MBT4Chor: A model-based testing approach for service choreographies

Alin Stefanescu[1], Sebastian Wieczorek[1], Andrei Kirshin[2]

[1] SAP Research, CEC Darmstadt, Bleichstr. 8, 64283 Darmstadt, Germany
{alin.stefanescu,sebastian.wieczorek}@sap.com

[2] IBM Haifa Research Lab, Haifa University, Mount Carmel, 31905 Haifa, Israel
kirshin@il.ibm.com

Abstract. Service choreographies describe the global communication protocols between services and testing these choreographies is an important task in the context of service-oriented architectures (SOA). Formal modeling of service choreographies makes a model-based testing (MBT) approach feasible. In this paper we present an MBT approach for SOA integration testing based on SAP proprietary choreography models called Message Choreography Models (MCM). In our approach, MCMs are translated into executable UML models using Java as action language. These UML models are used by a UML model execution engine developed by IBM for test generation and model debugging. We describe the achievements and challenges of our approach based on first experimental evaluation conducted at SAP.

Keywords: Choreography Modeling, SOA, Service Integration, Model-Based Testing, Model Transformation, Domain Specific Language

1 Introduction

Enterprise Resource Planning (ERP) software [15] supports business processes for whole companies, with SAP being a leading provider of ERP software. ERP software integrates many organizational parts and functions into one logical software system, posing unique challenges to software development and also testing [12,21]. Recently, service-oriented architecture (SOA) has come to be regarded as the next evolutionary step in coping with the software complexity of ERP systems where monolithic approaches are no longer applicable [23,16]. SAP simplifies the SOA adoption by delivering SOA-enabled software, SOA methodology guidelines, and professional services. Using the SAP approach, independent business components exhibit enterprise services that can be composed individually to implement customized business processes. For service integration to occur on a higher level of abstraction than component development, complex service interactions need to be modeled: hence, choreography languages have emerged. According to the W3C Web Service Glossary [17], "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". Thus, 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.

Choreography models play an important role in SOA development and can provide a basis for ensuring quality at several levels. In previous work [19], we defined precise requirements on choreography modeling languages that support the three software quality related use cases of design, verification, and testing. However, we observed that state-of-the-art choreography languages such as WS-CDL [13], Let's Dance [24], and BPMN [3] do not fulfill all these requirements simultaneously, mainly due to the high abstraction level, imprecise semantics, lack of a formal foundation, assumption of ideal channels, lack of termination symbols, etc. Recently SAP Research developed a proprietary choreography modeling language called Message Choreography Modeling (MCM) that satisfies the requirements from [19] (e.g., graphical state-based representation, explicit concurrency, detailed message types, local viewpoints, determinism, a distinction between global and local constraints) and implemented an MCM editor with verification and testing plugins.

In this paper, we describe a model-based testing (MBT) approach called MBT4Chor for service integration based on MCM. The goal is to generate integration tests for message-based communication between business components. We achieve this with two complementary means: (a) by directly implementing graph-coverage algorithms for the choreography models and (b) by a model transformation to executable UML where we can use an IBM research prototype tool for generating random tests.

The contributions of the paper are the following:

1. to present our experience of using MBT on a domain-specific language (DSL),
2. to sketch a transformation from our DSL (MCM) to a general purpose one (UML),
3. to describe a new UML based test generation tool that relies on the UML execution engine presented in [5], and
4. to share the lessons learned and challenges of an MBT approach in an industrial setting of SOA applications.

The paper is structured as follows. Section 2 provides an overview of MCM. Section 3 describes the proposed MBT approach and Sections 4, 5, and 6 present the translation from MCM to UML, the test generation tool for UML and the test concretization at SAP. Sections 7 and 8 provide related work and future activities.

2 Choreography Modeling

The SAP approach to SOA and SAP's internal software development lifecycle includes a rich modeling environment. While external SOA artifacts are based on web service open standards like SOAP, WSDL, and WS-Reliability, internal development is based on various proprietary models for business objects, service components, component interaction and integration scenarios. Recently, SAP Research developed a proprietary domain-specific language for modeling service choreographies called Message Choreography Modeling (MCM) together with a customized Eclipse-based modeling environment.

MCM complements the structural information of the communicating components (e.g., service interface descriptions and message types) with information on the message exchange among them. MCM consist of different model types each defining

different aspects of service composition. The remainder of this section informally describes these model types and their relations are informally described.

2.1 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 exchanges. Similar to an extended finite state machine (EFSM), a GCM consists of transitions with guards and side effects over different datatypes, leading to a possibly infinite state space. In the GCM, transitions are fired only when a global observer is able to detect that a message was consumed by the receiving component (i.e., the message left the communication channel). In contrast to common choreography languages that have their semantics based on send events, we noticed that the receive semantics of GCM provides a better testing approach because message racing can also be captured.

The left side of Figure 1 shows an example of a GCM for two components, a seller and a buyer. The buyer is able to send the messages *Request* and *Cancel* that are then received by the seller, while the seller can send the message *Response*, which is received by the buyer. For simplicity, in this example no guards, side effects or concurrent states are used, although such features are present in the MCM metamodel.

One of the requirements for GCMs is determinism. In the example in Figure 1 this is achieved already because in each state the outgoing transitions have different messages. In scenarios where this is not the case, determinism has to be enforced by assigning mutually exclusive guards.

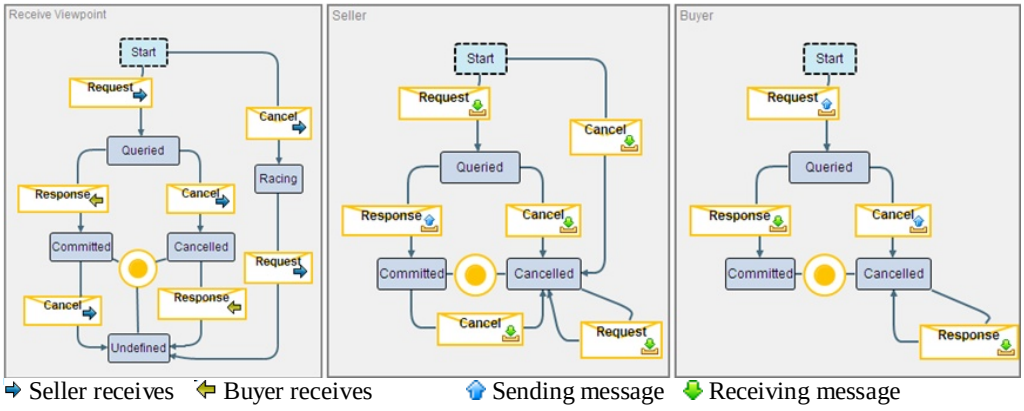

Figure 1. Global Choreography Model (GCM) on the left and the two corresponding Local Partner Models (LPMs)

Another requirement of GCMs is the marking of initial and target states in which the communication reaches an agreed goal of conversation. The test generators must generate test sequences starting in an initial state and ending in a target state. Note that in MCM, the target states allow outgoing transition because of business scenarios where one of the communicating partners is allowed to restart a negotiation process

infinitely often. This is in contrast to the final states of the UML state machines. In the GCM of Figure 1 the states *Committed*, *Undefined*, and *Cancelled* are target states, while the state *Start* is marked as an initial state.

2.2 Local Partner Model

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 removed. For exactly two involved partners, the global message receive events are copied to the receiving components, while the receive events of other partners are transformed into send events.

Figure 1 shows one possible set of LPMs for the GCM. Compliant to the global specification GCM, the LPM of the buyer consists of send events for the *Request* and *Cancel* messages and receive events for the *Response* message (and vice versa for the seller).

As mentioned, some of the events copied from the GCM have been removed in each of the LPMs. Note that despite these local reductions, in our example each global transition can still be reached, that is, all message sequences possible in the GCM can be simulated with local sequences of the sends and receives in the LPMs. These send and receive events take into account the communication channel properties (see below).

2.3 Channel Model

The channel model 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. For the channel definition, MCM uses the Web Service Reliable Messaging (WS-RM) standard [18]. In the example in Figure 1, we assume a reliable channel on which each sent message is received exactly once, but which does not necessarily preserve the message order. Therefore, the buyer may send the *Cancel* message only after sending *Request*, while the seller has to be prepared to receive either *Request* or *Cancel* first. Because the GCM describes the order of receive events, it also reflects the possible switching of the *Request* and *Cancel* messages on the channel.

3 Overview of MBT4Chor

In addition to a holistic software design purpose, the development of MCM was driven by the requirements of automatic model verification and test generation. This section gives an overview about the utilization of MCM for testing.

The core idea of model-based testing (MBT) is to use formal specifications for test generation. This implies that tests can only be as precise as the modeled content they use. By design, MCM offers the necessary information to drive the generation of test suites covering the specified interaction protocol. The generated test suites have to be

supplemented with additional information, because even though the local behavior is modeled in the LPMs, triggers for the local message sending events are not specified. This information cannot be easily modeled as it is deeply rooted in the internal behavior of the components. Note that MCM is not suited for the derivation of component tests because apart from the missing triggers mentioned, the behavior modeled in the LPMs focuses on communication only, leaving out internal steps that may happen in between the communication events.

However, using MBT for service integration promises to reduce the manual effort by automatically generating minimal sets of test cases for desired coverage of the choreography model. In [20] we discussed different coverage criteria that can be used to drive service integration testing and how to choose them accordingly, depending on effort and fault assumptions. Among the different possible coverage criteria, we decided to start with the transition coverage of the GCM, which requires the following MCM-specific three-step approach for test generation:

– *Step 1*: A test generator generates a set of globally observable message sequences that cover each transition of the GCM. According to our example from Figure 1, a generated sequence could be <*Seller:Request, Buyer:Response, Seller:Cancel*>. Given the receive semantics of GCM, this reads:

  <*Seller receives Request, Buyer receives Response, Seller receives Cancel*>.

– *Step 2*: The local event sequences corresponding to the test cases are computed. This is necessary because the GCM only specifies the order of receive events. Therefore the receive sequences have to be enhanced by their corresponding ***send*** events, taking the LPMs and channel model into account. According to Figure 1, the only possibility of achieving the generated sequence in Step 1 is this:

  <***Buyer sends Request***, *Seller receives Request*, ***Seller sends Response***, ***Buyer sends Cancel***, *Buyer receives Response, Seller receives Cancel* >.

  This step is MCM-specific and relies on the receive semantics of the choreography model. It can be done automatically without major issues.

– *Step 3*: Generated abstract test cases are translated into executable test suites. This step is semi-automatic. We can automatically generate the concrete test steps as well as the state checks on the local components. However, as triggers are not fully modeled in MCM, this information has to be added manually to the test sequences. The test concretization is further described in Section 6.

Step 1 is concerned with the integration of a test generator. The fact that the MCM editor is based on the Eclipse plugin technology opens up the possibility of integrating multiple test generators. For the moment we have experimented with the following two test generators:

– An SAP in-house solution for test generation – We implemented classical graph-coverage algorithms working directly on the global state space of the GCM. The major disadvantage is that it takes time and effort to reach the state of the art in MBT, so we also used an external tool (below) for that.

– An MBT prototype from IBM Research – The usage of this tool, described in Section 5, is enabled by the transformation from MCM to UML explained in Section 4. However the current version of the tool has a complementary coverage

criteria based on input coverage of the operation calls and their parameters. The operation calls are used to trigger the transitions in the UML state machine corresponding to the choreography.

Figure 2 depicts the tool architecture based on the MODELPLEX platform. In the middle it shows the MCM module including an MCM editor and a model importer from SAP's existing models. The toolset on the top right of the figure shows the UML test generator and debugger, which are part of the Simulation, Verification and Testing Workbench developed by the MODELPLEX project [14]. This workbench also includes tools for performance simulation and model verification that are not presented here. The connection between MCM and UML is made via the MCM2UML and the TPTP2SAP transformer modules. The SAP in-house MBT solution is not depicted.

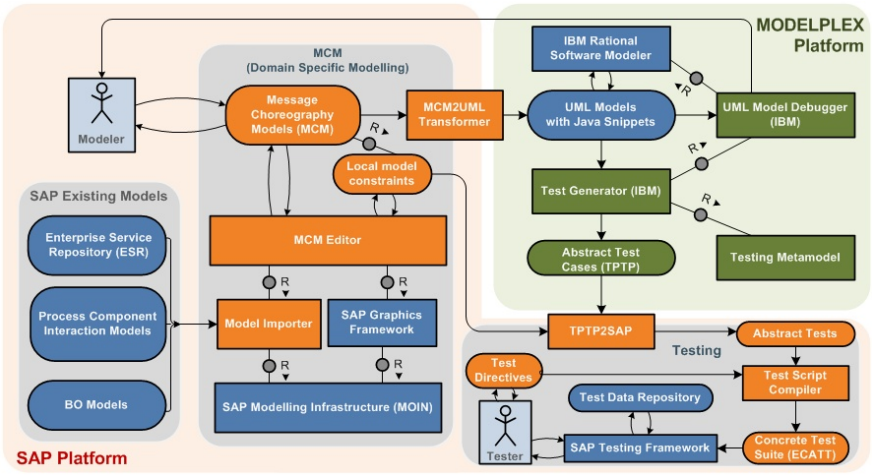

Figure 2. Tool architecture of the MBT4Chor approach

4 Translation from MCM to UML with Java annotations

This section describes how the MCM models are transformed into the annotated UML model that serves as the test model. These UML models include the UML classes and composite structures as structural constructs, and the state machines as behavioral constructs. The transformation was programmed in Java using the APIs provided by the MCM tooling and the APIs provided by the EMF-implementation *UML2*[1] for the Eclipse platform of the UML 2 OMG metamodel.

The mapping used in the translation of MCM to UML is sketched below:

a. For each message type *mt* of the choreography, we generate an UML Signal *signal_mt* (denoted by a classifier symbol with the keyword <*signal*>). The data

[1] *http://www.eclipse.org/modeling/mdt/?project=uml2*

type of the signal is similar to the data type of the message. To keep the test model simple, we ignore the structural information that is not relevant for testing. Additionally, we create a UML SignalEvent associated with each Signal. The signal events trigger the transitions of the UML state machine for the choreography.

b. For each partner p in the choreography, we generate a UML Class class_p. Then:
   - For each service *s_p* provided by a partner *p*, we generate a method *method_s* of the class *class_p*, with the same parameters as the service. The parameters are calculated from the WSDL description of the service. The data types of the parameters are, of course, compliant with the messages exchanged via the service.
   - The action associated with *method_s* is given as an opaque behavior with Java as the language. The code implements the sending of the signal *signal_s_p* associated with the service *s_p* using the signal sending API of the Model Execution engine (*MexSystem)*. This is the corresponding code:

     ```
     Signal_S_P signal = new Signal_S_P();
     signal.parameter=parameter;
     MexSystem.send(choreography_instance,signal);
     ```

c. For the choreography protocol, we create a UML Class class_chor and create associations from each of the partners to this choreography class (such that they can reference a choreography instance and send signals to it).

d. We create the initial configuration of the system as a UML composite structure with a choreography instance and an instance for each of the choreography partners. The instances of the partners are connected by UML connectors to the choreography instance.

e. The core of the transformation is given by the translation of the MCM choreography protocol into a UML state machine, which is associated as behavior to the *class_chor* defined above.
   - The concurrent states of MCM are translated into concurrent regions of the state machine.
   - MCM activation of message interaction can involve *OR* and *AND* operations over the MCM local parallel states. They are simulated with junction and join pseudo-states in the UML state machine. Moreover, the effect of an MCM message on the local parallel states is simulated using fork pseudo-states. The initial and end states are mapped to UML initial and final pseudo-states.
   - The MCM guards on the messages are translated into Java guards. Note that complex decision procedures need to be encoded in Java functions. For example, the existential and universal first-order quantifiers, *for all* (∀) and *there exists* (∃), we need Java helper methods to be able to implement constraints such as this:

     ```
     forall x: msg [SalesOrder. Item] (x [ProcessingStatusCode] ==
     CONFIRMED);
     ```

     meaning that all items of the sales order transmitted in the message are confirmed.

– The translation of MCM action code and MCM global variables to Java is straightforward and we do not explain it in detail.

Note that the above translation takes into account the special semantics of the executable UML models supported by the IBM tool described in Section 5.

5 UML Test Generator and Model Debugger

This section describes tools that were developed at the IBM Haifa Research Lab and used in our experiment. They are extensions of Rational Software Architect (RSA), which is used to import the MCM descriptions that have been transformed to UML. As explained in Section 4, the structure of the system is described using class diagrams and composite structure diagrams. State machines, activities diagrams, and Java code snippets are used to describe the behavior. In our scenario, the Model Execution Engine [5] executes the model "behind the scenes", while the "main actors" are the model debugger and test generator (see Figure 3, left).

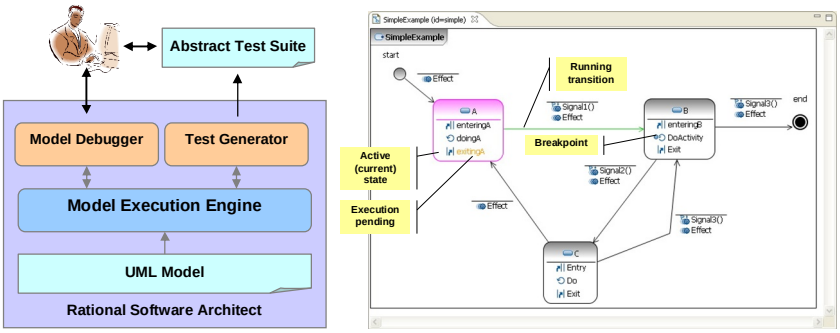

Figure 3. The architecture of the IBM prototype (left) and a debugging session of a UML model (right)

The model debugger verifies that the model describes the correct expected behavior of the System Under Test (SUT). It enables the user to interact with the executable model in two ways:

– *To control the execution* by sending inputs to the model: that is, create instances, invoke operations on instances, and send signals to instances.
– *To observe the execution* by observing the outputs of the model: that is, the attribute values, active states, and signals to the environment.

The model debugger helps answer the question "*Is there a defect in the model?*" at two different stages: before test generation during the modeling of the SUT, and after test generation, when a test case fails. In the latter case, the defect can be either in the SUT or in the model. If the defect is in the model (its behavior is wrong), the debugger localizes and fixes the defect, thus answering the question, "*Where is the defect?*". The debugger allows the setting of breakpoints on model elements. Figure 3 illustrates a running state machine with a highlighted active state and running transition. It also shows breakpoints and execution pending elements.

The test generator also uses the model execution engine and acts similarly to the model debugger by sending inputs and observing outputs. The inputs are recorded as test sequences of stimuli applied to the SUT. The outputs are also recorded as expected outcomes. In other words, the model is used as a test oracle predicting the correct behavior. During test generation the next input for a test sequence is selected by the test generator depending on the coverage task chosen by the user:

- *Random* generates random sequences of stimuli (no coverage)
- *Input Coverage* covers as many different inputs as it can (a specific input)
- *Input Step Coverage* covers as many different inputs in each step in the test case as it can (a specific input in a specific step in the test case)
- *Input Step Pair Coverage* covers as many different pairs of inputs in each pair of steps in the test case as it can (a specific pair of input values in a specific pair of steps).

The current version of the test generator only implements *input coverage algorithms*. Their advantage is that such coverages do not face the problem of state explosion. This problem occurs when test generators explore the whole state space of the test model, which can grow very quickly for large test data (as is the case with MCM).

Another advantage of the test generator is that it actually executes the model of the SUT. This enables us not only to generate the sequences of stimuli but also to precisely predict the expected behavior (outputs of the SUT). Test generators that statically analyze the model often have problems to predict these expected results.

Another strength of the test generator is that it uses the same model execution engine as the model debugger. In case of wrongly generated tests resulting from an incorrect test model, the problem can be easily located and fixed in the model using the model debugger. Test generators using their own execution engine might interpret the semantics of the model differently, which complicates the maintenance of the original model.

For the test suite format, we use the Eclipse Test & Performance Tools Platform[2] (TPTP). Test steps (stimuli and observations) reference model elements and a special editor was developed at the IBM Haifa Research Lab for convenient viewing and editing the generated tests. In addition to using the described test generator, tests can also be created manually using the editor. Finally, the test can be either translated to a specific test script for execution on the SUT or alternatively, TPTP can be extended to execute the test directly on the SUT.

6 Test Case Execution in SAP Backend

Once the abstract test cases (in TPTP format) are generated, they must be transformed into executable test scripts. As mentioned in [17], this task is very important and as observed in practice it can take up to half of the time spent on the whole model-based testing approach.

For our approach we have implemented a transformation from the abstract test cases to an internal SAP test language for integration testing. This language follows

[2] *http://www.eclipse.org/tptp*

the keyword-driven testing principles (see [17, Chapter 2]), i.e., it has a higher level of abstraction than the SAP's eCATT script language usually used for testing SAP applications [12]. According to nomenclature of [17, Chapter 8], we use a *mixed approach* for our test concretization, which is a combination of the test adaptation and test transformation modes. To increase the usability and help the testers to visualize the generated test case, we also implemented a transformation of the generated message exchange sequence into UML sequence diagrams.

Since the test data used for the real tests that can be executed on the SUT is very complex, it that it relies on existing master data and different system configurations, we currently do not generate it automatically. Instead we leverage the experience of the testers to reduce the effort and risk of rather difficult test data modeling in a new environment that could be error-prone.

7 Related Work

MBT is an active research area, but it is still not adopted by mainstream industry. In particular, we are not aware of any MBT tools directly running on models using classical choreography languages such as WS-CDL, BPMN, or Let's Dance. Existing MBT approaches for web service (WS) testing concentrate either on testing a single WS by adding state machines to the WSDL interface descriptions [7] or testing WS orchestrations based on BPEL [8]. No approach addresses the utilization of global choreography models for WS interaction testing. In a choreography setting similar to ours, we found only one tool called WS-Engineer [6] which transforms WS-CDL as choreography models and BPEL4WS as local models into labeled transition systems (LTSs). These LTSs are used for model verification, but not for test generation.

The fact that we generate UML models on the basis of MCM opens up the possibility of making use of existing MBT approaches and tools for UML. For instance, approaches for component integration testing based on UML are described in [1,11]. Examples of commercial tools for MBT based on UML are: ATG tool[3] from Rhapsody based on UML, Test Designer from Smart Testing[4] for UML with OCL as action code, and QTronic[5] from Conformiq for UML with Java annotations. However, using any of these tools or approaches should be decided after a careful analysis of capabilities, input test modeling and output test format, and semantics of the used executable UML.

8 Conclusions

This paper presents an end-to-end process for model-based testing of service choreographies, starting with the modeling using the domain specific language MCM, its transformation to UML, the test generation and finally test execution in the backend. Although we managed to automate a large part of this complex process,

[3] *http://modeling.telelogic.com/products/rhapsody/test/automated-test-generation.cfm*
[4] *http://www.smartesting.com*
[5] *http://www.conformiq.com/qtronic.php*

there is still room for improvement. This section describes some of the experiences we gained.

Lessons learned and challenges encountered: Given the proprietary modeling stack at SAP, we had to design a DSL called MCM for modeling choreography with the purpose of test generation rather than directly using UML (see also [9]). The designed language has a precise semantics and incorporates existing SAP metamodels and feedback from SAP architects and testers to foster internal adoption. Our approach relies on the mature test execution environment of SAP that provides keyword-driven testing tools. The disadvantage of a DSL is that mature MBT tools cannot be directly applied, but model transformations to general purpose languages are needed. We learned from the experiences of the AGEDIS project [10] and chose UML with Java annotations as the target language. The Java language has an imperative semantics (as opposed to OCL, for instance) and is expressive enough to capture our complex guards based on first-order logic. The model execution engine [5] used by the test generator described in Section 5 is able to execute UML with Java annotations. While the test generation and test execution can be automated, we are currently not able to fully automate the test data generation. This is due to the complex master data and test data constraints in an ERP system [21] that cannot be easily be modeled. Although some of the constraints on the test data can be captured in the MCM guards, it is still possible that we generate infeasible paths for which we cannot provide proper test data. These must be filtered out manually by the testers and the feedback incorporated into the test generator that must provide alternative paths. We will look at ways to improve these inconveniencies based on the test pilots currently running at SAP.

Future Work: Our plans are driven by the above mentioned challenges in the test data provision area. For that, we are currently working towards incorporating a model-checker and constraint-based solver that could help to reduce the number of infeasible paths due to data inconsistencies. This is, however, a difficult problem (even undecidable) in general. We also plan to better support and validate the semantical relation between the local and global views of MCM and the traceability link between MCM and the generated UML. Moreover, we will evaluate test effectiveness and bug detection capabilities based on different model coverage criteria. We also want to experiment with the UML-based MBT tools from Section 7, but first the effort to accommodate their different semantics (see also [4]) and corresponding model transformations must be evaluated.

Acknowledgments. This work was partially supported by the EC-funded project MODELPLEX [14]. We thank Roger Kilian-Kehr for useful comments on a draft of this paper.

9 References

1. Ali, S., Briand, L., Jaffar-Ur Rehman, M., Asghar, H., Iqbal, M.Z., Nadeem, A.: A State-Based Approach to Integration Testing Based on UML Models. Information & Software Technology 49 (11–12), pp. 1087–1106 (2007)
2. Benedetto, C.: SOA and Integration Testing: The End-to-end View. In: SOA World Magazine 6(8), (2006)

3. Business Process Modeling Notation (BPMN) Specification, Final Adopted Specification. Technical report, Object Management Group (OMG), Online at: http://www.bpmn.org
4. Crane, M., Dingel, J.: UML vs. Classical vs. Rhapsody Statecharts: Not All Models Are Created Equal. Software and System Modeling 6(4), pp. 415–435 (2007)
5. Dotan D., Kirshin A.: Debugging and Testing Behavioral UML Models. OOPSLA Companion 2007, pp. 838–839. ACM Press (2007)
6. Foster, H., Uchitel, S., Magee, J., Kramer, J.: WS-Engineer: A Model-Based Approach to Engineering Web Service Compositions and Choreography. In: Test and Analysis of Web Services, pp. 87–119, Springer (2007)
7. Frantzen, L., Huerta, M. N., Kiss, Z. G., Wallet, T.: On-The-Fly Model-Based Testing of Web Services with Jambition. In 5th Int. Workshop on Web Services and Formal Methods (WS-FM'08), LNCS. Springer (2009). To appear.
8. García-Fanjul, J., de la Riva, C., Tuya, J.: Generation of Conformance Test Suites for Compositions of Web Services Using Model Checking. In: Proc. of TAIC PART 2006, pp. 127–130. IEEE Computer Society (2006)
9. Hartman, A., Katara, M., Olvovsky, S.: Choosing a Test Modeling Language: A Survey. In: Haifa Verification Conference 2006, LNCS, vol. 4383, pp. 204–218. Springer (2006)
10. Hartman, A., Nagin, K.: The AGEDIS Tools for Model Based Testing. In: UML Satellite Activities 2004, LNCS, vol. 3297, pp. 277–280. Springer (2004)
11. Hartmann, J., Imoberdorf, C., Meisingerm M.: UML-Based Integration Testing. In: Proc. of ISSTA 2000, pp. 60-70. ACM Press (2000)
12. Helfen, M., Lauer, M., Trautwein, H.M.: Testing SAP Solutions. SAP Press (2007)
13. Kavantzas, N., Burdett, D., Ritzinger, G., Lafon, Y.: Web Services Choreography Description Language Version 1.0. W3C Candidate Recomm., Technical report (2005)
14. MODELPLEX Project. Funded by European Commission, FP6, Grant no. 034081. Online at: http://www.modelplex.org
15. O'Leary, D.E.: Enterprise Resource Planning Systems – Systems, Life Cycle, Electronic Commerce and Risks. Cambridge University Press (2000)
16. SAP AG, Enterprise SOA in a Nutshell. (2007), Online at: http://help.sap.com/redirect_sdn_esoa/redirect_esoainanutshell.htm
17. Utting, U., Legeard, B.: Practical Model-Based Testing – A Tools Approach. Morgan Kaufmann Publ. (2007)
18. Web Services Reliable Messaging (WS-ReliableMessaging), Version 1.1. OASIS Consortiom. Online at: http://docs.oasis-open.org/ws-rx/wsrm/v1.1/wsrm.pdf
19. Wieczorek, S., Roth, A., Stefanescu, A., Charfi, A.: Precise Steps for Choreography Modeling for SOA Validation and Verification. In: International Symposium on Service-Oriented Software Engineering (SOSE'08), pp. 148–153. IEEE Computer Society (2008)
20. Wieczorek, S., Stefanescu. A., and Großmann, J.: Enabling Model-Based Testing for SOA Integration Testing. In: Proc. of 1st "Model-based testing in practice" workshop (MOTIP'08), pp.77–82. Fraunhofer IRB Verlag (2008)
21. Wieczorek, S., Stefanescu, A., Schieferdecker, I.: Test Data Provision for ERP Systems. In: Int. Conf. on Software Testing, Verification and Validation (ICST'08), pp 396-403. IEEE Computer Society (2008)
22. World Wide Web Consortium (W3C): Web Service Glossary. Version 20040211. Online at: http://www.w3.org/TR/ws-gloss
23. Woods, D., Mattern, T.: Enterprise SOA – Designing IT for Business Innovation. O'Reilly (2006)
24. Zaha, J. M., Barros, A., Dumas, M., ter Hofstede, A.: Let's Dance: A Language for Service Behavior Modeling. In: International Conference on Cooperative Information Systems (CoopIS 2006). Springer (2006)