# Enabling Model-Based Testing for SOA Integration

Sebastian Wieczorek and Alin Stefanescu
SAP Research
Bleichstraße 8, 64283 Darmstadt
{sebastian.wieczorek, alin.stefanescu}@sap.com

Jürgen Großmann
Fraunhofer Institute for Open Communication Systems (FOKUS)
Kaiserin-Augusta-Allee 31, D-10589 Berlin
juergen.grossmann@fokus.fraunhofer.de

Abstract: Model-based testing (MBT) aims to improve the test automation by generating test cases from models. However, to become really successful in practice, MBT needs several prerequisites like an approach to make the generated test case executable, a platform allowing automatic test execution, rich behavioural test models and test data, and, not least, clear test objectives and test coverage criteria driving the test generation. In this paper we concentrate on the latter two mentioned aspects. More precisely, these aspects are analysed in the context of integration testing for service-oriented architectures (SOAs). The paper reflects our experience in the MBT process that spawns from modeling and test objective definition to the derivation of coverage criteria.

## 1. Introduction

The core competence of SAP, the world's leading provider of business software, is Enterprise Resource Planning (ERP) [8] software, which supports business processes for whole companies. ERP software integrates many organizational parts and functions into one logical software system, its complexity being given not only by the size but also by the huge data volume that it is managing.

SOAs promise a flexible and effective environment for integrating diverse functionalities provided as services. Independent services are thus composed to implement the specific business processes of a company. Responding to the ever rising complexity of ERP systems, SAP's recently announced product "Business ByDesign"[1] is utilizing the concepts of Enterprise SOA (eSOA) [15]. Process components, consisting of business objects, offer services that are accessible through the Enterprise Service Repository

[1] http://www.sap.com/solutions/sme/businessbydesign

(ESR). These services can be coupled to form complex business functionalities that previously were provided by large, monolithic ERP systems only.
The fundamentals of SOA, that is decomposition of systems into services, have been provided by already established concepts like the Common Object Request Broker Architecture (CORBA) and the Distributed Component Object Model (DCOM). Additionally, the application of web-service-like interaction in SOA adds the advantage of interconnection in an object-model-independent way and hence makes interaction between services much easier. On the other hand, the obtained composite application poses new challenges to the testing process given by its heterogeneity, distribution, dynamicity and size. A model-driven approach to SOA development addresses such problems by providing the right level of abstraction to reduce the complexity. As a natural extension, models could also play an important role in the testing process.

MBT approaches use abstract behavioral and structural system information (either from a system model or a separate test model) to generate a test suite (set of test cases). By deriving test suites from models, test generators aim to cover model features like the states or transitions of a state machine. While generic test generation algorithms can produce large to infinite numbers of test cases easily, resource restrictions on testing impose the generation of smaller test suites that maximize the fault detection by proper coverage criteria. These criteria are related to the structure of the models. As system models are usually not complete in terms of data modeling, common generators produce abstract test cases which have to be further refined by adding concrete data.

In this paper we share our experience of enabling MBT for SOA service integration. By reasoning about test objectives, test models and coverage criteria in that context it could be used as a practical guideline, pointing out necessary decisions in each area as well as their implications.

The paper has the following structure. Section 2 introduces the specifics of SOA integration testing. Section 3 describes different aspects to be covered by metamodels for SOA interaction. Section 4 describes what objectives could be considered for integration testing. Section 5 provides coverage criteria for finite state machines (FSM) and extended FSMs and discusses their fault detection capabilities.

## 2. SOA Integration Testing

### 2.1. Challenges of SOA Testing

The development of SOA applications imposes some unique challenges to software testing due to the complexity and new dimensions of the SOA implementations. While some traditional prerequisites and assumptions are still valid, others have to be re-

laxed or do not hold anymore. We briefly emphasize these new challenges and their impacts below.
*Unobservable global states*. The global state of a software application is not only determined by its actual process step or the values of its variables at runtime but also by the stored data that the system is able to access. This data affects the state because it might influence the next step of computation, too. For example insufficient funds in a client account may restrict the interaction between the banking software and an employee serving this customer. The ability to observe the state of the system at testing time directly influences the information whether the system responded correctly to a given test stimulus. Distribution of SOA components leads to a decomposition of data, too. Consequently the global state of a SOA application is hardly observable anymore.

*Dynamic system states.* Another important testing paradigm is the reproducibility of tests. As SOA applications are highly complex in functionality and data, such a requirement is difficult to meet. For example it is practically impossible to bring ERP systems back into a defined initial state because the effort is too high even in a developing phase. As each test execution changes the system state, a rerun of the same test case will most likely find the system in a different condition. Consequently a testing strategy for SOA has to address the challenge of a dynamic state during test execution.

*Challenging test data provision*. State changes heavily depend on input data (transactional data) and system internal data (master data and state variables). When providing test data for complex systems such as SOA applications, various data constraints have to be considered [14]. Moreover, test data with complex structure (containing dozens of required fields) originating from heterogeneous systems out of the control of testers should be properly handled.

*Complex interaction patterns*. The decomposition into components at SOA applications shifts the focus of testing. Not only functional correctness has to be guaranteed, but also interoperability and integration should be assured. Protocols are used to define interaction patterns and should be used to test the correctness of the communication. As of today various description languages are emerging, each with a different scope and granularity but unfortunately the support for automatic test generation has not been fully achieved so far in the context of SOA.

## 2.2. Integration Testing

Weyuker [13] identifies at least three stages of correctness testing that are absolutely necessary to ensure the reliability of component-based systems:

- *unit* or *component testing* that addresses individual components,

- *integration testing*, in which the subsystems formed by the integration of the individually, already tested components are tested as an assembly, and
- *system testing*, in which the complete system functionality is verified. This might include not only functional related tests but also robustness tests as well as performance tests.

In the SOA context, integration testing aims to verify the interaction of different components bundled in a composite application. Integration testing is essential given that the business processes are implemented at this level. Due to its complexity, SOA's assembly-based approach strongly favors the use of automation and test-driven development techniques. Until now, this desideratum was partially achieved at the level of testing single services or their conformance to protocols like SOAP [3,11,16] rather than testing the service integration using the global protocol of their choreography or orchestration [1,ch. 10]. In this paper we support the latter one by means of MBT.

Even though integration testing for SOA is not thoroughly studied, much of the mature concepts for protocol testing in telecommunication area can be adapted. Communication protocols are formal descriptions of the interactions that occur between a defined set of components in general and between a set of software components in particular. One of the main issues of protocol testing is to check whether an implementation conforms to a given standard. Standardized procedures for protocol testing and protocol testing processes have been developed by ISO and ETSI. Good summaries are given in [7] and [6].

In [6] different kinds of protocol tests are distinguished:

- Tests that are carried out during the development phase of a component by developers. They are similar to classic software unit tests.
- Conformance tests that aim to check whether an implementation conforms to a given protocol specification. In the case of standardized protocols the standard is used as a specification.
- In many cases the conformance to a specification is not sufficient to guarantee the interoperability of interacting components that emanate from different distributors and implementers. To fill this gap, interoperability tests address the interoperability between components and different implementation of the same protocol.

## 3. Test Models

At present there is no generally accepted way to describe the interaction of services for SOA applications, even though various standards are emerging: e.g. WS-CDL, UML diagrams, BPEL4Chor, iBPMN, Let’s Dance (see for instance [5] for further pointers). In order to support model-based integration tests, an appropriate choreography modeling language has to be chosen or developed. This decision has to be taken most carefully, because it might already restrict the later testing. We identified the following considerations as necessary:

- *Explicit send and receive.* It can be necessary to distinguish between sending and receiving a message when there are fine granular constraints associated (e.g. sending of a cancellation is permitted for a component as long as it does not receive a confirmation). Usually such constraints are enforced locally and hence out of scope for integration testing. However there might be an instance that is able to globally enforce them or observe their violation and hence justifying such detailed model.
- *Asynchronous communication.* For various reasons like performance, many SOA implementations rely on asynchronous communication. A modeling language that implicitly assumes synchronous communication (e.g. iBPMN) might therefore not always be applicable.
- *Activity or state based.* Many modeling languages (e.g. WS-CDL) are interaction based. If interactions heavily depend on the interaction history (captured by the state) they unfortunately lead to cluttered models.
- *Explicit parallelism.* If the communication is triggered and controlled by just one participant, parallelism usually does not occur. If multiple parties might be acting independently at the same time a means to model parallel interactions is an advantage, but usually complicates the later validation and verification.

In the rest of the paper, we will focus on extended finite state machines (EFSMs) to model SOA component interaction. They have an expressiveness that does not limit us in any of the above mentioned points. Finite state machines (FSM) allow us to explicitly model send and receive and the interaction states. Parallelism can be modeled by interleaving of independent transitions in every possible order. Additional to the transitions and states of FSMs, EFSMs permits the use of variables and guards. The obvious advantage of EFSMs is that they are able to express large and even infinite state spaces in a more compact way. An EFSM can model a system under test (SUT) more accurately while its visible states define an FSM that can drive the test generation.

In order to use choreography models for MBT, further information has to be annotated that supports the test generation. Such annotations could have different functions, e.g. to compensate a lack of model content available from the communicating components or to define additional data constraints for the testing process.

## 4. Test Objectives

Test objectives are describing the features of the software that have to be examined during the test. They are used to restrict the scope of a certain test campaign and to delimitate it from already existing ones. To define the test objectives for SOA integration testing, it is necessary to be clear about the fault categories that have to be tested. For example local correctness of participating components should not be evaluated as this can be done much more efficient in isolation.

As already mentioned integration, testing is applied during the integration of components to larger entities and aims to detect failures in the interoperability of and the communication between components. It mainly addresses the correctness of interfaces, messages, protocols and checks the functionality of the underlying communication infrastructure. Besides structural aspects (correctness of interfaces and message) it focuses on the aspects of component functionality that handles the interaction with other components. Moreover non-functional aspects like timing and robustness might be considered. For the integration of components different integration strategies are established [4,9].

In [4] the failures detectable by integration testing are classified. The following listing provides a summary for message based communication that additionally identifies possible failure causes:

- *wrong messages delivery* (including omitted messages) may be caused by functional errors on component or communication layer level, due to protocol misinterpretation or incompatible interfaces
- *wrong messages handling and interpretation* may be caused by functional errors on component level or due to misinterpretation of the message specifications
- *wrong timing of message delivery* (either too late or too fast) refers to timing or scheduling failures or resource problems (memory, load, performance).

For state based systems with a formal specification like FSM or EFSM we can provide a refined list that directly relates the model with fault classes. Our classification is based on [6]:

- *input fault*: a missing or incorrect input event (a valid input message is ignored)
- *output fault*: transition emits wrong message (the wrong thing happens as a result of a transition)
- *transfer fault*: transition leads to wrong state or is missing
- *extra state fault*: number of states is larger than specified, this implies transfer faults from and to the extra state
- *missing state fault*: number of states is smaller than specified, this also implies transfer faults
- *sneak path:* a message is accepted when it should not be
- *illegal message fault:* an unexpected message causes a failure
- *trap door:* the implementation accepts undefined messages.

## 5. Test Coverage

An entity or property of a model is covered, if every occurrence is exercised by a test case. After deciding on the modeling language and reasoning about the test objectives for SOA integration testing, it is possible to derive coverage criteria for the test model, so that all the test objectives are satisfied with a minimum of effort. Subsection 5.1 gives a small summary about the coverage criteria for FSMs and relates it to coverage criteria of EFSMs. Subsection 5.2 discusses the fault detection capabilities of the various coverage criteria.

### 5.1. Coverage Criteria for FSMs and EFSMs

For classical test approaches different kinds of coverage are distinguished. Liggesmeyer [9] describes the coverage of requirements, the coverage of control flow elements and the coverage of data flow and of decisions. Most of the approaches are related to white box testing but can be adopted for model based testing as well. In the following we characterize different coverage criteria that are directly related to the structural elements of an FSM. Hence we consider input and output messages, transitions and states. For further information on other criteria please refer to [6,12].

- *Message coverage* intends to cover all messages (input and output messages) of an FSM. It mainly checks the communication infrastructure and the communication interfaces and ensures that the messages are transmitted.
- *Piecewise coverage* intends to cover all states and all messages of an FSM. It mainly addresses the communication infrastructure and the communication interfaces. It ensures that all states are reachable and all messages are transferable but does not cover relevant parts of the behavior.
- *Transition coverage* is obtained when every transition is exercised at least once. It includes the test of all states and messages but does not explicitly cover whether a correct state has been reached after a transition has been executed.
- *All n-transition sequence coverage* is obtained when every n-tuple of subsequent transitions is exercised at least once. It reveals some state faults but incorrect states may not be discovered.
- *m-cycle coverage* is obtained when every existing cycle in the model is exercised zero to m times (i.e. for m=2, each cycle is exercised 0, 1 and two times in independent test cases). It reveals some faults related to cycle side effects.
- *Round trip path coverage* is obtained when every sequence of specified transitions beginning and ending in the same state is executed at least once and additionally all simple paths from the initial state to the final state are included. It reveals some incorrect or invalid states but cannot guarantee completeness.

For EFSMs, the application of additional coverage criteria becomes necessary to cover the additional elements of EFSMs. Binder [4] proposes an implicit decision coverage and boundary value coverage for EFSMs by deriving so called test transitions from guarded transitions. By deriving the test transitions, the EFSM is transformed to an FSM that is less expressive than the original EFSM but is testable under consideration of the coverage criteria in previous section. The approach described by Binder can be mapped to decision coverage approaches known from white box testing: simple decision coverage, condition/decision coverage, multiple condition coverage, minimal multiple condition coverage, and modified condition/decision coverage [12].

## 5.2. Fault Detection Capabilities

The different coverage criteria mentioned above have different capabilities to reveal faults. Table 1 provides a correlation between fault classes from Section 4 and the coverage criteria from Subsection 5.1. The table specifies whether a certain coverage criteria may (sometimes) reveal faults, always reveals faults or never reveals faults. The described fault classes in the first column relate to the implementation. For example a missing transition means that a transition was defined in the model but not implemented.

We also distinguish between coverage of the test model and the complementary model (last two columns in the table). A complementary FSM as defined in [2] is derived from a given FSM by adding an error state and faulty transitions in order to obtain a full specification.

## 6. Conclusion

In this paper we introduced special characteristics of SOA integration testing and related them to existing approaches in the field of protocol testing. We further discussed how model-based testing can be applied to SOA integration testing. We presented our experience in choosing an appropriate modeling language and our view on necessary test objectives. Finally we presented common coverage criteria for MBT and gave an overview about their fault detection capabilities. Further research will have to evaluate the benefits and resource consumption of different combinations of coverage criteria.

*Acknowledgements*: This work was partially supported by the project MODELPLEX [3].

Table 1.

| **Fault Classes** | **Test Model** | | | | **Complementary Model** | |
|---|---|---|---|---|---|---|
| | **State coverage** | **All transi-tions coverage** | ***n*-transition coverage** | ***m*-cycle coverage** | **All transitions coverage** | ***n*-transition coverage** |
| Guard faults (wrong definition of guard predicates) | sometimes | sometimes | sometimes | sometimes | sometimes | sometimes |
| Missing input | sometimes | sometimes | sometimes | sometimes | never | never |
| Incorrect input | sometimes | sometimes | sometimes | sometimes | sometimes | sometimes |
| Output faults (transition emits wrong message) | sometimes | always | always | sometimes | always | always |
| Transfer faults (transition leads to wrong state or is missing) | sometimes | sometimes | always with appropriate *n*[2] | sometimes | sometimes | always with appropriate *n*[2] |
| Extra state faults (number of states is larger than specified, implies transfer faults from and to the extra state) | never | never | never | never | always | always |
| Missing state faults (number of states is smaller than specified, also implies transfer faults, i.e. all transitions are deleted) | always | always | always | sometimes | always | always |
| Sneak path (extra transition) | never | never | never | never | always | always |
| Side effects in cycle | sometimes | sometimes | always with appropriate *n*[2] | always with appropriate *m*[3] | sometimes | sometimes |

[2] when *n* steps are necessary to generate distinguishing behavior

[3] when *m* steps are necessary to generate distinguishing behavior, where $n \geq m$

## 7. References

1. P. Baker, Z.R. Dai, J. Grabowski, Ø. Haugen, I. Schieferdecker, and C. Williams. *Model-driven testing – using the UML testing profile*. Springer, 2007.
2. F. Belli and A. Hollmann. Test generation and minimization with “basic” statecharts. In Proc. of SAC’08, ACM Press, 2008.
3. A. Bertolino, G. De Angelis, L. Frantzen, and A. Polini. Model-based generation of test-beds for Web Services. In Proc. of TESTCOM/FATES’08, LNCS. Springer, 2008.
4. R. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley, Object Technology Series, 1999.
5. G. Decker and M. von Riegen. Scenarios and techniques for choreography design. In Proc. of BIS’08, LNCS 4439, pp. 121-132, Springer, 2007.
6. H. König. Protocol Engineering Prinzip, Beschreibung und Entwicklung von Kommunikationsprotokollen,Teubner, 2004.
7. R. Lai. A survey of communication protocol testing. Journal of Systems and Software 62(1), pp. 21-46, 2002.
8. D. E. O'Leary: Enterprise resource planning systems - systems, life cycle, electronic commerce and risk. Cambridge University Press, 2000.
9. P. Liggesmeyer. Software-qualität, testen: analysieren und verifizieren von Software. Spektrum Akademischer Verlag, 2002.
10. MODELPLEX: MODELling solution for comPLEX software systems. 6th Framework Programme EU Project. Reference: IST 34081 (IP). Webpage: http://www.modelplex.org.
11. I. Schieferdecker and B. Stepien. Automated testing of XML/SOAP based web services. Proc. of KiVS’03, pp. 43-54, Springer, 2003.
12. M. Utting and B. Legeard. *Practical model-based testing, a tools approach*. Morgan Kaufmann Publishers, 2007.
13. E. Weyuker. Testing component-based software – a cautionary tale. IEEE Software 15(5), pp. 54-59, IEEE Computer Society, 1998.
14. S. Wieczorek, A. Stefanescu, and I. Schieferdecker. Test data provision for ERP systems. In Proc. of ICST’08, IEEE Computer Society. 2008.
15. D. Woods and T. Mattern. *Enterprise SOA. Designing IT for business innovation*. O’Reilly Media. 2006.
16. Web-Service Interoperability (WS-I) Organization. Testing tools working group. Documents and tools at: *http://www.ws-i.org/deliverables/workinggroup.aspx?wg=testingtools*.