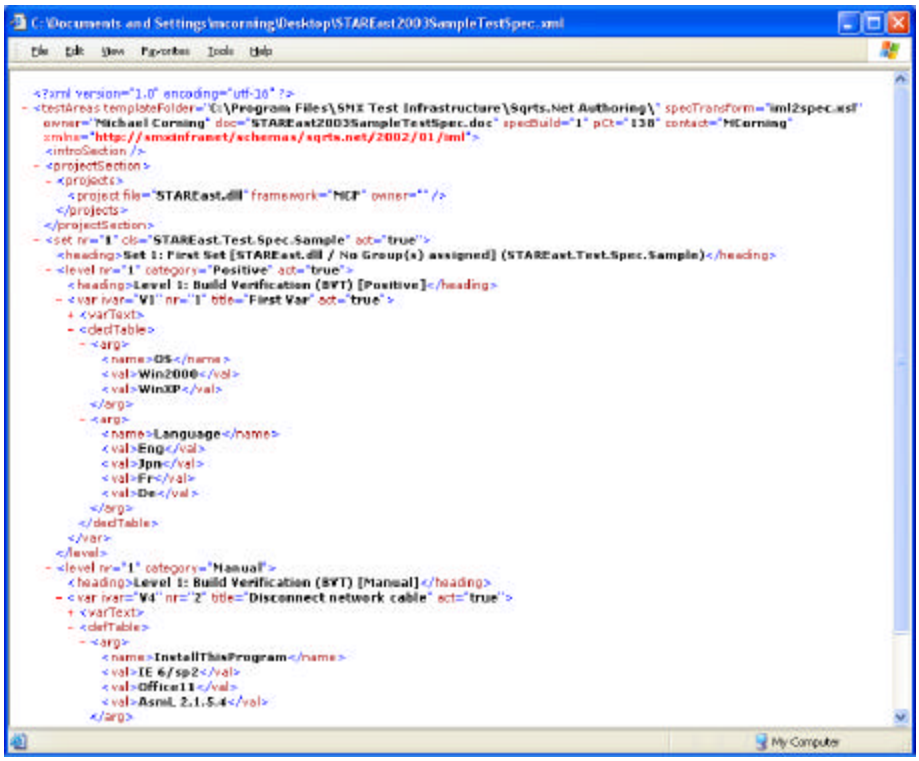

Figure 3: Typical IML

The other implementation specific data inside each IML file is the `projects` node near the top of the `testAreas` node and the `cls` attribute (in Figure 3, only the `set` node uses a `cls` attribute, but the `cls` attribute can appear on `level` and `var` nodes, as well). The `file` attribute of the `project` node contains the name of the dll the test execution runtime will run. The `cls` attribute cites the name of the class in the dll that actually provides the execution context for all `vars` under the `set` node.

> If the `cls` attribute appears on child nodes, the class reference will override any class set by ancestor nodes, otherwise all descendents inherit the highest class reference.

The `act` attribute on `set`, `level`, and `var` nodes enables the tester to temporarily disable a test, perhaps because of an unresolved blocking bug. Toggling the value of this `act` attribute is done in the Word document's user interface.

Note in Figure 3 that there are two `level` nodes, each using the number 1, but each using a different `category`. As you will see below, the manual tests will be separated and collected in their own node in the XIML file. This way the manual test manager system can easily access the data necessary to render manual tests as web forms.

The `ivar` attribute on the `var` node is an immutable `var` id. You can identify each `var` in a test by its `set.level.var` number or by its `ivar` number. The SLV number acts like a name and can change from one version of a spec to the next. The `ivar` value, however, never changes; it is simply the letter "v" with an appended integer incremented by the Word document every time a new `var` is added to the spec.

Last, but not least, Figure 3 includes both a Declared test and a Defined test. The `declTable` element contains the names of the arguments (the `name` elements) and the possible values (the `val` elements) of those arguments used by the `Run` method of the class cited in the `var`'s `set` parent element. The `Run` method is mandated by the interface specified by the developer of the test execution runtime. Every runtime class implements a `Run` method, and the test execution runtime provides the `Run` method with a context object that includes the SLV value of the currently running `var` along with any runtime data specified by the tester and contained in the `var`'s `rec` tag(s). It's up to the tester to implement (and hopefully reuse) test code for each `var` in the spec.

As you will see next, the immediate effect of the `declTable` is to expand one specified `var` into eight runtime `var` nodes. The system generates the Cartesian product of the number of arguments and the number of possible values of each argument. Care must be taken for tables with even a small number of rows and anything more than ten columns (columns act like a power function so 4 rows and 10 columns can produce almost s million test cases!). The system warns the tester when expansion will produce more than 100 test cases. Our test execution runtime is capable of optimizing pair-wise combinations of parameter values.

The XIML Schema

Figure 4 is the XIML file transformed from the IML file shown in Figure 3 above. The first thing to note is that the system has flattened the IML. Now each `var` has a (possibly duplicated) value for `set` and `lvl`. The next thing to note is that the `cls` attribute has the value given either to the `var` node in the IML or the `var's level` or `set` nodes. Next, note that the first `Var` in the IML has expanded into eight `var` nodes in the XIML file. Eight is the Cartesian product of two arguments with two and four possible values.

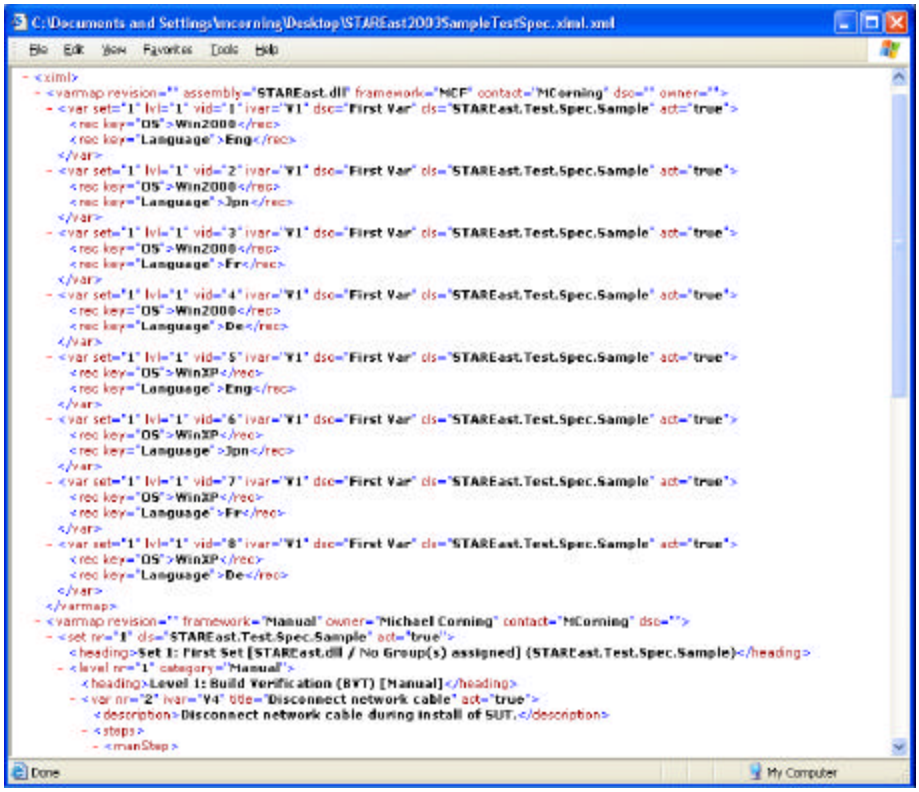

Figure 4: XIML generated from IML

Finally, note that the manual test appears in its own `varmap` node. As explained above, we treat manual tests as first class citizens and permit the executable software test specification to include both manual and automated tests in a single document. However, since a different test execution "runtime" (a human) will run the manual tests, the XIML file needs to group the manual tests in separate nodes. This way the next processing step, generating runtime xml for the test execution runtime, can more easily transform XIML into the runtime xml schema.

The Runtime XML Schema

The final transformation of serialized test data occurs for the benefit of the test execution runtime. Figure 5 shows the runtime xml generated by the XIML file from Figure 4 above.

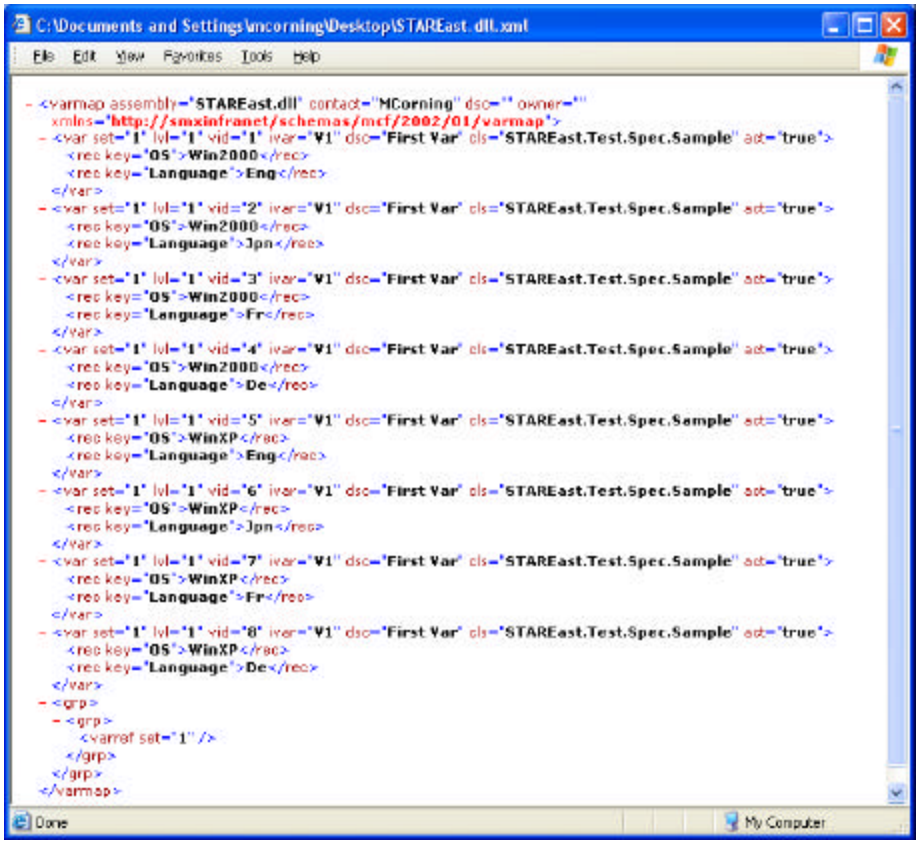

Figure 5: Runtime XML data

The data in the `varmap` node is practically indistinguishable from the `varmap` node in the XIML. For automated tests this means the final processing step merely writes each XIML `varmap` node to the file system with its own name (a name taken from the `assembly` attribute in the `varmap` tag). At this point, the tester is ready to begin executing tests.

> If a model generates test cases (instead of a tester doing it by hand), the model's xml file is transformed directly to the runtime xml schema above. There is no need for the IML or XIML file. And where the IML file gets transformed into HTML for data-driven tests, our system transforms model-driven runtime XML into HTML. The end result is a web form that looks the same for both data-driven and model-driven tests, and this makes peer review of specs easier.

What We Learned

The system outlined in this document has been evolving at Microsoft for over twelve years. Its current incarnation fully exploits the power and flexibility of XML and the Common Language Runtime. The model of a test represented by the three constructs, Set, Level, and Var continue to serve testers well as they work hard to develop tests for ever more powerful software systems such as Microsoft Application Center, Microsoft Operations Manager, and future generations of Microsoft Server Management software.

Our team has made a substantial investment of intellectual capital in this executable software test specification system, and we continue to enjoy the dividends from this investment. A common, consistent, and effective software test specification document format has streamlined the peer review process yielding test specs of measurably superior quality. One testament to the quality of test was the minimal number of bugs we needed to fix in the first service pack for Microsoft Application Center 2000.

But the advent of model-based testing has provided the greatest surprises, some good, some -- well, not so good. It took almost three years of development to produce a user interface in Word that was easy to use and effective at producing high quality serializable and executable test specifications; but within months of shipping version 2.0 of that UI, test managers made a full commitment to model-based testing. The consequence was that the Word-based UI was obsolete. Now instead of typing dozens of pages in Word, testers spent their time specifying models of the system under test. With the push of a button, those models could spit out thousands of test steps, far more than they could have conceived manually. As a result, functional coverage of tests increased dramatically.

The first pleasant surprise came when we saw how easy it was to use XSLT to transform the XML generated by our model-based testing tool into the runtime XML format expected by our test execution runtime. By relying on a common XML grammar, we were able to follow good object oriented design practices and encapsulate variability; in our case, we encapsulated the variability of test case generators. That is, by producing a common XML schema from both data-driven and model-driven test specification user interfaces, we were able to use either technique without changing the way our test execution runtime worked. As a result, it took less than two months to produce an executable software test specification system that was orders of magnitude more powerful than our previous data-driven system (that took almost three years to develop).

Improved efficiency notwithstanding, even model-generated test cases require peer review, and here is where we saw our second pleasant surprise: with no changes to the original executable software test specification infrastructure, we were able to write an XSLT transform that could convert the runtime XML data generated by the model into the same HTML that previously rendered IML data. The result was that even model-generated test cases could be reviewed the way data-driven test specifications had been; so all the benefits of peer review accrued to model-based as well as data-driven testers. Indeed, given the additional data provided by the model (and only occasionally provided by testers who manually specified test cases), the review of model-based test cases was better than reviews of most data-driven test specs.

After all these years, testers on our team are beginning to think it's possible to get a firm grip on the testing challenges posed by even our most advanced next generation server management architectures. As one of our best testers once said to me, "You have what I need to be a great tester." I have never received a higher compliment.