Model-driven Service Integration Testing - A Case Study

Sebastian Wieczorek and Alin Stefanescu and Andreas Roth
SAP Research
Darmstadt, Germany
{*name.surname*}*@sap.com*

***Abstract*—This paper presents a case study for the modeling and model-based testing (MBT) of enterprise service choreographies. Our proposed MBT approach uses proprietary models called Message Choreography Models (MCM) as test models. The case study illustrates how MCM-based service integration testing allows to formalize design decisions and enables full integration into an existing industrial test infrastructure by using the concepts of domain specific languages and model transformations. Further, the MBT tools integrated into the testing framework have been compared based on one concrete use case.**

***Keywords*-Model-based Testing; Enterprise Systems; Service-oriented Architecture; Case Study; Service Choreographies**

I. Introduction

Service-oriented architectures (SOA) provide frameworks and methods to compose single services in order to realize complex business scenarios. At the lower end, a single service is described as a set of operations and message types, its functioning relying on a simple request-response pattern. Modeling and implementation of single services using standards like XML, SOAP, and WSDL is well mastered both in theory and practice. At the service integration level, more complicated specifications are needed to capture not only the message exchange and the underlying message types, but also the dependencies between these exchanged messages, i.e., both control-flow and data-flow dependencies. Thus, the challenging part of the SOA-based development lies in integration of different services according to the defined business processes. While SOA adoption is gaining pace towards becoming mainstream [1], the need of SOA quality assurance becomes an activity of paramount importance, with SOA testing filling a central spot. While single service testing is usually well researched [2], [3] and consistently deployed in practice, the field of service integration testing poses several new challenges [4]. The difficulties to be overcome are due to the heterogeneity, high distributivity, dynamicity, and loose coupling of the service-based systems.

The case study presented in the paper has been carried out in the context of SAP. Being a leader in the area of business software, SAP also delivers SOA via its service-enabled software (e.g., SAP Business ByDesign[1], SAP Business Suite[2]) and its SOA-based, open technology platform SAP NetWeaver[3]. The Enterprise SOA developed in a model-driven way at SAP has a dozen types of models containing modeling information of business objects, deployment units, service components, service interfaces, integration scenarios, business process variants, and service choreographies [5]. It contains several thousands of services and several hundreds of choreographies and tens of millions of lines of code implementing them. Testing such a huge software system involves thousands of testers using several types of testing. It is essential that powerful automation tools and techniques are used to address the sheer complexity of the system. MBT [6] is an ideal candidate to use the high-level information from the SOA models for test generation.

In order to tackle the challenges of service integration testing in such industrial setting, in previous work we developed a domain-specific language (DSL) that can be used to derive integration tests, called Message Choreography Modeling (MCM) [7]. Further, we created a testing framework that realizes model-based service integration testing by incorporating different MBT tools [8], [9]. The main contribution of this paper is to evaluate our modeling and model-based testing approach for service integration in the SOA development. First, the applicability of the approach in the industrial setting is verified. Second, its efficiency and effectiveness were investigated.

The paper is structured as follows. Section II sketches the SOA development approach that is targeted by the case study. Section III describes the design and context of the case study. The case study execution, including the modeling and test generation, is described in Section IV. Section V explains the results of the case study, while Section VI summarizes the paper.

II. Development Process

As in every customer oriented scenario, a SOA development process starts with the definition of user and market requirements. For Enterprise SOA applications, functional requirements are described by the business processes that have to be supported. As illustrated in Figure 1, this top-down approach of software development, starting with a high level description of requirements, can be combined easily with the concepts of model-driven development and

[1]http://www.sap.com/businessbydesign
[2]http://www.sap.com/solutions/business-suite
[3]http://www.sap.com/platform/netweaver

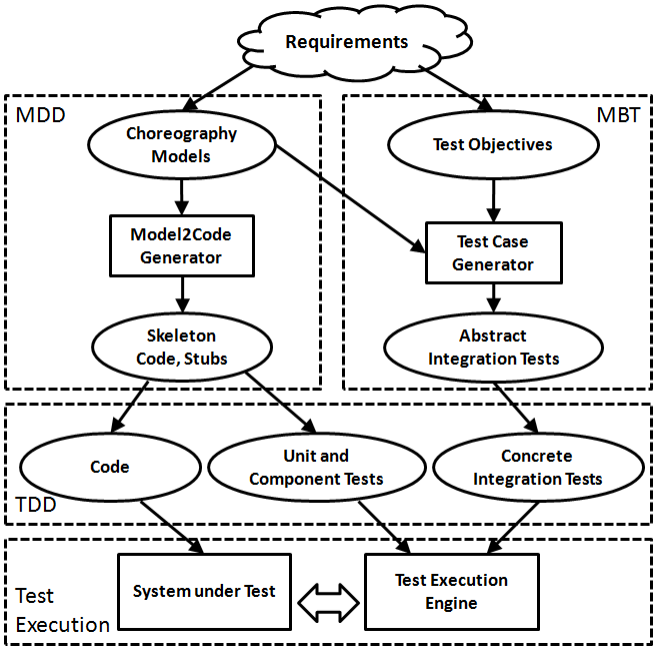

Figure 1. Envisioned development and testing process for SOA integration

MBT, where general specifications of a system are stepwise refined by adding relevant domain specific information. In the following, an overview of the depicted development process is given, with a focus on service integration.

Model-driven Development (MDD): By performing the design steps of MDD, the initial requirements are gradually refined into development models. These comprise of structural models, identifying and connecting the service components, and behavioral models of the business process flow. Having a formal definition of the communication protocol allows applying automatic model verification and validation techniques. For example the absence of deadlocks, livelocks, inconsumable messages, or local enforceability can be proven at this early development stage.

Applying pure MDD techniques in the continuing development process would mean that the development models are further refined, with the ultimate goal to automatically derive code. However, various unsolved challenges have been identified (cf. [10], [11]) for the industrial application of MDD concepts for code generation, such as: lacking tool support for model-level debugging, lacking user expertise, and lacking support for versioning and merging of models. Consequently, the industrial application of MDD concepts on low abstraction layers is usually bypassed.

Test-driven Development (TDD): Instead, as illustrated in Figure 1, the development models can be used as input for a *Model2Code Generator*, enabling automated generation of skeleton code and stubs, with the intention that the generated code will be refined manually. After automatically generating code stubs from the structural models, developers start to create tests for the functions they have to implement (i.e., unit and component tests). In this way, the developers are able to validate their own code automatically by running these tests.

Model-based Testing (MBT): As depicted in Figure 1, integration testing is carried out in parallel to the development of the software components. A recent study [12] shows that a key factor of success is to apply continuous integration throughout the development. The core of continuous integration is to combine and try out the developed functionality very frequently in order to spot problems as early as possible. Especially for applications whose components are loosely coupled, as it clearly is the case for SOA, tests of the communication and interaction are vital and therefore should be integrated into the TDD cycles.

MBT approaches promise to effectively support automatic test generation for service integration. By applying MBT in the integration phase, not only the effort for test case generation can be decreased, but also test coverage and overall test effort can be controlled in an easy and transparent way. Further, model-based integration testing provides the means of carrying out continuous integration, as the test cases can be generated even before the first line of code is written. Hence, integration testing can be carried out throughout the development.

Test Execution: The considerations that lead to manual programming instead of automatic code generation also apply to the test concretization of the generated test cases. The missing modeling support on lower abstraction layers hinders the industrial application of fully automatic test case generation. Therefore, especially the provision of appropriate test data is currently the responsibility of the testers. In the context of SAP, a tool called Test Data Migration Server (TDMS) exists, which supports this activity by deriving consistent reference data for testing from customer systems. If these data samples are available, testers are able to choose the appropriate input for each test case from that source, otherwise they have to create it.

For test concretization, a mixed approach is used (according to the nomenclature of [6, ch. 8]). The manual refinement of the abstract test cases follows the keyword-driven testing principles. Keyword-driven testing (or action-word testing) uses action keywords in the test cases, in addition to data. Each action keyword corresponds to a fragment of a test script (the adapter code), which allows the test execution tool to translate a sequence of keywords and data values into executable tests [6]. The test execution environment is provided by the SAP Test Workbench.

III. Case Study Design

In this section, we first describe the context of the case study and then the planned case study activities.

A. Setting

We start by shortly presenting the system used in the case study and afterwards, the users participating in the case study are characterized.

System under Test: The concepts described below have been developed in collaboration with the SAP product group that is responsible for the development of the SOA-based solution SAP Business ByDesign. This solution is created on top of SAP NetWeaver, which provides a SOA technology platform that includes a modern messaging infrastructure [13]. At the time when the case study was conducted, Business ByDesign was quite mature in terms of functionality and quality and was already released to selected customers. Currently, Business ByDesign is freely available in many countries.

The design of SAP Business ByDesign has been captured by various modeling artifacts, based on proprietary SAP languages [5]. For the development of the service components, structural information has been provided. These models includes interface descriptions for each service, component integration models specifying which service components are connected, and class descriptions for the objects inside a service component. The implementation based on these models was carried out by distributed development teams. The applied development process consisted of periodic implementation phases, interleaved with testing and documentation activities.

Case Study Participants: Two groups of users have been involved in the case study. Their background and relevant qualification is given below:

- *Integration experts* have the task to coordinate the integration of service components during development. They have a good understanding of the communication processes between service components and are experienced with structural modeling. As behavioral modeling was not extensively used for the development of SAP Business ByDesign, the integration experts did not have much exposure to such concepts.
- *Integration testers* are deriving and executing test cases for service component integration testing based on functional descriptions of the system. These descriptions are provided in natural language by developers (i.e. technical documentation) and business analysts (i.e. customer requirements). They are trained and experienced to use proprietary testing tools, but do not have any MBT expertise.

B. Approach

In the initial planning of the case study, we intended to follow the whole service integration process, as described in Section II, i.e., from producing MCM models as choreography descriptions down to executing the derived test scripts. However, as the targeted SAP product itself already entered the final testing phase, the envisioned utilization of MCM models in the implementation phase was omitted from the scope of the case study. Further, the availability of the above described participants was very constrained, which made it necessary to conduct the case study in a guided fashion.

The case study consisted of *four use cases*. Below we describe the performed modeling and testing activities.

Modeling: The identification of the four suitable use cases (pilots) was mainly driven by organizational considerations and hence rather random. It was intended that for each pilot an integration expert conducts the choreography modeling autonomously on a stable version of the MCM editor after having some additional modeling guidelines and initial training. However, due to the mentioned time constraints, it was planned that the first draft of MCM models would be sketched in *guided sessions of* 1 *hour per pilot*, followed by a consolidation and refinement phase conducted by the authors. Afterwards the models were planned to be validated by the pilot users in *another guided session of* 1 *hour*.

Testing: There were three test generators [8], [9] used: (1) an in-house FSM (Finite State Machine) based generator using classical graph-coverage algorithms like the Chinese Postman algorithm, (2) an MC (Model-Checking) based generator using a model-checker for test generation, that tackles also extended finite state machines features like guards and side effects on global variables, and (3) an open-source heuristic-based test generator called MbtTigris[4].

For each of the four use cases, one test suite was generated and concretized. First, the generated abstract test cases were used to automatically generate and load test scripts into the test environment of the development teams. Second, the corresponding MCM model and automatically generated UML message sequence charts for each test case have been supplied for each pilot. Third, the testers were asked to concretize the generated test scripts autonomously by adding test data information and to execute them on the system under test.

Analysis: As described, the aim of the case study was to gain evidence that the described SOA development approach is applicable, efficient and effective in an industrial setting. To derive the results, two sources of information are used:

1) The supervision of the case study execution by the authors allowed to gather some unbiased observations (e.g., execution time of the test generation, number of uncovered faults).
2) Participants were questioned in semistructured interviews. This interview technique, which is mixing open-ended and specific questions, was chosen, because we assumed that according to [14], the responses might lead to further discussions, which better captures information of such unknown structure.

[4]http://mbt.tigris.org

IV. Case Study Execution

In the previous section, an overview of the case study context and activities have been provided. In this section, information about the case study execution, divided into the modeling and testing activities is given. Afterwards, for one of the pilots, the test generation is described explicitly in order to compare the different MBT tools that are integrated into the testing framework.

A. Modeling

According to the plan described in Section III, the creation of the pilot models was conducted in 2 *guided sessions that lasted about* 1 *hour*. In addition to that, the authors conducted *another* 2 *hours* of refinement and consolidation of the results.

After the second session, semistructured interviews were conducted with the pilot users. The response was very positive. The participants perceived the possibility to formally describe the design as most beneficial, as it has the potential to ease the communication between development teams of communicating services significantly. Further, the full integration of existing modeling content (e.g. interface and component specifications) was highlighted. The graphical modeling approach using a state-based representation was generally perceived intuitive,

On the other hand, the proprietary constraint language for the guards of the transitions usually needed some clarifications. Further, the participants had problems in understanding whether and how their modeling decisions would affect the test generation, which was mainly due to the fact that in their role as integration experts they were not deeply involved in the actual testing process and were unfamiliar with the MBT concepts. However, in the case study the provided guidance mitigated such issues.

B. Test Generation

As described in Section III, for each pilot an abstract test suite has been generated automatically. The generated test suites had sizes ranging *between* 4 *and* 8 *test cases with an average length of* 10 *test steps*. A test step, in this case, refers to the triggering of a message. For the four pilots, two times the FSM-based and two times the MC-based test generator have been used. The heuristic-based test generator was used only extra for comparison.

The participating testers were able to read, understand, and enhance the generated test suites with concrete test data and message triggers, even without having detailed knowledge of the tested integration. *The concretization effort per pilot test suite was estimated to be between around* 4 *hours.* The consequent test execution did not uncover any fault in the development system[5].

[5]However, one bug in the testing framework was found.

After successfully running the test suites, semistructured interview sessions with the pilot teams were conducted. The overall response again was positive. For all pilots, the test generation produced reasonably small test suites. The pilot users had confidence in the quality and completeness of the tests and perceived a design-based test generation as beneficial. In all cases, the generated test suites covered at least the already existing integration tests.

Although it was impossible to compare the concretization effort with the effort of implementing the test cases by hand, all participants agreed that the evaluated approach was time-saving, due to the automatic script generation and the concept of enforcing a high reuse of generic scripts for the test steps. *On average, a saving of 50% was estimated by the pilot users for the test generation and concretization tasks.* Also the seamless integration of the tool into SAP's testing framework and the consequent usability of the test scripts for automatic regression testing was appreciated.

C. Comparison of Test Generators

As mentioned, the identification of the four suitable pilots out of about 200 existing service choreographies was mainly driven by organizational considerations and hence rather random. Surprisingly though, the derived choreography models had relatively equal complexity. All MCMs incorporated some constraints on the exchanged messages, but half of them did not contain dependent transitions (i.e., the side effects on one transition influence the guards on another transition) and hence were suited for FSM-based test generation. In the following, one such pilot is used, as it allows to compare the three test generators.

Figure 2 shows the anonymized MCM of the pilot that is chosen for the description of the case study execution. States are depicted in rounded rectangles, messages in envelopes. Diamond-shaped connectors enable the message sending in a certain state, while arrow-shaped connectors describe the message effect.

The original choreography model of the pilot has the same structure, but the communication is used in a business conversation which is not related to business task management, as implied by the naming of the modeling elements. In the following, the results of applying the three different test generators mentioned in Section III are described.

FSM-based generation: Applying the FSM-based test generator to the given pilot, the following test cases are derived from MCM. The computation was carried out in *less than* 1 *millisecond.*

Test 1: *Create, Stop, Revoke, Release, Close, Restart, Block, Unblock, Close.*

Test 2: *Create, Change, Delete.*

Note that the implemented algorithm provides a test suite that covers the model with the minimal number of transitions. Sometimes, the FSM-based test generation generates a test suite contains few but long test cases and this was the

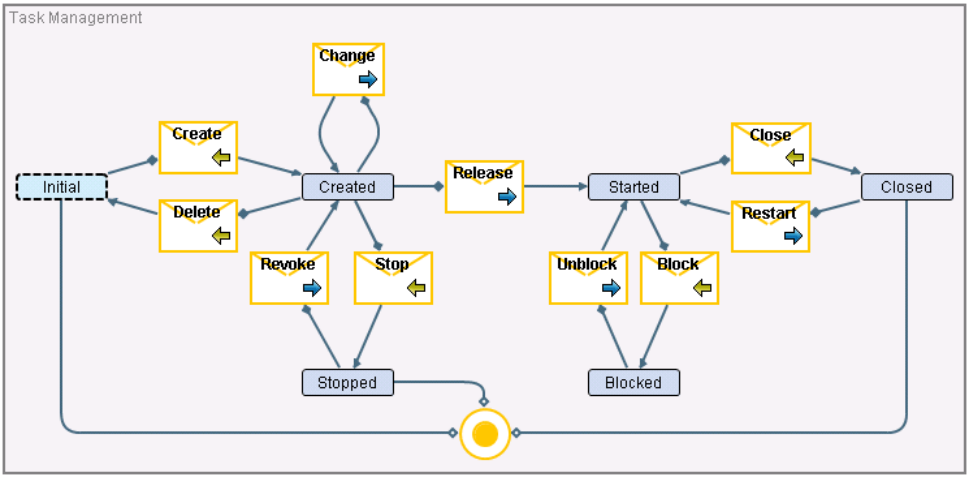

Figure 2. The MCM of an example pilot

case above. The problem with long test cases is that they are much harder to maintain and to debug in case of errors. Also, FSM-based generation may generate infeasible test cases when applied to choreography models with dependent transitions. Therefore its use is limited.

MC-based generation: The MC-based test generator provides a test suite that covers all the transitions of the model minimal number of transitions. The resulting test suite for the given pilot is given below and has been *produced in about* 0.4 *seconds*:

Test 1: *Create, Release, Block, Unblock, Close.*

Test 2: *Create, Release, Close, Restart, Close.*

Test 3: *Create, Change, Stop, Revoke, Delete.*

Given the fact that the MC-based algorithm implemented a breadth-first graph traversal strategy, the test suite contains 1 more test case and traverses 3 more transitions than the FSM-based suite. However, the longest test case of the suite contains only 5 transitions, compared to 9 transitions in the FSM-based suite.

A disadvantage of this approach is that typically the model-checking based test generation suffers from the state space explosion problem. For pilots with a higher complexity (e.g., containing more states and integer variables), advanced tuning of the model checker was necessary to produce a result in acceptable time. This might be a problem for testers with no formal methods background.

Heuristics-based generation: When applying the heuristics-based test generator to the given pilot and asking for a test suite that covers all transitions or terminates after 100 steps, the following results can be obtained after 10 runs.

- All runs terminated, because a test suite with full transition coverage was found.
- The average test suite contained 46 test steps, distributed over 5 to 6 test cases.
- The average execution time to generate a test suite was 3 seconds.
- After optimization, the average test suite contained 21 test steps, distributed over 2 test cases.

It can be seen that this test generation approach does not produce optimal results. However, it has the advantage to always return a result in acceptable time. For the other pilots of the case study, it was possible to find test suites with at least 90% transition coverage on average with equal settings, while the computation time was constantly 3 seconds per generated test suite.

Results: Each test generation method has its strengths and weaknesses in certain context. After the performed comparison we can envisage the following strategy for choosing between them: "Use the FSM-approach in case the transitions of the MCM are independent of each other. If this is not the case, the MC-based approach may be applied. However, if MCM is too complex and the MC-based approach does not terminate, a test suite could finally be obtained by using the heuristic-based approach." However, note that the third case does not guarantee optimal results.

V. Discussion

The aim of the case study was to gain evidence for the applicability, efficiency and effectiveness of the described MCM approach. In the following, these three points are discussed.

Applicability: The case study showed that it was possible to model randomly chosen service communications of a SOA-based product using MCM. These results imply that MCM is expressive enough to capture relevant service communication. Further, these choreography models were suitable to automatically derive test suites that could be concretized and executed. According to the pilot users, the test suites were covering all tests that had been created manually in the previous testing.

Efficiency: As described in Section IV, there has been positive feedback regarding the automated test generation and the utilized reuse concept of test scripts. The time saving potential of the reuse concept was further demonstrated when concretizing all the generated test suites. After deriving the first executable test suite, the generic reuse of concretized test steps allowed to run the other test suite after *only* 10 *minutes of minor adaptations*. This implies that extending previously generated test suites or applying test generators with more complex coverage criteria will only increase the automatic test execution but not the semi-automatic concretization effort.

Effectiveness: The fact that no fault could be discovered during for the four pilots has various reasons. First of all, the targeted system was already tested rigorously and had even been shipped to pilot customers, who heavily used them. Further, the choreography models have been created in collaboration with the development teams that had implemented these choreographies. Therefore, the discovery of misinterpretations of the initial requirements were unlikely. Consequently, no clear judgment over the effectiveness based on the case study is possible.

However, in the interviews that followed the case study, the pilot users stated that the generated test suites were covering all manually derived integration test cases. These statements are hard to prove, because the manual test cases are not directly contained in the generated test suites and have not been created in relation to a choreography model. Nevertheless, these statements indicate that the testers greatly believe in the effectiveness of the approach.

VI. Conclusion

In this paper, we presented a detailed case study of application of MBT into an industrial SOA domain. Our experiment confirmed the applicability and resource-saving potential of the described approach. These positive results were convincing enough to allow the continuation of the application of MBT on a larger scale at SAP, which is now in progress. This is in line with other accounts from research (see [6]) and industry (see e.g. [15] for the positive MBT experiences at Microsoft). Given the new concepts of the SOA domain and especially the complex data structures and test databases, there are still challenges ahead. We are currently working on test data modeling adapted to the Enterprise SOA domain, that should further increase the automation degree in the proposed MBT approach.

Acknowledgment

This work was partially supported by projects Modelplex[6] and Deploy[7] (EC-grants no. 034081 and 214158).

[6]http://www.modelplex-ist.org

[7]http://www.deploy-project.eu

References

[1] R. Heffner, "Across all vertical industry groups, the majority of SOA users are expanding its use," Forrester Research, Research Report, May 2009.

[2] L. Baresi and E. Di Nitto, *Test and Analysis of Web Services*. Springer, 2007.

[3] W.-T. Tsai, Y. Chen, R. A. Paul, H. Huang, X. Zhou, and X. Wei, "Adaptive testing, oracle generation, and test case ranking for web services," in *29th Int. Computer Software and Applications Conference (COMPSAC'05)*. IEEE Computer Society, 2005, pp. 101–106.

[4] G. Canfora and M. D. Penta, "Service-oriented architectures testing: A survey," in *Software Engineering: International Summer Schools, ISSSE 2006-2008, Revised Tutorial Lectures*. Springer-Verlag, 2009, pp. 78–105.

[5] S. Kätker and S. Patig, "Model-driven development of service-oriented business application systems," in *Business Services: Konzepte, Technologien, Anwendungen*. sterreichische Computer Gesellschaft, 2009, vol. Band 1, pp. 171–180.

[6] M. Utting and B. Legeard, *Practical model-based testing, a tools approach*. Morgan Kaufmann, 2007.

[7] S. Wieczorek, A. Roth, A. Stefanescu, V. Kozyura, A. Charfi, F. M. Kraft, and I. Schieferdecker, "Viewpoints for modeling choreographies in service-oriented architectures," in *Proc. of 8th IEEE/IFIP Conference on Software Architecture (WICSA'09)*. IEEE Computer Society, 2009, pp. 11–20.

[8] S. Wieczorek, V. Kozyura, A. Roth, M. Leuschel, J. Bendisposto, D. Plagge, and I. Schieferdecker, "Applying model checking to generate model-based integration tests from choreography models," in *Proc. of the 21st IFIP Int. Conf. on Testing of Communicating Systems (TESTCOM'09)*, ser. LNCS, vol. 5826. Springer, 2009, pp. 179–194.

[9] A. Stefanescu, S. Wieczorek, and A. Kirshin, "MBT4Chor: A model-based testing approach for service choreographies," in *European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'09)*, ser. LNCS, vol. 5562. Springer, 2009, pp. 313–324.

[10] S. Teppola, P. Parviainen, and J. Takalo, "Challenges in the Deployment of Model Driven Development," in *Proceedings of the Fourth International Conference on Software Engineering Advances (ICSEA'09)*. IEEE, 2009.

[11] A. Uhl, "Model-driven development in the enterprise," *IEEE SOFTWARE*, vol. 25, pp. 46–49, 2008.

[12] T. E. Murphy, "Using continuous build to drive quality," Gartner Research Report No. G00166848, 2009.

[13] M. Krimmel and J. Orb, *SAP NetWeaver Process Integration*, 2nd ed. Galileo Press, 2009.

[14] C. Seaman, "Qualitative Methods in Empirical Studies of Software Engineering," *IEEE Transactions on Software Engineering*, vol. 25, no. 4, p. 557, 1999.

[15] W. Grieskamp, "A success story for model-based testing: Microsoft's protocol documentation quality assurance program," Invited Talk at TESTCOM/FATES'09 Conference, 2009.