Model-based Integration Testing of Enterprise Services

Sebastian Wieczorek,
Alin Stefanescu
SAP Research CEC Darmstadt,Germany
forename.surname@sap.com

Ina Schieferdecker
Fraunhofer Institute for Open Communication Systems (FOKUS),Germany
ina.schieferdecker@fokus.fraunhofer.de

Abstract

The success of service-oriented architectures (SOA) depends on faultless and seamless service integration. Formal modeling of global communication protocols between services enables a model-based integration testing (MBIT) approach. In this paper we present an MBIT approach based on SAP proprietary choreography models called Message Choreography Models (MCM). We explain how MBIT fits into the SAP testing methodology for SOA and give some insights into the experience we gained from the work.

1. Introduction

The goal of integration testing is to show that a combination of different software components interact correctly. Especially for applications whose components are loosely coupled, as it clearly is the case for SOA, tests of the communication and interaction are as vital as the functional correctness of the single communicating parts. To determine the success of integration testing, specific coverage criteria have to be applied. Local component test coverage criteria like code coverage are unfortunately not sufficient in determining whether two or more components are able to operate with each other under the agreed circumstances. Only the application of a global test concept can provide that.

Enterprise Resource Planning (ERP) software integrates various organizational parts and functions into one software system. Its heterogeneous and distributed nature poses unique challenges to software development and testing. Service-oriented systems are regarded as a next evolutionary step to cope with the ever growing complexity of ERP systems where monolithic approaches are not applicable anymore. SAP is a leading provider of ERP software. In SAP's approach to SOA, independent business components exhibit enterprise services. They can be composed individually to implement customized business processes (see Figure 1). As service integration takes place on a higher level of abstraction than component development, complex service interactions need to be defined in a structured way. Choreography languages describe such interactions from a global point of view.

This experience paper presents a model-based approach for integration testing of service choreographies. The foundations of the approach have been provided in previous papers: [6] describes the details of the domain specific modeling language MCM, [7] shows the test generation approach and translations of MCM into executable UML with Java annotations, [8] discusses the different coverage criteria and their fault detection for integration testing, and [9] presents challenges in the area of ERP test data provision. In this paper we focus on the integration of our approach into the SAP test strategy and the experiences we gained.

The paper is structured as follows. Section 2 introduces MCM, which is the basis of our model-based integration testing (MBIT) approach. Section 3 shortly describes the test generation and in Section 4 we explain how MBIT integrates with SAP's testing framework. Section 5 concludes with the lessons learned.

2. Message Choreography Models

Choreography models play an important role in SOA development and can provide a basis for ensuring quality at several levels. According to the W3C Web Service Glossary, "a choreography defines the sequence and conditions under which multiple cooperating independent agents exchange messages in order to perform a task to achieve a goal state". More precisely, a choreography model describes the interaction protocol from the perspective of a global observer between a set of loosely coupled components communicating over message channels.

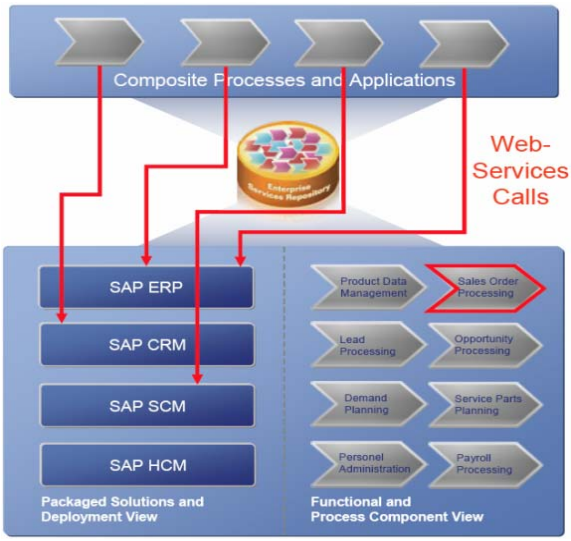

Figure 1. Service composition via an enterprise repository

In previous work [6], we defined precise requirements on choreography modeling languages that supports not only software design activities but also the testing process. State of the art choreography languages such as WS-CDL [4] or BPMN [2] cannot be directly utilized for model-based test design, mainly due to a high abstraction level, imprecise semantics, lack of a formal foundation, assumption of ideal channels, lack of termination symbols, etc. In [7] we therefore introduce a choreography modeling language called Message Choreography Modeling (MCM) that provides the missing elements mentioned before. Further we implemented an MCM editor with verification and testing plugins (see Figure 2).

MCM complements the structural information of the communicating components (e.g. service interface descriptions and message types) with information on the message exchange between them. MCM consists of different model types each defining different aspects of service composition:

Global Choreography Model. The global choreography model (GCM) specifies a high-level view of the conversation between service components. Its purpose is to define every allowed sequence of message receptions.

Local Partner Model. The local partner models (LPMs) specify the communication-relevant behavior for exactly one participating service component. Due to the design process of MCM, each LPM is a structural copy of the GCM with extra constraints on some of the local transitions, usually leading to the affected transitions being deactivated.

Channel Model. The channel model (CM) describes the characteristics of the communication channel on which messages are exchanged between the service components. It determines for example whether messages sent by one component preserve their order during transmission.

Model-based testing (MBT) approaches are able to effectively support automatic test generation from component interaction models as well as to execute and evaluate their success. Because MCM was designed with testing and formal verification in mind, it is highly suited for test automation techniques like MBIT. Nevertheless manual work cannot be eliminated totally making it necessary to restrict the size of the generated test suite to a minimum.

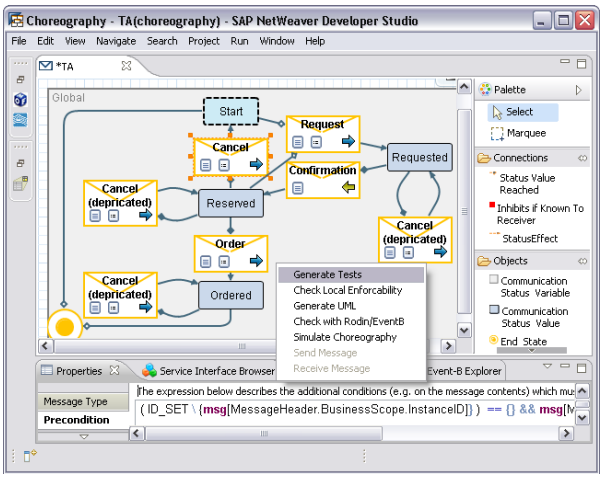

Figure 2. Screenshot of the MCM editor

3. Test Generation

Using MBT for service integration promises to reduce the manual effort by automatically generating minimal sets of test cases for a desired coverage of the choreography model. In [8] we discussed possible coverage criteria that can be used to drive service integration testing and how to choose them accordingly depending on effort and fault assumptions.

As described in [7] we decided to follow a three-step approach for test generation, which takes into account the MCM characteristics:

Step 1: A test generator is used to generate a set of globally observable message sequences according to a given model coverage of the GCM (e.g., state coverage, transition coverage, all-transition pairs coverage). FSM-based approaches can be used if none of the annotated constraints relates the current interaction to previous behavior. Otherwise (e.g. when a transition is dependent on a global counter variable) approaches like constraint solving, model checking,

theorem proving, or symbolic execution have to be applied. We have already started evaluating such techniques, but we will not report here on the results.

Step 2: The local event sequences corresponding to the test cases are computed. This is necessary because the GCM specifies the order of receive events only (receive semantics is suitable to uncover message racing). Therefore the receive sequences have to be enhanced by their corresponding send events taking the LPMs and channel model into account. Fortunately this can be done automatically without major issues. Further test oracles can be automatically inserted into test cases using information about how LPM states are related to the concrete component states as this information is annotated to the LPM states.

Step 3: The generated abstract test cases are translated into executable test suites. This step is semi-automatic. We can automatically generate the concrete test steps of each test case as well as state checks on the local components. However, as the actions that invoke message sending are not fully modeled in MCM, this information as well as the test data has to be added manually to the test cases.

Among different available coverage criteria we investigated transition coverage of the GCM. Our motivation was that transition coverage promises to uncover a significant amount of integration faults with relatively small effort. For example in the MBIT approach of [1], transition coverage of a global communication model is able to detect about 90% of integration related faults. In order to detect 95% of these faults combined transition coverage of the local behavior of the involved components has to be applied. In this case the test suite is increased 25 times. 100% fault detection is realized by a test suite that covers all local path combinations but test effort is again doubled compared to the combined transition coverage. It has to be said, that in the case of more complex behavior models i.e. incorporating loops or more than three states per component, the increase in effort for each coverage criteria would have been much higher.

When applying MCM-driven integration testing only the results for the global transition coverage are applicable. LPMs are abstracting from the local behavior by leaving out communication irrelevant transitions thus eliminating the possibility of checking that each local state is implemented compliant to the abstract communication state it is associated with. In fact most of the time transition coverage of the GCM also results in transition coverage of the involved LPMs. The better fault detection capability of pair-transition coverage in integration testing is due to better coverage of local states. The behavior of enterprise service components, incorporating various cycles and hundreds of states makes it difficult to test each possible local state combination in a service composition anyway. However if unit component tests are already checking the behavioral conformance of each local state to the associated abstract state in the LPM, then pair-transition coverage does not lead to better results than transition coverage.

4. Embedding MBIT into SAP's current test framework

According to the nomenclature of [5], chapter 8, we use a mixed approach for the test concretization phase, described in Step 3 of the previous section, which is a combination of the test adaptation and test transformation modes.

More precisely, we implemented a transformation from the abstract test cases to an internal SAP test language for integration testing. This language follows the keyword-driven testing principles[1] (see [5], chapter 2), i.e., it builds upon SAP's eCATT test script language [3] and was designed to address the requirements of integration testing at a higher level of abstraction. This test language contains constructs that can create and modify local business objects, can trigger the sending of messages between the business components via the available enterprise services and can check the values of the internal local states against the expected values in order to decide the failure or success of a test.

Test data used for the test runs on the system under test (SUT) is usually very complex and has to be compliant with existing master data and the actual system configurations. Automatic test data generation would demand test engineers to specify rather difficult test data models in a different modeling environment. Therefore we currently leverage the experience of the testers by manually providing test data.

To minimize the manual effort for the test concretization, we transform generated abstract test cases in a modular way. Each test step (i.e. local state checks and message triggering) is transformed in a separate script while for each test case a master script is generated that calls the test step scripts in the appropriate order. In this way we enforce a high reuse

[1] *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 [5]. Keyword-driven test cases are useful because they can fill the gap between the abstract test cases generated by the MBT engine and the the executable lower level test scripts.

that results in less effort and enables parallelization of the manual work, which is a big advantage for integration testing with different development areas concerned.

The test execution environment is provided by the SAP Test Workbench and SAP Solution Manager. These frameworks support the whole testing process starting from the test planning, test execution until the final test reporting. The main steps of the supported test process are shortly described.

First, the system integrators decide about test goals for the involved communicating business components and consequently test coordinators are drafting a test plan that satisfies these goals. Afterwards test case descriptions are generated, grouped into test suites, and filled into the test plan. At this point it is also decided which of the tests are carried out manually and which are automated. The automated tests are usually implemented in eCATT [3]. When model-based testing techniques like MBIT are used, deciding on test goals in terms of model coverage is the only manual work that has to be carried out up to this stage (as described in Section 3).

Afterwards, the necessary test data is made available. This is not a trivial task especially for distributed and heterogeneous systems, where the test metadata and database contents must be consistent. SAP provides tools like the Test Data Migration Server (TDMS) that is able to derive consistent reference data from existing systems. It is also quite common that reference test data is provided by customers or internal departments as additional information to the requirement specification. All these data samples are available so that the testers are able to choose the appropriate input for each test case.

The test execution is controlled by the Test Workbench, where test plans are executed automatically and periodically in case of regression tests. The results of the test runs are centrally reported including different coverage criteria based on source code, model elements, or requirements.

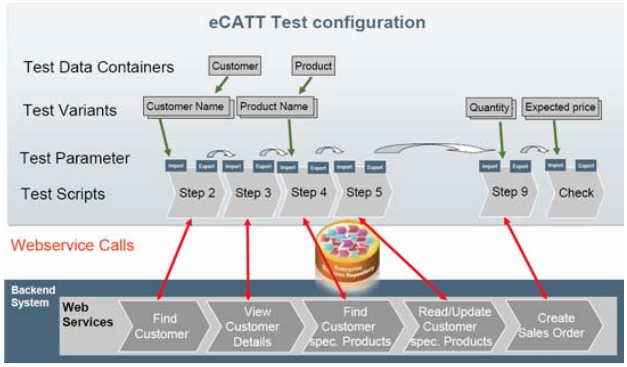

Figure 3. Service integration testing implemented using SAP's eCATT framework

Figure 3 shows how eCATT automates the integration test execution, by having different test scripts calling the involved enterprise services. The results of one script are transferred to the next script using exporting and importing functions. MBIT supports this approach by automatically generating the sequences of scripts together with their interfaces.

5. Lessons Learned

The pain points usually identified by ERP customers during integration test automation are: (a) creation of automated test cases is time consuming and expensive to maintain via skilled test specialists, (b) automated test cases get damaged by change events and need time consuming repair by test specialists, and (c) creation of appropriate test data for automated test cases is difficult. MBIT can certainly help with the points (a) and (b) with an upfront investment in modeling of service composition. In theory, the benefits of MBIT are: faster test design, higher test quality, better test coverage, easier test maintenance, and more test re-use. To realize these advantages in practice, several difficult problems have to be solved in each of the main MBIT steps. Our experience when introducing MBIT in an industrial setting is synthesized below.

Modeling of the SUT: Design decisions like the abstraction level of the model or the used patterns determine the quality of the test suite significantly. Modelers therefore must have an understanding of the test generation process in order to anticipate the consequences of their design decisions. A (possibly automatic) decision support, e.g. based on anti-patterns might be helpful.

Abstract test generation: Determining the right model coverage for the test generation should be based on various aspects like the error assumptions, model granularity and resulting test effort. Again, training of testers and decision support has to be provided. We further found that generating test suites consisting of test cases with a relatively small number of test steps have been preferred by testers over test suites where the overall number of test cases is lower. Shorter tests cases have the advantage that they are much easier to maintain and to debug in case of an error.

Test concretization: As mentioned in [5], this step can take up to half of the whole model-based testing effort. We note ourselves that this is an important step in the whole MBIT chain, which has been addressed little in the literature. Especially the test data provision is lacking tools that are able to cope with complex data types, even though automatic data picking from existing sources seems to be a promising approach.

Test Execution: An advanced test execution and test management framework, preferably including keyword-driven technologies, must be available. Fortunately this is the case at SAP. Moreover, due to complex test data involved, the test cases are not executed on the fly but after the test generation, i.e. offline MBT (see also [5]).

Test analysis: To assign verdicts to generated tests is a hard and error prone task. To tackle this, we linked the global states of MCM to their corresponding local states of the components such that we can generate test oracles automatically.

User acceptance and dissemination: We tried hard to come up with a tool that satisfies the users in functionality and ease of use. Therefore we defined a domain specific language MCM based on internal requirements and sharing some of the meta-model elements with existing SAP proprietary models, such that we can leverage the previous experience of the users. In our tool we incorporated information from other development models e.g. local components and offer it in different forms (e.g. auto-completion) to ease the navigation through the message data types and local business objects. We kept the users in the loop during our tool development, constantly giving internal demos for quick feedback. Currently we are piloting our approach with a group of 20 integration experts, developers, and testers from different areas for tool fine tuning before the release it on a larger scale. Most of the pilot users were comfortable with the graphical modeling and its underlying semantics, but sometimes had problems with first-order logic textual guards on the transitions. In the presence of an MCM expert, the users were able to draft a first MCM for their choreography within 2 hours, which can be used to automatically generate test suites containing 7 to 10 test cases, each having 4 to 8 steps. Given the fact that a complex application could sometime have up to a hundred peer-to-peer choreographies, one can envisage the saving potential of the MBIT.

Future work: In our current setting, service dynamicity is not addressed because the set of communication services is not changed at runtime using e.g. service discovery. Current enterprise software, once set up, is seldom reconfigured in respect to ad-hoc component integration at runtime and hence dynamic binding is not common in business critical processes. However it would be interesting to see if our approach can be adapted to the case of dynamic enterprise service compositions. Moreover, we will concentrate our future work on ways to automate the test data provision.

Acknowledgments. This work was partially supported by the EC-funded projects Modelplex[2] and Deploy[3] (grants no. 034081 and 214158).

6. References

[1] S. Ali, L. Briand, M. Jaffar-Ur Rehman, H. Asghar, M. Z. Iqbal, and A. Nadeem, "A State-Based Approach to Integration Testing Based on UML Models", *Information & Software Technology*, 49 (11–12), Elsevier, 2007, pp. 1087–1106.

[2] Business Process Modeling Notation (BPMN) Specification, Final Adopted Specification, *Technical report*, Object Management Group (OMG), Online at: http://www.bpmn.org

[3] M. Helfen, M. Lauer, and H. M. Trautwein, *Testing SAP solutions*, SAP Press, 2007.

[4] N. Kavantzas, D. Burdett, G. Ritzinger, and Y. Lafon, "Web Services Choreography Description Language Version 1.0. W3C Candidate Recommendation", *Technical report*, 2005.

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

[6] S. Wieczorek, A. Roth, A. Stefanescu, and A. Charfi, "Precise Steps for Choreography Modeling for SOA Validation and Verification", *Proc. of Int. Symposium on Service-Oriented Software Engineering (SOSE'08)*, IEEE Computer Society, 2008.

[7] A. Stefanescu, S. Wieczorek, and A. Kirshin, "MBT4Chor: A model-based testing approach for service choreographies", *Proc. of European Conf. on Model-Driven Architecture (ECMDA'09)*, LNCS 5562, Springer, 2009, pp. 313–324.

[8] S. Wieczorek, A. Stefanescu, and J. Großmann, "Enabling model-based testing for SOA integration testing", *Proc. of Workshop on Model-based testing in practice (MOTIP'08)*, Fraunhofer IRB Verlag, 2008, pp. 73–82.

[9] S. Wieczorek, A. Stefanescu, and I. Schieferdecker, "Test Data Provision for ERP Systems", *Proc. of Int. Conf. on Software Testing, Verification and Validation (ICST'08)*, IEEE Computer Society, 2008, pp. 396–403.

[2] http://www.modelplex-ist.org
[3] http://www.deploy-project.eu