Model-Based Quality Assurance of Protocol Documentation: Tools and Methodology

Wolfgang Grieskamp, Nicolas Kicillof, Keith Stobie
Microsoft Corporation

Victor Braberman
University of Buenos Aires

Abstract— Microsoft is producing high-quality documentation for Windows client-server and server-server protocols. The Protocol Engineering Team in the Windows organization is responsible for verifying the documentation to ensure it is of the highest quality. Various test-driven methods are being applied, including when appropriate, a model-based approach. This paper describes core aspects of the quality assurance process and tools that were put in place, and specifically focuses on model-based testing (MBT). Experience so far confirms that MBT works and that it scales, provided it is accompanied by sound tool support and clear methodological guidance.

***Index Terms*—quality assurance process, test-driven, model-based, protocols, Spec Explorer.**

1 Introduction

Protocol documentation is an important part of Microsoft's compliance obligations with the US Justice Department and the European Union. Microsoft is committed to producing high-quality documentation for certain Windows client-server and server-server protocols. These documents enable third parties to interoperate with Microsoft's client and server operating systems. The Protocol Engineering Team in the Windows organization acts as a major stakeholder inside of Microsoft in the quality assurance process for this documentation. More than 25,000 pages of documentation for over 250 protocols have to be thoroughly verified to ensure that they are *accurate*, so that developers can implement protocols from the information they contain plus basic domain knowledge. This significant effort requires the application of innovative methods and tools. The team has devised a methodology called the *protocol documentation quality assurance process* (PQAP), with model-based testing (MBT) as one of its cornerstones. Model-based testing has a decade-long tradition inside of Microsoft. The first tools supporting it were deployed at the end of the 1990s. Various MBT tools exist today in the company, including the product family called Spec Explorer [2][3][4], which was originally developed at Microsoft Research, and today is productized as part of this effort.

This paper gives an overview of the PQAP and its supporting tools. Though MBT has been applied successfully to features and products before [5][2], this is the first attempt to use it in such a large scale and in the context of a business-critical area within Microsoft, and to the best of the authors' knowledge throughout the whole industry. Experience so far confirms that MBT *works* and that it *scales*, provided it is accompanied by good tool support and clear methodological guidance, and backed up by investment in training.

2 Overview of the Process

The basic idea behind the process presented in this article is *test-driven analysis of the documents*. Deriving a functional test-suite from protocol documentation ensures that it is thoroughly studied and that its

normative statements are converted into assertions to be checked against the actual implementation. This formalization process naturally also validates internal consistency. It is furthermore close to the actual task of writing an implementation from the document, simulating a potential implementer's attempts.

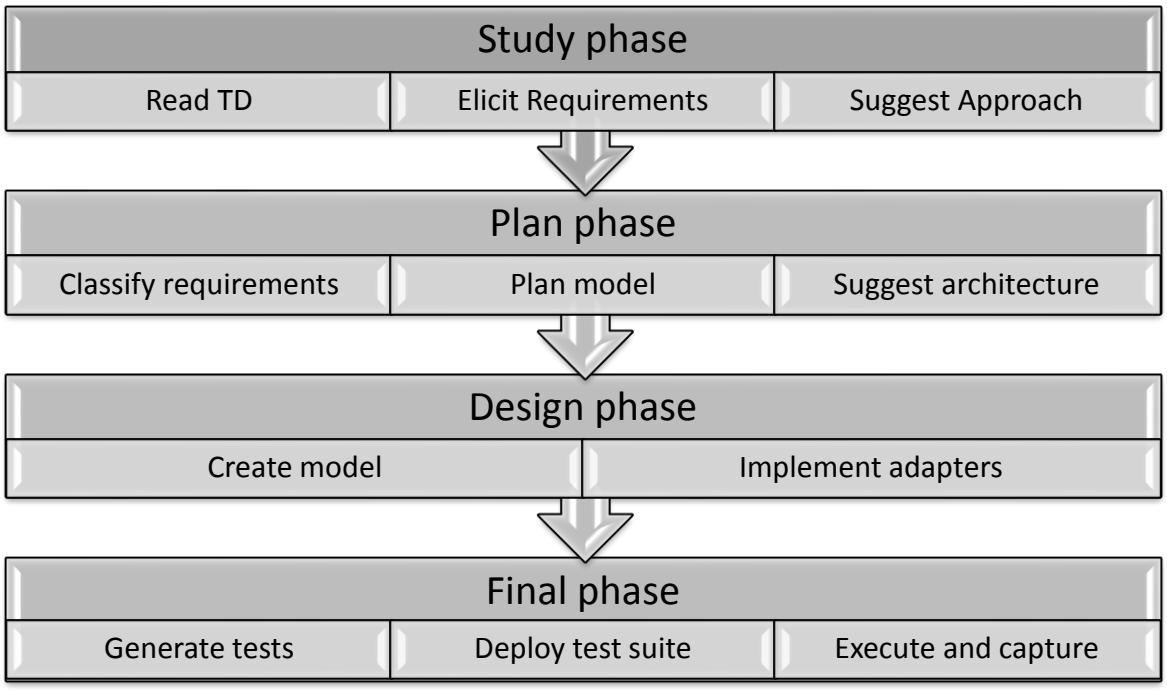
All test suite development in this effort is performed by vendors in India and China. Employing vendors has a significant methodological dimension, since it establishes a clean-room environment that excludes interference of knowledge Microsoft employees may have about protocol implementations. Employing the large number of vendors required for the task also poses significant challenges for the *management* and *control* of the test suite development work itself. This was one of the major design drivers for the PQAP.

This section provides an outline of the PQAP. Later sections drill down on some of its particular aspects, in particular those related to model-based testing.

2.1 Phases of the PQAP

The PQAP is phase-oriented, guiding the development of a test suite for a given protocol from a Technical Document (TD) through all the required steps. Its phases are described in Figure 1.

Figure 1: Phases of the PQAP



Every phase is followed by a formal review. A disposition of Conditional Accept or Re-review by reviewers requires the test suite development team to (partially) reiterate the phase and submit a new review request.

The progress and results of the PQAP are reflected in the PQAR (protocol quality assurance report), a single document to report on all phases of the process. Test suite developers fill in the PQAR incrementally. It is used as a basis for the reviews, and it constitutes the major artifact documenting the test suite. The PQAR is a template-based document with a fixed structure and a number of accompanying guidance documents for the various steps, stages, and phases.

2.2 Requirements Specification

While the PQAR is a central document to the process, the *requirements specification* (RS) is equally important.

Figure 2: A Sample Requirements Specification

Req ID	Doc Sect	Description	Pos	Neg	Derived	Scenario	Scope	Pri	Inform /Norm	Verification
TSCH_R113	2.3.10	The column field of the TASK_XML_ERROR_INFO structure MUST contain the column where parsing failed.				S11,S12	Server	p0	Normative	Adapter
TSCH_R142	2.4.1	The client MUST set the File Version (2bytes, it contains the Version of the .JOB file format) field of the FIXDLEN_DATA structure to 0x0001.	R1102	R1131			Client	p0	Normative	
TSCH_R145	2.4.1	The server MUST ignore the value in the App Name Len Offset field of the FIXDLEN_DATA structure.					Server	p0	Normative	Non-testable
TSCH_R146	2.4.1	The Trigger Offset (2 bytes) field of the FIXDLEN_DATA structure MUST contain the offset in bytes within the .JOB file where the task triggers are located.	R1102	R1131			Client	p0	Normative	
TSCH_R1332	3.2.5.4.6	Upon receipt of the SchRpcGetSecurity call, the server MUST return S_OK on success.				S16, S17	Server	p0	Normative	Test Case
TSCH_R1333	3.2.5.4.7	The SchRpcEnumFolders method MUST retrieve a list of folders on the server.				S18	Both	p0	Normative	Adapter
TSCH_R1335	3.2.5.4.7	Through the SchRpcEnumFolders method if client requests items 1-10 and then 11-20, the second call MAY return duplicate entries if the folder list has changed in between calls.					Server	p2	Informative	Unverified
TSCH_R1337	3.2.5.4.7	path field MUST contain the full path associated with a folder using the format specified in section 2.3.11.(R117 -R120)	R1364	R1350			Client	p0	Normative	
TSCH_R1350	3.2.5.4.7	Upon receipt of the SchRpcEnumFolders call, the server MUST return the value 0x80070003, the HRESULT version of the Win32 error ERROR_PATH_NOT_FOUND, if the path argument does not name a folder in the XML task store, or if the caller does not have either read or write access to that folder.				S18	Server	p0	Normative	Test Case
TSCH_R1351		Upon receipt of the SchRpcEnumFolders call, the server MUST return the value 0x80070003, the HRESULT version of the Win32 error ERROR_PATH_NOT_FOUND, if the path argument does not name a folder in the XML task store.			R1350:c	S18	Server	p0	Normative	Test Case
TSCH_R1352		Upon receipt of the SchRpcEnumFolders call, the server MUST return the value 0x80070003, the HRESULT version of the Win32 error ERROR_PATH_NOT_FOUND, if the caller does not have read access to that folder.			R1350:c	S18	Server	p0	Normative	Test Case

Figure 2 shows an example of an RS. This document is a table derived by test suite authors from the protocol documentation containing one entry for each explicit or implicit normative statement, which defines a unique identifier for the requirement to be referenced from other artifacts in the process. Requirement gathering does not require previous domain knowledge and is not specific to protocol documentation. It consists in identifying statements in a document that can be proven right or wrong.

The RS specifies a description (usually a verbatim statement from the technical document, sometimes slightly modified to include context information) and other properties used to classify the requirement. The most important here are the protocol endpoint (Scope) to which the requirement associates (usually client or server), whether it is normative (Inform/Norm), and which approach is taken to validate it (Verification).

The Verification column in the RS is used to express several (mutually exclusive) verification decisions through keywords. It is left blank for informative requirements. For a normative requirement, gatherers have to ask themselves whether the requirement can be tested. Requirement testability is based on the fact that test suites produced focus on interoperability and hence only aim at controlling or observing behavior over the wire (at the protocol transport level). In addition to such requirements, protocol documentation also contains guidance for the implementation of endpoints that cannot be externally observable, which is considered *Non-testable* for the purposes of the PQAP. If a requirement is testable, testers may choose for a variety of reasons not to actually test it, and consequently mark it as *Unverified*, although this needs to be done sparsely to keep the overall requirement coverage percentage within acceptable limits (e.g. over 80% for most protocols). Finally, if a requirement is testable and will be verified, the decision has to be made to test it in a *Test Case* (either manually written or generated from a model) or in a test *Adapter*. Typically, behavior or data that depends on previous history is validated in a test case (e.g. “packet A follows packet B”, or “field F contains the value previously written by packet C”). Requirements pertaining to data format are validated in an adapter (e.g. “the contents of field F must be of type T”).

Test suites usually play the role of one endpoint in order to test another one. For example, a client test suite is used to test a server implementation as a system under test (SUT).

Test plans given in the PQAR refer to the RS; so do models and test suites; and also test logs generated during test execution. This allows tracking back from test execution results to the technical documentation via the requirements. Requirement coverage is also the main measurement of the completeness of a test suite.

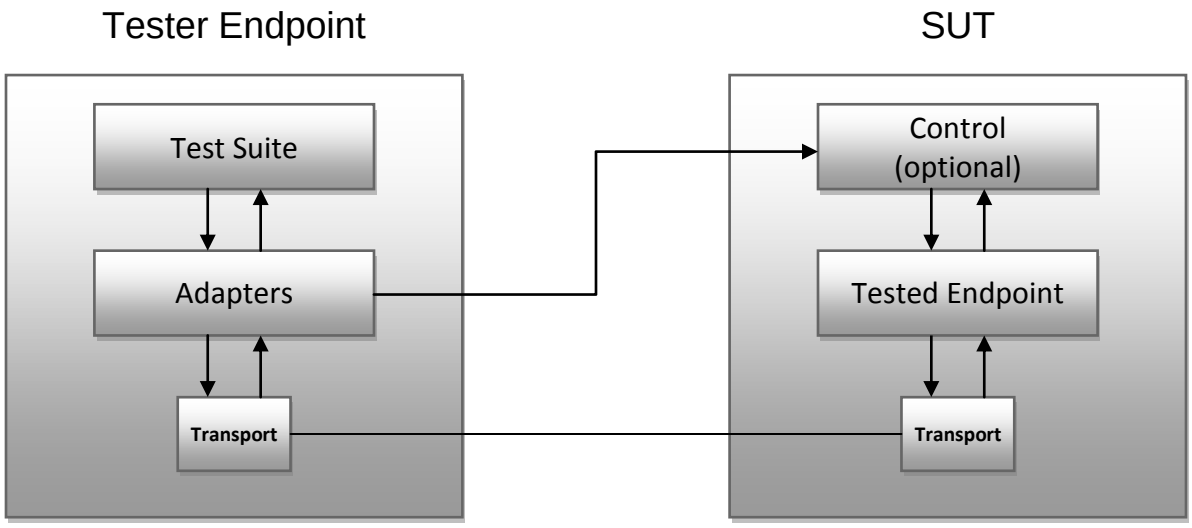
2.3 Testing Approach

Test approach definition starts in the study phase and is refined and finished in the plan phase. It covers two major aspects: how to harness the system-under-test, and which approach is used to define test cases.

2.3.1 Approach to harnessing

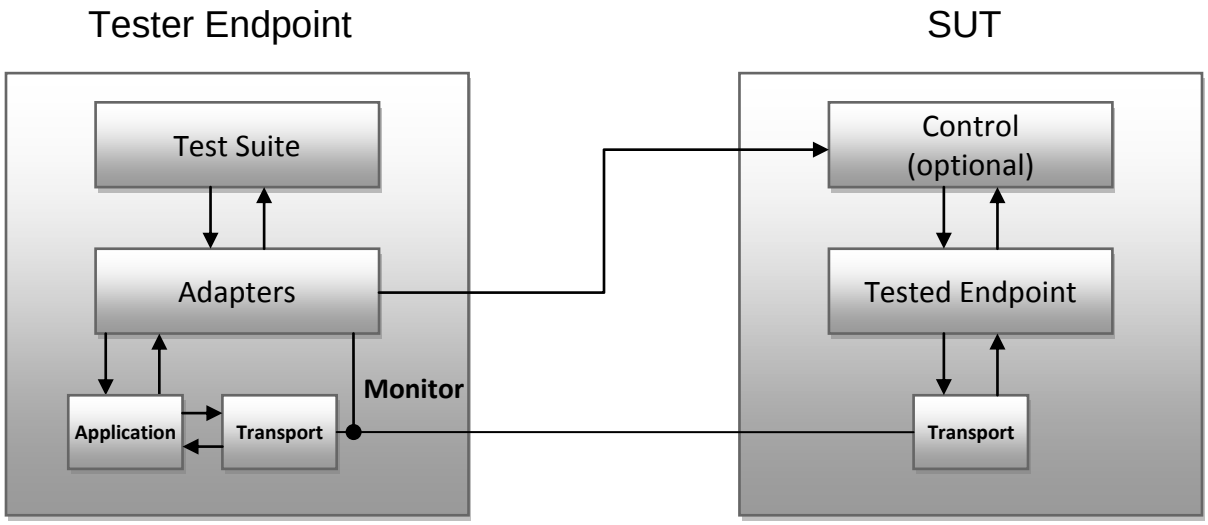
One of two general approaches is usually applied to harness a protocol for testing. In the first approach, one protocol endpoint is completely replaced by a synthetic endpoint, consisting of the test suite and adapters, which drives and monitors the tested endpoint. Typically, the tested endpoint is thereby a protocol server, but in general it can be also a protocol client. This approach is depicted in Figure 3.

Figure 3: Replacing One Endpoint



In a second approach, an existing component or application is used to generate protocol traffic, monitored and validated using a network monitor, as shown in Figure 4.

Figure 4: Driving an Existing Application



Both approaches use test adapters (described in Section 3) to abstract the access to the SUT and harness for the actual test cases. Both approaches may require, in addition to the operations provided by the tested protocol, some additional control of the tested endpoint, shown in both figures as "Control".

The preferred approach is to completely replace an endpoint for testing, since driving an existing application introduces additional challenges. First, the monitor might not be able to interpret encrypted traffic on the protocol transport. Second, an existing application may not produce all behaviors required to achieve suitable test coverage. However, engineering a synthetic testing endpoint is overly complex or even not possible at all for some protocols, in which case testers fall back to the approach in Figure 4.

Note that the chosen approach for harnessing does not affect the one used for test case definition: both harnessing approaches can be used for model-based as well as traditional testing.

2.3.2 Approach to defining test cases

The PQAP does not prescribe whether to use traditional (manually coded) or model-based testing for defining test cases. Rather, test suite developers are asked to submit a suggestion for the approach after the study phase, to be considered by reviewers.

It is rather common to apply a hybrid approach, where model-based testing is used to develop the main test suite, but some special scenarios are tested by manually written tests. The hybrid approach is supported by a unique test adapter that can be used both from models and from traditional test suites. Test adapters are defined in the managed *Protocol Test Framework* (PTF), an extension to Visual Studio Test Tools, and are discussed in more detail in the next section.

3 Test Representation and Test Adapters

One of the major problems for developing test suites for protocols is to get protocol elements on the wire and back. Elements can be data packets in various encodings, remote procedure or web service calls, and so on. The problem is well known in the protocol testing community, resulting in efforts like TTCN-3.

For this project, an extension to Visual Studio Test Tools (VSTT) has been developed, called Protocol Test Framework (PTF), based on the Visual Studio Unit Testing framework and .Net. Using a mainstream programming system constitutes an advantage, since it allows leveraging a high-level programming environment and IDE, a language well known to test engineers working in the project (C#), and access to the variety of features provided by the .Net framework, Microsoft's major platform for interoperable code. The Unit Testing framework, in addition, provides a concise way to represent test cases and to manage their execution under VSTT.

PTF adds to VSTT custom support for dealing with protocols, including ways to automatically serialize and de-serialize data packages based on declarative definitions in .Net types and attributes, and access to RPC calls directly from .Net. In the rare cases where necessary, Visual Studio's C++/CLI subsystem, which allows mixed managed/native development, is used to deal with issues such as access to the Windows API or other low-level system levels. In general, the usage of C/C++ in adapter and test code is discouraged; as languages like C# are considered to generate simpler and more robust test suites.

Test adapters are a central concept of PTF. A test adapter defines an interface to the protocol under test or its harness at a problem-oriented level of abstraction. It is given as a managed (C#) interface that contains methods for sending data to the server, and events for receiving data back.

Figure 5: Test Adapters

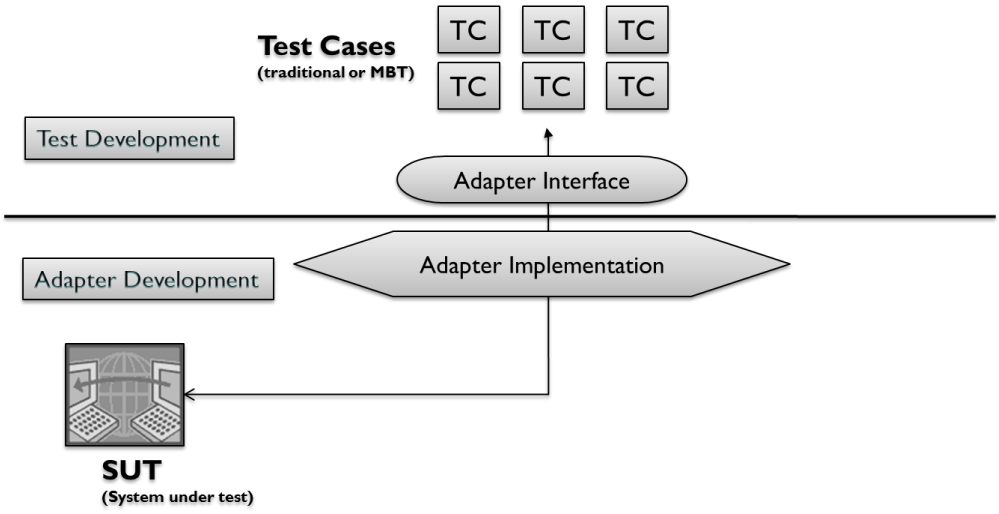


Figure 5 depicts the role of test adapters in the process. Test adapter interfaces are developed and used independently of their implementation. At deployment time, test adapter implementations are associated with adapter interfaces, supporting different implementations with the same logical functionality for different protocol transports and/or operating systems.

Like any interface in component-oriented design, an adapter interface is a grouping of related methods following the usual principles of high cohesion and low coupling. The PQAP includes guidelines to design adapter interfaces. For instance, the level of abstraction for adapter interfaces should be potentially useful both for a model and a traditional testing approach. Test validation steps usually happen in the actual test cases, but some checks can also be performed by the adapter. The general guideline is that protocol behavior should not be encoded in the test adapter, whereas state-independent data invariants on protocol elements can well be validated there.

Test adapters are thus specifically designed for each protocol so that they provide to test cases a high-level interface, abstracting away details about data encoding and formatting. Packet fields or parameters that are only relevant to message-local data format validations, such as buffer sizes or encryption flags are typically hidden from the test cases (or model) and resolved at the adapter level for communication in both directions.

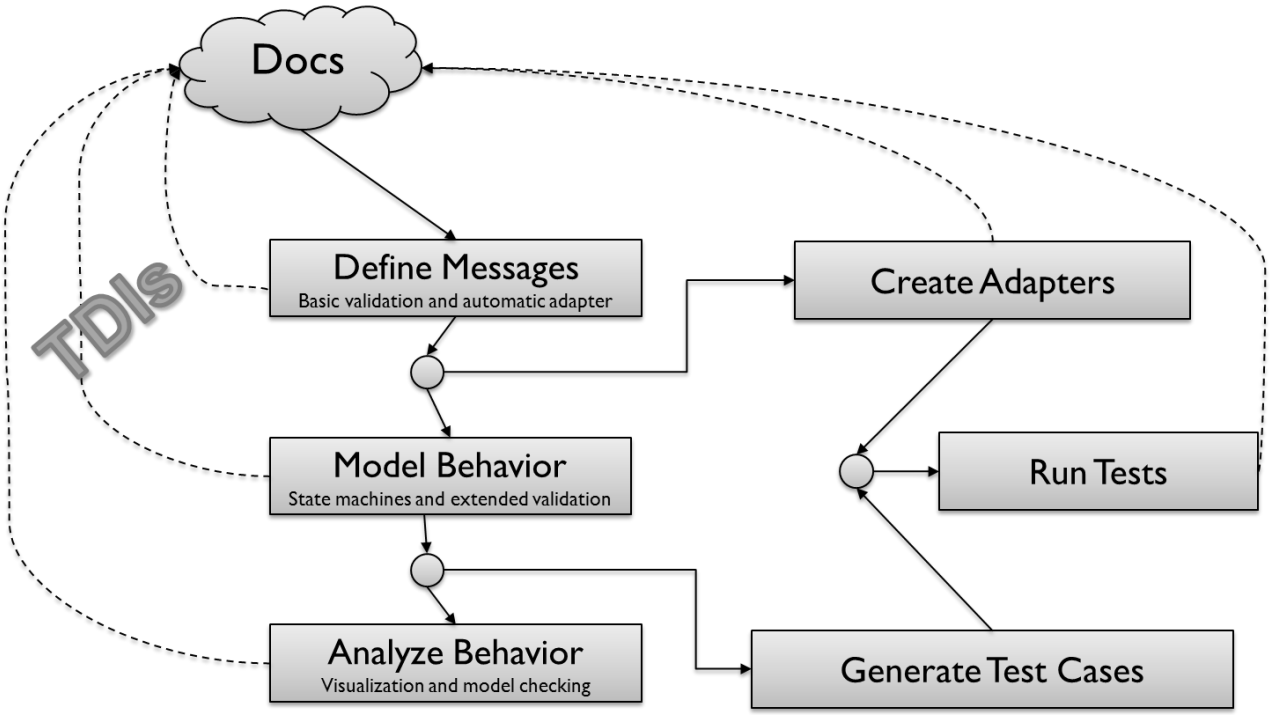
An interesting example of data abstraction by a test adapter is the Abstract Identifier pattern, defined as part of the PQAP. The pattern is applied to information coming from the SUT that cannot be determined at modeling or test case coding time, such as identifiers or encryption keys. The pattern guides testers to maintain a table mapping concrete values for these data points to abstract (integer) values that are used for the communication with the test cases. A test case (or model) will thus expect the identifier for the first session it establishes to be simply 0, while the second session will be 1. The adapter will add an entry to the table when the first concrete identifier is received from the SUT mapping it to value 0. From then on, a 0 received from a test case for a session identifier parameter will be sent over the wire as the actual identifier, and this session identifier will be sent to the test case as 0 whenever it is received from the SUT.

Test adapters can be implemented in diverse ways in PTF. It is possible to wrap an existing API or program, or to synthesize packets and directly parse them from the wire. The last approach is supported by the Protocol Adapter Compiler (PAC), a tool that comes with PTF, which extracts definitions of protocol elements directly from the protocol document, and generates data structures ready to be consumed by the PTF framework. Protocol element serializers and de-serializers can be thus directly generated from the technical document with a minimal amount of human intervention.

4 Model-Based Test Suite Development

Model-based testing has been discussed extensively in academic and industrial communities. For a comprehensive text book see e.g. [10]. The general workflow of how MBT is applied as part of the PQAP is shown in Figure 6.

Figure 6: MBT in the PQAP



Dotted lines represent feedback towards the document, in the form of TDIs (Technical Document Issues – documentation bugs). As seen in Figure 6, TDIs are generated in most steps. Over 10,000 TDIs have been found as a result of applying the PQAP, of which more than 8,000 have been resolved as fixed so far (acknowledged as documentation issues and corrected). Typically, more than 50% of the TDIs in the project are actually detected before any test is ever executed, confirming community wisdom that the value of MBT is not only in the testing itself but in the mere development of the model.

In the context of the project for which the PQAP was developed, any discrepancy detected between the actual implementation and the documentation is considered an error in the latter, which must conform to the shipped version of the product. Nothing prevents the same process to be applied in a setting where the documents are considered a "golden" specification and conformance failures are thus treated as product bugs that need to be fixed.

In fact, model authoring usually requires test-suite developers to not only consider the SUT's expected behavior (oracle), but also forces them to think in generic (parametric) terms what the tester endpoint (SUT environment) may generate as valid stimuli for the SUT. Traditional test cases, although also useful in practice, do not necessarily push the envelope of that kind of analysis, which is especially rewarding in terms of detecting issues in the documentation.

The use of MBT is strongly recommended when protocol behavior is history dependent. History dependency may either come from modalities of allowed actions or from heavy data dependencies of actions from the current SUT data state. In all those cases, models tend to closely mimic technical document statements about protocol endpoints. The technical document usually describes a protocol action as a set of conditions under which the action is enabled in terms of an abstract data state model, plus a change (update) of this state when the action happens. This (informal) pre/post-condition description maps directly to the modeling style used in the project.

4.1 Spec Explorer

The model-based testing tool used in the project is Spec Explorer. It has been developed as the third incarnation in a series of MBT tools [4][3]. Its conception and foundation has been described in previous papers [2][6][8]. An introduction to its application in the protocol documentation domain has been given in a preliminary version of this paper [11]. Here, that introduction is expanded by providing details and samples to illustrate each of the phases in the methodology.

Spec Explorer is a modeling and model-based testing environment originally developed at Microsoft Research to overcome some of the obstacles that prevent a broader adoption of these techniques in the software development lifecycle. It is fully integrated into the development environment of Visual Studio, supporting multiple notations (programs, diagrams) and modeling styles (state machines, scenarios), with an emphasis on the capability to combine those different behavioral descriptions using model composition. It is internally based on the framework of action machines [6], which allows for uniform encoding of models stemming from a variety of notations.

4.2 Model Programs

The core notation used in Spec Explorer for state-oriented modeling derives from Abstract State Machines (ASM) [9]. Models are described in a programmatic style by guarded update rules on a global data state. The resulting artifacts are called *model programs*. Rules describe transitions between data states and are labeled with actions corresponding to method invocations on a test adapter or on the actual SUT. Rules can be parameterized (and their arguments then also appear in the associated action labels).

Model programs have similarities with extended finite state machines (EFSM) in the way they are written, but there are substantial conceptual differences. First, the machine defined by a model program is usually not finite, as the state can consist of complex data elements like collections, objects that can be dynamically created, etc. The *slicing* of an infinite state space into a finite subset for the purpose of state exploration and test generation is therefore a central aspect of MBT with Spec Explorer, discussed in Section 4.2.1. Second, inputs and outputs are distinguished in state transition labels, and multiple outputs can be enabled in one state allowing for specification and/or implementation non-determinism, the semantic framework for test conformance is not based on finite state machine test theory, but on transition systems and alternating simulation [15]. Detailed discussions of Spec Explorer's notion of conformance can be found in the literature [3][6].

4.2.1 Example Model Program: Chat Room Server

As an example for demonstrating modeling in Spec Explorer, a chat room server is used. This simple example has sufficient complexity to illustrate the usage of rich (infinite) model state and alternating simulation as a conformance notion.

The chat room server allows users to logon, logoff and post messages to be broadcast to all users in a session. Message broadcasting has to obey certain basic ordering constraints: the messages posted by one user must arrive in the order they have been emitted; however, messages posted by different users may arrive in any order.

The state of the chat room server is defined in the following C# fragment:

```
enum UserState
{
    WaitingForLogon,LoggedOn, WaitingForLogoff
}
class User
{
  internal UserState state;
  private MapContainer<int, SequenceContainer<string>> waitingForDelivery;

  internal bool HasPendingDeliveryFrom(int senderId) {
    return waitingForDelivery.ContainsKey(senderId);
  }
  internal bool HasPendingDeliveryFrom(int senderId, string message) {
    return HasPendingDeliveryFrom(senderId) &&
           waitingForDelivery[senderId].Contains(message);
  }
  internal string FirstPendingMessageFrom(int senderId) {
    return waitingForDelivery[senderId][0];
  }
  internal void AddLastMessageFrom(int senderId, string message) {
    if (!HasPendingDeliveryFrom(senderId))
      waitingForDelivery[senderId] = new Sequence<string>();
    waitingForDelivery[senderId].Add(message);
  }
  internal void ConsumeFirstMessageFrom(int senderId) {
    if (waitingForDelivery[senderId].Count == 1)
      waitingForDelivery.Remove(senderId);
    else
      waitingForDelivery[senderId].RemoveAt(0);
  }
}
static MapContainer<int, User> users = new MapContainer<int,User>();
```

The static variable `users` describes the entire state of the model. It is a mapping (dictionary) from user identifiers to objects representing user state. Each user object consists of a value indicating the user's current status, plus a queue of messages that have been posted to the user but have not yet been delivered. This queue is again a mapping from identifiers of senders who have posted messages for the given user to an ordered sequence of message contents for each sender. This representation allows for two levels of non-determinism. On the one hand, the user who will next receive a broadcast message in the `users` mapping is not fixed. On the other hand, the sender who will be selected for the given user in the `waitingForDelivery` mapping can also vary.

The listing below shows guarded update rules for two actions of the system. The first action stands for posting a message to the room. This is an input stimulus to the SUT. The second one represents receiving a message. This is an output response from the SUT. At the model program level, this difference is not relevant, as rules just state when these actions are enabled and what will be their effect on model state.

```
[Rule]
static void BroadcastRequest(int userId, string message)
{
  // enabling condition
  User user = GetLoggedOnUser(userId);
  // update
  foreach (User loggedOnUser in users.Values)
    loggedOnUser.AddLastMessageFrom(userId, message);
  }
}

[Rule]
static void BroadcastAck(int userId, int senderId, string message)
{
  // enabling condition
  User user = GetLoggedOnUser(userId);
  Condition.IsTrue(user.HasPendingDeliveryFrom(senderId));
  Condition.IsTrue(user.FirstPendingMessageFrom(senderId) == message);
  Requirement.Capture(5, “Messages from one sender MUST be received in order”);
  // update
  user.ConsumeFirstMessageFrom(senderId);
  if (EveryoneReceived(senderId, message))
    Requirements.Capture(7, “All logged-on users MUST receive a broadcast”);
}
static User GetLoggedOnUser(int userId)
{
  Condition.IsTrue(users.ContainsKey(userId));
  User user = users[userId];
  Condition.IsTrue(user.state == UserState.LoggedOn);
  return user;
}
static bool EveryoneReceived(int senderId, string message)
{
  return !users.Exists(u => u.Value.state == UserState.LoggedOn &&
    u.Value.HasPendingDeliveryFrom(senderId, message));
}
```

The enabling condition of an action is declared in the C# statement `Contracts.Requires`, which calls a particular modeling library. In contrast to earlier versions of Spec Explorer based on an extension of the C# language called Spec# [3], all the infrastructure for modeling is provided in the current version by libraries; no language extensions are required. `Contract.Requires` statements can appear anywhere in the body of an update rule and even in helper methods (e.g. helper method `GetLoggedOnUser`). If the Boolean condition passed as an argument is false in the current state, then the model execution path containing the call is cut off, and any state updates performed until that point are forgotten.

In this example, posting a message (`BroadcastRequest`) is only enabled for logged-on users, and causes the message to be distributed to the queues of all users registered in the room. Receiving a message (`BroadcastAck`) is also only enabled for logged-on users, and for the first queued messages associated with the given sender. If these conditions are met, the message is consumed from the queue.

The model captures two requirements in the rule for action `BroadcastAck`. Requirement 5 ("Messages from one sender MUST be received in order") is declared to be captured by fulfilling the enabling conditions of the rule, which restrict the implementation to acknowledge the reception of the first message received from one sender before other messages from the same source are consumed. Requirement 7 ("All logged-on users MUST receive a broadcast") is considered to have been captured only when the last receiver has consumed the message. Both validations rely on the model behavior imposed by the rule for action `BroadcastRequest`. Validation of requirement 5 is based on the fact that messages are added at the end of the queue. Capturing requirement 7 leverages the knowledge that each message is added to all logged on users.

Note that the above model program is unbounded in various respects.

- First the `message` parameter to a post is not fixed to a finite domain. (Note that, given a number of users logged on to the system, all other parameters of the update rules *are* fixed indirectly by the constraints that relate them to model state: user identifiers can only be drawn from the domain of the global user mapping or the delivery map of an individual user, as ensured by the enabling condition through a `ContainsKey` call).
- Second, there is no bound on how often a user can post messages and when these posted messages are actually received.

So the model program state is obviously infinite. The next section will describe how this is dealt with.

4.3 Slicing

Slicing of unbound model state is a central activity when using Spec Explorer. Even for a bound finite model state, slicing is a way to extract manageable subsets from an otherwise huge and intractable finite state space.

Slicing relies on the assumption that, from a testing perspective, a user can make the explicit decision to prune away some of the potential *stimuli* (inputs) accepted by the SUT. This does not alter the correctness of test results; it just produces fewer tests. In contrast, a user should never prune away *responses* (outputs) as this may result in tests that fail after slicing, although they would have passed in the unsliced full model. Henceforth, (proper) slicing is a human intervention that may impact test coverage and completeness, but not correctness.

Mastering the "art" of slicing is the key to mastering state explosion and making model-based testing practical. Several Spec Explorer features support the user in this activity. They are intended to be applied not as part of the model program itself – which is supposed to represent the pure contract of the behavior and shall remain unmodified – but from the outside, using a configuration and coordination scripting language to augment the model program. This language, called Cord, has been first described by Grieskamp and Kicillof [8].

Cord allows configuring a model program for slicing and test generation in the following basic ways:

- **Parameter selection and combination**: allows assigning parameter domains to actions, including domains derived from the dynamic model state; as well as defining combinatorial goals like pairwise and n-way interactions.
- **State predicate filtering**: allows setting bounds on states by pruning state space exploration from states that do not satisfy a Boolean predicate defined on the state variables.
- **Trace pattern filtering**: allows restricting the model graph to traces matching a certain pattern, where the pattern is given in a regular-expression style, which may be mixed with control-flow-dependent state predicates.

- **Coverage-based reduction**: when applied to an already finite model graph, allows restricting it to a smaller sub-graph that still guarantees certain model coverage criteria. A typical criterion is requirement coverage, where capturing of requirements by individual actions is declared in the model program with a special statement.

The main principle followed by practitioners (besides never pruning away output responses) when creating slices together with their models is to maintain a basic separation of concerns between expected system behavior and test purposes. A model has to be correct in itself (no spurious states or transitions) representing all potential protocol behaviors, and only those, independent of slices. Modelers cannot assume that the model will be used only in combination with the provided scenarios, as other users may want to define different scenarios for checking or testing different properties.

Two mechanisms prevent modelers from accidentally slicing away requirements that their model was supposed to capture. A static check is enforced when applying coverage-based reduction in combination with other forms of slicing. The user must specify the requirements that should be captured either by identifier or by count. The reduction algorithm will check that the requirement coverage goal is achievable by comparing the set (or count) provided by the user with those reachable in the exploration graph. A more general dynamic check is applied by a coverage reporting tool that collects information from logs after running the test suites against an implementation. This tool compares the set of requirements contained in execution logs with those declared in the Requirements Specification and outputs a human-readable report with enumerations, counts and percentages of requirements verified versus those extracted from the documentation.

Among these techniques, a closer look will be taken, in the context of the chat room server example, at trace patterns, which turn out to be the most widely used mechanism in current practice.

4.3.1 Example Trace Pattern: Chat Room Server

As discussed earlier, the state space of the chat room server model program is unbounded, as it allows both arbitrary values for the message parameter and an arbitrary number of broadcasts. Using a trace pattern and composing this with the actual model program, a finite trace can be extracted.

This can be done in Spec Explorer's Cord language, for example, as follows:

```
machine BroadcastSlice() : Actions
{
    LogonSequence; BroadcastRequest(_, "a"); BroadcastRequest(_, "b");
    BroadcastAck*
||
    ModelProgram
}
```

The above machine composes a trace pattern with the actual model program using the parallel composition operator ||, which forces both of its operands to synchronize in lockstep (the transitions possible in the composition are those and only those that can be performed by both operands). The trace pattern consists of a logon sequence (described in another machine, including logging on two users to the system), followed by two broadcast requests, and then by an arbitrary number of broadcast acknowledgments. Note that parameter values in the trace pattern are only partially fixed, as is the control flow, allowing any possibilities determined by the model program in the composition result.

Figure 7: Excerpt from the state exploration graph of sliced Chat

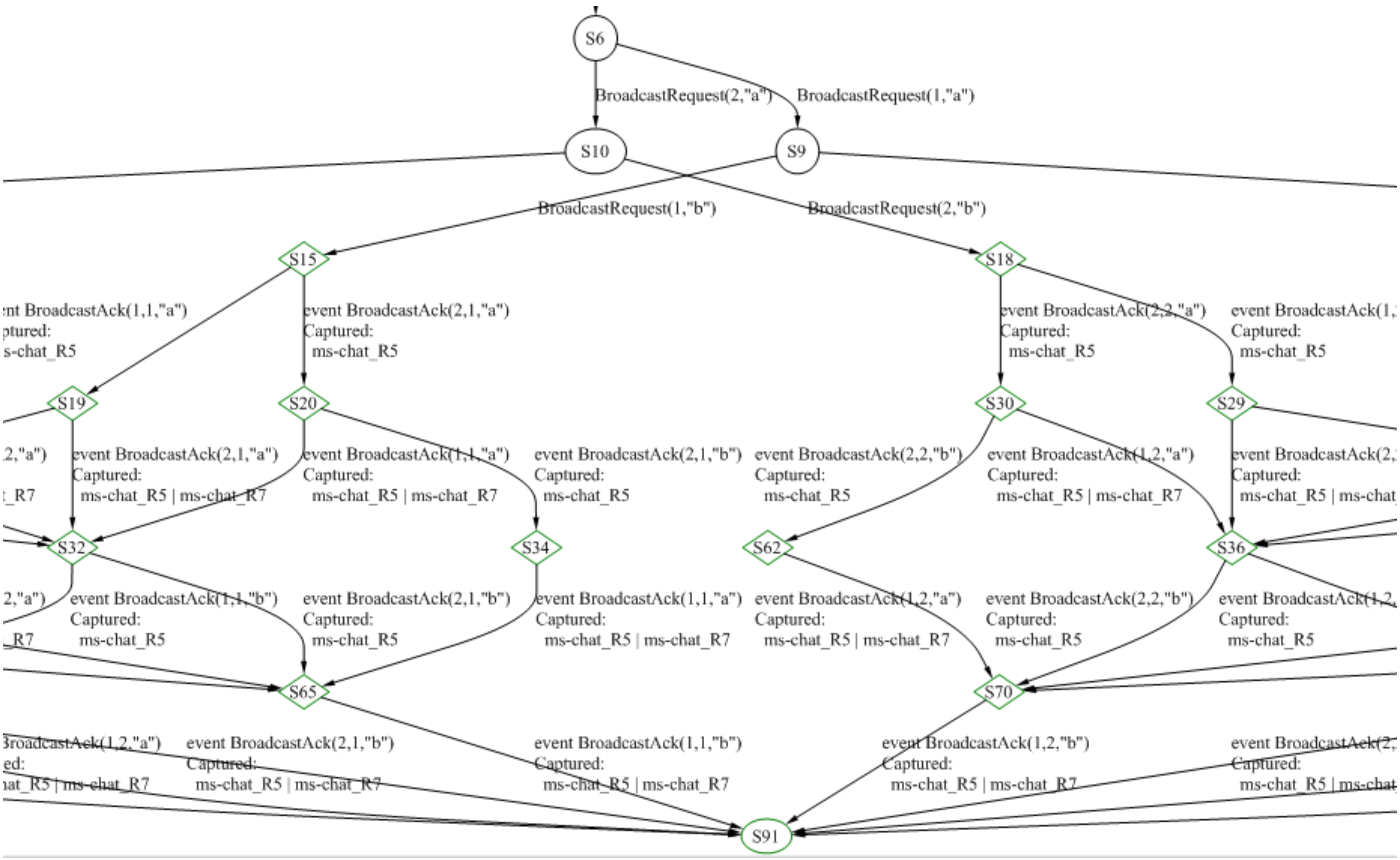


Figure 7 shows an excerpt of the graph generated by Spec Explorer for machine `BroadcastSlice`. The snapshot corresponds to the case where the same user broadcasts two messages, which results in a strict ordering. Requirement capture can be seen in transition labels (e.g. "Captured: ms-chat_R5"). The part of the graph not shown covers the case where different users broadcast, and the number of possible orderings is therefore higher. Spec Explorer shows states as nodes with two different shapes. Ellipses stand for input states (states in which a test case would send a stimulus to the SUT). Diamonds are output states (states in which a test case would expect a response from the SUT).

Note that Spec Explorer's notion of test conformance using alternating simulation implies that output responses of the system under test are buffered; henceforth it is possible to first issue a number of stimuli and then analyze the responses.

Application of trace patterns go beyond extraction of finite behaviors for testing. In general, slices are also used for model analysis. With a well-defined slice, a user can visually validate whether the model presents the expected behavior. Slices can also be used for model-checking. To that end, a user can define a trace pattern ending in an unwanted state, and check whether the resulting composition with the model program is non-empty, in which case the graph of that composition represents the counter examples. Cord's trace-pattern syntax supports this by allowing Boolean conditions on state variables mixed into the patterns.

4.4 Test Generation and Execution

Spec Explorer supports both on-the-fly testing, where model exploration is fused with testing, and generation of test code that can run standalone, independent of the model. In the protocol document testing

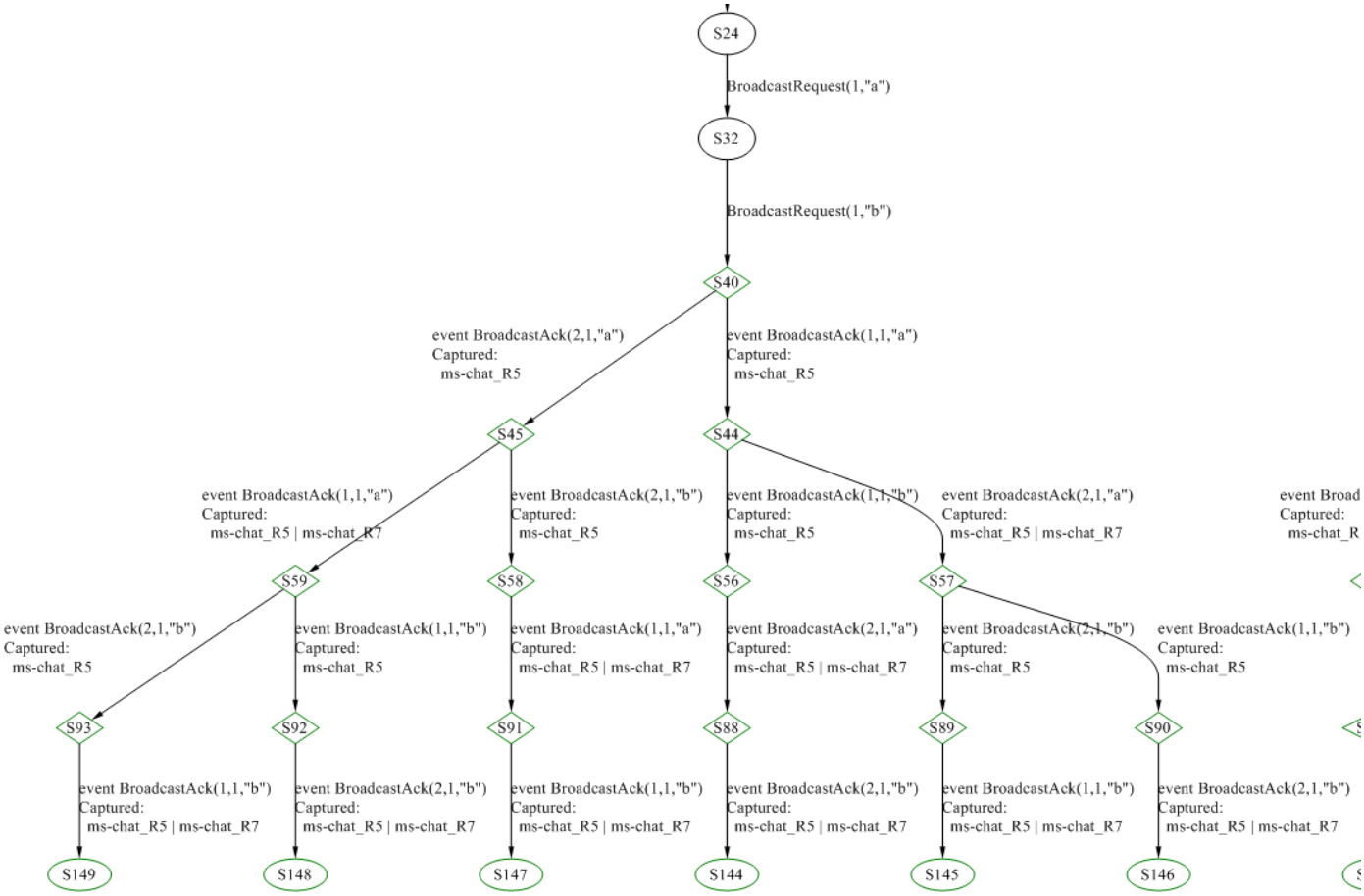
context, only the test code generation feature is used, since it allows for systematic and explicit engineering of test suites and does not depend on random exploration strategies.

Generated test suites are represented by default in the VSTT Unit Testing format, but the code generation infrastructure can easily be reconfigured to target other test frameworks via provided extension points.

Before test suite code can be generated, test selection strategies must be applied in the form of traversals on the exploration graph. The most widely used strategy is transition coverage. In Spec Explorer, traversals are transformations on the model graph. They result in a new graph that fulfills the coverage goal of the test selection strategy and is in Test Normal Form: every state has at most one stimulus outgoing transition. This means that all testing choices (which stimulus to provide to the system at each point) have been made before test-execution time. System choices, on the other hand, are preserved by traversals, as they represent potential non-determinism on the part of the tested system. Test code generated for a graph containing such a choice will accept any of the potential system responses at that point, and proceed accordingly.

This principle is illustrated in Figure 8, which contains the graph for one of the test cases generated from the sliced model in Figure 7, using All-Transitions as a coverage criterion. For this test case, the algorithm has made the testing choices resulting in user 1 broadcasting both messages. Other test cases cover the case where user 2 broadcasts both messages and those where each user broadcasts one message in both possible orders. The traversal algorithm has however left system choices intact (in states S40, S44, S45, S49 and S57). The resulting generated code will consequently accept all possible system responses in these states.

Figure 8: Test case generated from the sliced Chat



For systems with unfair non-determinism (some non-deterministic paths might never be chosen by specific implementations), and models for which no "winning strategy" (meaning covering any transition is possible independent of choices made by the SUT) exists, test generation as it is currently performed may result in test suites missing possible coverage. The solution for this issue is subject of ongoing work, and needs to be treated by moving the traversal computation for these particular systems to test-execution instead of test generation time.

4.5 Requirement Tracing

Tracing requirements gathered from an informal specification all the way to test-run reports is an integral aspect of the PQAP. Spec Explorer supports declaratively associating requirements to preconditions and updates by calling specific library methods in C# code. Requirements covered in each step (and in the path leading to each state) are recorded as part of exploration results and transferred to generated test code. At test-run time, dynamic requirement capturing is logged and output as part of test reports. In addition, PTF provides direct mechanisms for adapter and traditional test code to explicitly log the requirements they cover. These combined features provide requirement coverage information all along the process, an essential feedback to determine the correctness of slicing, to analyze exploration and model checking results, to debug individual test case execution and to interpret global test suite verdicts.

4.6 Adoption

When the protocol documentation QA effort was started, the biggest concern for rolling out a model-based testing approach in this context was whether this technology can be effectively adopted by lay people or is just confined to formal methods experts. Currently, almost all of the Windows protocols to be tested were done by vendors performing the test development. This has built confidence, empirically established, that the MBT approach works and that it is more productive than traditional testing. This evidence is backed by the fact that, in addition to MBT, traditional testing techniques are being applied to some documents in the same domain with the same people.

The key factors for adoption seem to be providing systematic training for new hires, and a fairly self-supporting critical-mass community growing at vendor sites. As an example, the vendor team in China hired many professionals with limited testing experience directly out of college; after attending a three-day ramp up training, the majority became capable of modeling simple protocols; and after three months of working in teams with experienced colleagues, they were able to smoothly model complex protocols by themselves. The biggest challenge for adoption appears to be in the paradigm shift, not actually in mastering the complexity of the modeling problem and tools. This results in inexperienced people performing relatively better during ramp up than those bringing in mature patterns and practices from other paradigms as baggage. Thus, the experience indicates a steep but not very high learning curve.

5 Effort Measurements and Comparison

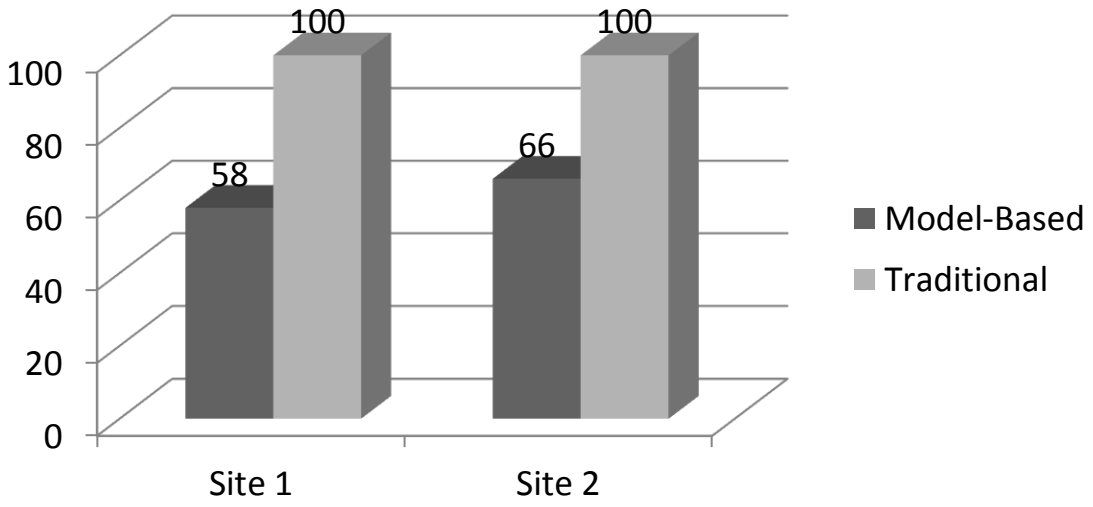
Table 1 shows the time spent performing several tasks in the protocol documentation testing project, in terms of the *total* requirements identified in the documents. The sample includes all 105 documents tested in one of the testing sites, as a way to eliminate any potential bias introduced by differences in effort recording and distribution of work between sites.

Statistical analysis indicates that the number of requirements is a good predictor for the effort, with significant correlation coefficients at the 0.01 level with respect to other indicators, such as number of pages in a document (0.867), testable requirements (0.881), requirements covered in test cases (0.690), requirements covered in adapter (0.410), number of modeled test cases (0.485) and number of traditional test cases (0.216).

Table 1: Effort per Total Requirement

Task	Average person-days per requirement
Technical document review	0.13563
Requirement gathering	0.10444
Model authoring	0.05803
Traditional test coding	0.06760
Adapter development	0.15014
Test case execution	0.07469
Final adjustments	0.04429
Total test suite development	**0.635409**

Figure 9: Effort Reduction using MBT



Preliminary analyses show a significant advantage when comparing test suites that were developed using MBT with the remaining test suites (see Figure 9). The average effort per *tested* requirement for test suites where MBT was applied was 1.39 person-days, whereas testing each requirement in test suites not applying MBT took in average 2.37 person-days. (Note the difference between total requirements used in Table 1 and tested requirements used for the effort comparison. Requirements related to the behavior of a client were collected but not tested in the project.) That resulted in a 42% productivity gain from the usage of MBT the test site also listed in Table 1, with a similar number of requirements tested with each approach (9,844 requirements in test suites using MBT; 8,728 requirements in non-MBT test suites).

While the comparison between approaches yields a lower performance improvement (34%) for the other site, the team in that location has applied MBT in most of their test suites, resulting in a much less balanced distribution of requirement coverage. Only 2,411 out of 15,892 were tested in test suites not using MBT, which renders the comparison for this site less significant.

Notice that the effort strictly invested in model authoring according to Table 1 is relatively low compared with the overall efficiency gain from applying MBT. This is due to the fact that modeling has an impact on all phases of the process, not just on model creation. Also, the table measures the effort in terms of total requirements (as an early estimate of future phases), while the comparisons above are in terms of tested requirements, a magnitude known only after a test suite is concluded.

6 Related Work

To the authors' knowledge, there is no comparable project in industry or Academia applying model-based testing in a comparable scale and with a scope similar to those of the project described here. The closest is a project conducted by ISPRAS for the testing of the Linux operating system, using the UniTESK toolset [16]. In that project, besides other activities, POSIX standards are translated into models to generate conformance tests for different Linux versions. However, the goal is mostly API testing, rather than network protocol testing. Moreover, the model and test suite development is not part of an industrial scale process. From the same group also comes work for deriving formal specifications from standards [23], applying a process for gathering requirements from an English document similar to the PQAP's. However, this work does not yet apply a full tool chain down to model-based testing from these requirements.

On the tools level, material about Spec Explorer has been published and compared with other work before. A summary is provided here, but the reader is referred to previous articles [2][3][6][8] for a more detailed comparison. Spec Explorer is often classified as finite state machine testing, though there are only some commonalities: notably that its guarded-update machines are similar to those found in extended finite state machine notations, and that similar traversal techniques for the final step are applied in both domains. The approach is actually closer to labeled transition systems (LTS) based testing and IOCO [17], except that Spec Explorer uses alternating simulation with buffering as the conformance notion. Alternating simulation has been introduced by Alur et al. [15], and its use for Spec Explorer has been described in several papers [3][6]. As of today, Spec Explorer seems still to be the only MBT tool which uses alternating simulation together with output response buffering as the underlying conformance notion. In the authors' opinion this has advantages over IOCO, as it does not require input completeness of the implementation and therefore treats model and implementation symmetrically.

Methodologically, the usage of trace patterns for slicing, as has been proposed for Spec Explorer first by Grieskamp and Kicillof [8] (at that point in time, trace patterns were simply called scenarios) has proven to be essential for the success of the model-based testing approach in the context of the PQAP. The approach by itself is not new, and has been employed in tools like TorX [18], TGV [19] and others before. However, these tools have special languages for defining test purposes that are applied to the test generation algorithm. Spec Explorer supports trace patterns not as a special-purpose language, but just as another modeling notation. The application of a trace pattern to a model program is therefore treated as a general model composition problem. Consequently, trace patterns can be used for model checking or independent exploration as well.

Spec Explorer uses symbolic state space exploration techniques and constraint solving, based on the XRT software model checker [20] to deal with partial models (such as trace patterns), model programs with unfixed parameter domains, and their composition. Symbolic execution in software model-checkers for testing has been proposed for Java Pathfinder [21], however, the authors are not aware of any model-based testing tools stemming from this conceptual work. Conformiq Qtronic [22] is a commercial model-based testing tool that uses symbolic execution, although it does so for on-the-fly testing and not for slicing.

7 Conclusion

This article presents the quality assurance process for protocol documentation established at Microsoft. It uses novel methodologies and techniques in various dimensions: a test-driven approach applied in general to the protocol documentation, a detailed methodology for applying this approach in a large-scale vendor context, and the employment of advanced testing technology such as model-based testing with Spec Explorer. It contributes to Microsoft's intention to deliver high quality documentation to the community, which enables third parties to create interoperable server products.

There are numerous instances of model based testing being successfully used in the industry, but to the authors' knowledge, none has the scale of the one presented here. The magnitude of the current effort is expected to help model-based testing become main-stream in the software industry. The feasibility and scalability of MBT is evident in the fact that the project has delivered "model to metal" test suites for over 75 protocols, and this number is growing. At the end of the project, around half of the 250 protocols in scope will have been modeled, reflecting an investment of over 50 person years in model-based testing application alone. In addition a substantial investment has been made in tool development, based on a continuous feedback loop from the test suite development process into the Spec Explorer development team. According to a preliminary statistical analysis, the application of MBT resulted in a 42% productivity gain when compared with traditional test suites in a site where similar numbers of requirements have been verified with each approach.

An aspect to note of this success story is that the application of MBT in the PAQP is confined to medium-sized systems. Protocols can be considered to be, based on their complexity, somewhere between embedded systems and general software systems. Typically, model programs developed in this project consist of up to 1000 lines of C# code (while many are significantly smaller) with a dozen or more slices. From the first prototype to successfully running MBT tests, test suite teams require around 1 to 2 person-months. Slicing is essential even for these medium size systems, as the state space is too large for exhaustive test generation. With the aid of slicing, there is no technical reason why MBT should not scale to larger systems.

Ramping up engineers without a background in formal methods to do model-based testing has worked well in the context of the PQAP. The experience indicates that while ramping is steep because of new and unacquainted concepts, the final skills necessary to master the technology are not particularly challenging. They may in fact be much more moderate than professional software engineering requires, indicating that model-based testing can become a mainstream technology in testing.

Acknowledgements

This work would not have been possible without the tireless efforts of the Protocol Engineering Team, in Redmond, Hyderabad, and Beijing, and with process and reviewing assistance from Robert V. Binder and MVerify. Thanks also go to colleagues at Microsoft Research who contributed to create the Spec Explorer technology: Colin Campbell, Yuri Gurevich, Lev Nachmanson, Wolfram Schulte, Nikolai Tillmann and Margus Veanes. The authors would like to particularly thank The Technical Committee and especially Harry Saal, as well as the EU Monitoring Trustee Neil Barrett and his staff, who have evaluated and influenced this work.

References

[1] Microsoft. MSDN protocol documentation (http://msdn.microsoft.com/en-us/library/cc216514.aspx)

[2] Wolfgang Grieskamp. *Multi-Paradigmatic Model-Based Testing*. Invited talk in Klaus Havelund and Manuel Nunez and Grigore Rosu and Burkhart Wolff, FATES/RV, LNCS 4262, 2006.

[3] Colin Campbell, Wolfgang Grieskamp, Lev Nachmanson, Wolfram Schulte, Nikolai Tillmann, and Margus Veanes. *Model-based testing of object-oriented reactive systems with Spec Explorer*. Formal Methods and Testing 2008, LNCS 4949, Springer, 2008, pp. 39-76.

[4] Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. *Generating finite state machines from abstract state machines*. In Proceedings of ISSTA'02, volume 27 of Software Engineering Notes, pages 112–122. ACM, 2002.

[5] Keith Stobie. *Model based testing in practice at Microsoft*. In Proceedings of the Workshop on Model Based Testing (MBT 2004), volume 111 of Electronic Notes in Theoretical Computer Science. Elsevier, 2004.

[6] Wolfgang Grieskamp, Nicolas Kicillof, Nikolai Tillmann. *Action Machines: a Framework for Encoding and Composing Partial Behaviors*. International Journal of Software Engineering and Knowledge Engineering 16(5): 705-726, 2006.

[7] Jonathan Jacky, Margus Veanes, Colin Campbell, Wolfram Schulte. *Model-based software testing and analysis with C#.* Cambridge University Press, 2008.

[8] Wolfgang Grieskamp and Nicolas Kicillof. *A schema language for coordinating construction and composition of partial behaviors*. In Proceedings of the 28th International Conference on Software Engineering & Co-Located Workshops – 5th International Workshop on Scenarios and State Machines. ACM, May 2006.

[9] Yuri Gurevich. *Evolving Algebras 1993: Lipari Guide*. In E. Boerger, editor, Specification and Validation Methods, pages 9–36. Oxford University Press, 1995.

[10] Mark Utting and Bruno Legeard. *Practical Model-Based Testing.* Morgan-Kaufmann, 2007.

[11] Wolfgang Grieskamp, Dave MacDonald, Nicolas Kicillof, Alok Nandan, Keith Stobie, and Fred Wurden. *Model-Based Quality Assurance of Windows Protocol Documentation*. In Proceedings of the 1st IEEE International Conference on Software Testing (ICST 2008), Lillihammer, Norway, April 2008.

[12] C. Willcock, T. Deiss, S. Tobies, S. Keil, F. Engler, S. Schulz, *An Introduction to TTCN-3*, Wiley, 2005.

[13] A. Grinevich, A. Khoroshilov, V. Kuliamin, D. Markovtsev, A. Petrenko, V. Rubanov, *"Formal Methods in Industrial Software Standards Enforcement*". In I.Virbitskaite, A.Voronkov, eds. Proceedings of PSI'2006, LNCS 4378, pp.456-466.

[14] M. B. Dwyer, G. S. Avrunin, J. C. Corbett: *Patterns in Property Specifications for Finite-State Verification*. ICSE 1999: 411-420

[15] R. Alur, T. A. Henzinger, O. Kupferman, M. Y. Vardi: *Alternating Refinement Relations*.In *Proceedings of the Ninth International Conference on Concurrency Theory* (CONCUR), LNCS 1466, Springer, 1998.

[16] Linux verification center at IPSRAS, http://www.linuxtesting.org.

[17] J. Tretmans. *Test generation with inputs, outputs and repetitive quiescence*. Software, Concepts and Tools, 17(3):103-120, 1996.

[18] 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.

[19] J. Fernandez, C. Jard, T. Jeron, 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

[20] W. Grieskamp, N. Tillmann, and W. Schulte: *XRT - Exploring Runtime for .NET - Architecture and Applications*. In *SoftMC 2005: Workshop on Software Model Checking*, Electronic Notes in Theoretical Computer Science, July 2005.

[21]S. Khurshid, C. S. Pasareanu, and W. Visser: *Generalized symbolic execution for model checking and testing*. In Proc. 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pages 553–568, April 2003.

[22] Antti Huima: *Implementing Conformiq Qtronic.* In *Testing of Software and Communicating Systems* (TestCom/FATES 2007), LNCS 4581, Springer 2007.

[23]A. Grinevich, A. Khoroshilov, V. Kuliamin, D. Markovtsev, A. Petrenko, V. Rubanov, "Formal Methods in Industrial Software Standards Enforcement", in I.Virbitskaite, A.Voronkov, eds. Proceedings of PSI'2006, LNCS 4378, pp.456-466.