Modeling and Model-based Testing of Service Choreographies

vorgelegt von
Diplom-Informatiker
Sebastian Wieczorek

Von der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
Dr. Ing.

Promotionsausschuss:

Vorsitzender: Prof. Dr. Sabine Glesner
Berichter: Prof. Dr. Ina Schieferdecker
Berichter: Prof. Dr. Lionel Briand

Tag der wissenschaftlichen Aussprache: 07.05.2010

Berlin 2010
D 83

Abstract

The testing of service-based applications is an important but challenging activity. Especially the integration testing is a difficult task that needs to cope with the message-based communication in the service-oriented world. In this thesis, a model-based approach to service integration and integration testing is proposed. The necessary research work to realize its phases is the main contribution of the dissertation.

First, MCM, a domain-specific language for service choreography modeling, is introduced together with a precise semantics that makes it suitable for integration testing. Then, a framework for generating service integration tests is presented, incorporating three different model-based test generation techniques that can be chosen according to the test context. Further, it is explained how the generated test cases are transformed into concrete test scripts, thus enabling their execution on an enterprise service-based application. Finally, the conducted case study of the MCM-based approach in an industrial setting is described.

Contents

1 Introduction **11**
1.1 Topic . . . 11
1.2 Goal . . . 13
1.3 Thesis Structure . . . 13
1.4 Scientific Contributions . . . 15

2 Related Work **19**
2.1 Testing . . . 19
2.1.1 Overview . . . 19
2.1.2 Test Coverage Criteria . . . 21
2.1.3 Automated Test Generation Techniques . . . 22
2.2 SOA . . . 25
2.2.1 Challenges . . . 25
2.2.2 Service Design and Development . . . 26
2.2.3 Service Composition . . . 27
2.3 SOA Quality . . . 30
2.3.1 SOA Quality Assurance . . . 30
2.3.2 Challenges of SOA Testing . . . 31
2.3.3 The SOA testing stack . . . 33
2.3.4 Service Integration Testing . . . 35

3 Problem Statement **37**
3.1 A simple buyer-seller example . . . 37
3.2 Complex Interaction Patterns . . . 38
3.2.1 SOA Service Integration . . . 38
3.2.2 SOA Choreography Modeling . . . 41
3.3 SOA System State . . . 45
3.3.1 System Data Challenges . . . 47
3.3.2 Input Data Challenges . . . 49
3.4 Summary . . . 51

4 General Approach **53**
4.1 Overview . . . 53
4.2 Choreography Modeling . . . 58
4.2.1 Domain-specific Modeling . . . 59
4.2.2 Choreography Viewpoints . . . 59
4.2.3 Consistency . . . 64
4.3 Model-based Integration Testing . . . 66
4.3.1 Error assumption . . . 67
4.3.2 Coverage . . . 68
4.3.3 Industrial Requirements . . . 70
4.4 Summary . . . 71

5 Message Choreography Model **73**
5.1 MCM Overview . . . 73
5.2 MCM Syntax . . . 76
5.3 MCM Semantics . . . 79
5.3.1 Trace Definitions . . . 79
5.3.2 Consistency Relation of GCM and CS . . . 80
5.4 MCM Transformation Semantics . . . 81
5.4.1 Introduction to Event-B. . . . 82
5.4.2 Design Considerations of the Transformation. . . . 82
5.4.3 Transformation Description. . . . 83
5.5 MCM Tool Support . . . 86
5.5.1 Ensuring Consistency between Requirements and MCM 87
5.5.2 Ensuring Consistency between MCM Viewpoints . . . 87
5.5.3 Ensuring Consistency between MCM and Implementation . . . 90
5.6 Summary . . . 91

6 Test Generation **93**
6.1 Test Generation Process . . . 93
6.2 Test Generation Framework . . . 96
6.3 FSM-based test generation . . . 98
6.4 Model-checking based test generation . . . 101
6.5 Heuristics-based test generation . . . 103
6.6 Summary . . . 106

7 Case Study **107**
7.1 Case Study Design . . . 107
7.1.1 Setting . . . 107
7.1.2 Approach . . . 109
7.2 Case Study Execution . . . 110
7.2.1 Modeling . . . 110
7.2.2 Test Generation . . . 110

7.2.3 Comparison of Test Generators 111
7.3 Case Study Evaluation 113
7.4 Summary 114

8 Conclusion 117
8.1 Summary 117
8.2 Outlook 118

List of Figures

1.1 High-level structure of the thesis 14

2.1 A common classification of software testing 20
2.2 SOA testing layers 33

3.1 A sketched protocol of the running example 38
3.2 Example scenarios of asynchronous communication 40
3.3 Decomposition of SOA 45
3.4 Challenges related to SOA test data 47
3.5 Illustration of input data constraints in an ERP system . . . 50

4.1 Envisioned development process for SOA 54
4.2 Overview of the test concretization and execution 57
4.3 Service integration testing implemented using SAPs eCATT framework 58
4.4 Message event sequences for the running example 61
4.5 The global receive viewpoint for the running example 62
4.6 Local viewpoint models for the running example 64
4.7 Consistency relations in Choreography Modeling 65
4.8 Assuming an EO channel, global coverage does not imply local coverage 69
4.9 Assuming an EO channel, local coverage does not imply global coverage 70

5.1 GCM (top) of the choreography and LPMs of the buyer (left) and the seller (right) 75
5.2 Overview of MCM modeling tools 87
5.3 Interactive simulation of MCM 88

6.1 The test generation framework 96
6.2 Steps of the test generation for finite state machines 99
6.3 Integration of the IBM test generator 104

7.1 The GCM of an example pilot 112

Chapter 1

Introduction

1.1 Topic

Software development is able to generate powerful means for solving diverse problems. The demand for software that is not only providing an answer for a specific question, but a comprehensive system guiding through various scenarios however has lead to complexity issues in software development. Decomposition of software systems architecture on various levels (e.g. libraries, objects, sub-components) has been the answer so far. Also the latest trend of providing loosely coupled components, which can be composed dynamically to form powerful solutions for complex problems heads in this direction.

The fundamentals of Service Oriented Architectures (SOA) - 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). The application of Web-Service-like interaction SOA however adds the advantage of interconnection in an object-model-independent way and hence makes interaction between services much easier.

SOA provides methods and frameworks to compose single services in order to realize complex business scenarios. Modeling and implementation of such services based on technical specifications like XML, SOAP, and WSDL is well-understood. The challenging part of the SOA-based development is the integration of different services according to the defined business processes. 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. 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. Choreography languages like WS-CDL [W3C04b], BPMN [OMG08a], or Let's Dance [ZBDtH06] were introduced to describe the interaction protocols between a set of loosely coupled components communicating over message channels from the perspective of a global observer.

From an industrial perspective, SOA is gaining pace towards becoming mainstream. Forrester studies and surveys [For08, Hef09] of more than 2200 IT decision makers across North-America and Europe show that 2/3 of the companies expect to be using SOA by the end of 2009 while 60% of those already using it are expanding their usage. Leading firms now use SOA on more than 50% of their solution delivery projects.

Such a widespread SOA adoption implies that the quality assurance of the service-based systems becomes an activity of paramount importance, with a special focus on SOA testing. While service unit testing is usually well researched [BBMP08, TCP$^+$05, OX04, BD07] and consistently deployed in practice, the field of service integration testing poses several new challenges [CP09, BHB$^+$07]. The difficulties to be overcome are due to the heterogeneity, high distributivity, dynamicity, and loose coupling of the service-based systems. These complexity properties are taking their toll on the testing process.

In general, new architectural and programming paradigms, also create a demand for innovation in the program verification and validation approaches. In the past, for example the concept of model-driven software development (e.g. OMG's MDA strategy) has promoted the use of models in testing, too. The term 'Model-based Testing' (MBT) emerged, standing for any test generation method based on models. In general MBT approaches use abstract behavioral and structural system information to generate a set of test cases. The test cases can be seen as execution traces defining the navigation through the system states. Test generators aim to cover model features (e.g. every state or all transitions in a state chart) by deriving sets of test cases from models.

A model-driven approach to SOA service integration helps to address the mentioned complexity challenges, as it allows for a general solution, applying state of the art tools and techniques for formal reasoning about service models [BD07] and model-based testing [UL07, BDG$^+$08]. Furthermore it has the prospect to capitalize on the research and experience made in protocol conformance testing (e.g. [DSU90, Lai02]). However, the enabling prerequisite for the adaptation of such techniques is a modeling language that on the one hand captures the essential properties of the service integration at a suitable level of granularity and on the other hand is precise enough to allow formal analysis and productive test generation.

1.2 Goal

As of today, various service choreography languages are emerging, each with a different scope and granularity. They provide means for describing intended service compositions, which are a prerequisite for a successful design and development of SOA applications. Further, having a language for formal specification with the right level of abstraction, enables to utilize mature quality controlling techniques, such as formal verification or model-based testing, in the context of SOA. Unfortunately, none of the current choreography languages is suitable for such an intended, quality-driven SOA development process.

Aim of the thesis is to contribute a development approach for SOA that especially supports and drives the testing of service compositions. This includes to provide a modeling method, which allows to specify service compositions simultaneously from a global and from local component's view. The envisioned models should supply consistent, formal and deterministic descriptions in order to drive a model-based service integration. Further, the thesis should give evidence of the practicability of the envisioned approach. Therefore, means for the design and quality control of service compositions should not only be developed in theory, but implemented and evaluated in an industrial environment.

The work is conducted in collaboration with SAP, a leading provider of business software with a core competence in Enterprise Resource Planning (ERP) software, which supports business processes for whole companies and integrates organizational parts and functions into one logical software system. Besides delivering SOA-enabled software, SAP further provides SOA methodology guidelines and professional services [WM06]. At SAP service integration testing is associated with huge efforts even though the test execution has already been automated to limit the resource consumption in terms of manpower. Model-based testing techniques are believed to additionally realize a high degree of automation in the test development phase.

1.3 Thesis Structure

As explained, the thesis goal is to contributes to a development approach for SOA, which focuses on the design and testing of service compositions. Apart from the introduction and conclusion, the thesis is divided into four main parts, as depicted in Figure 1.1. In the following, their content and relation is shortly discussed.

- *Analysis.* Chapter 2 gives an overview of related research fields that have been identified in Sections 1.1 and 1.2. Based on this review, the problem statement for the dissertation is derived in Chapter 3. It

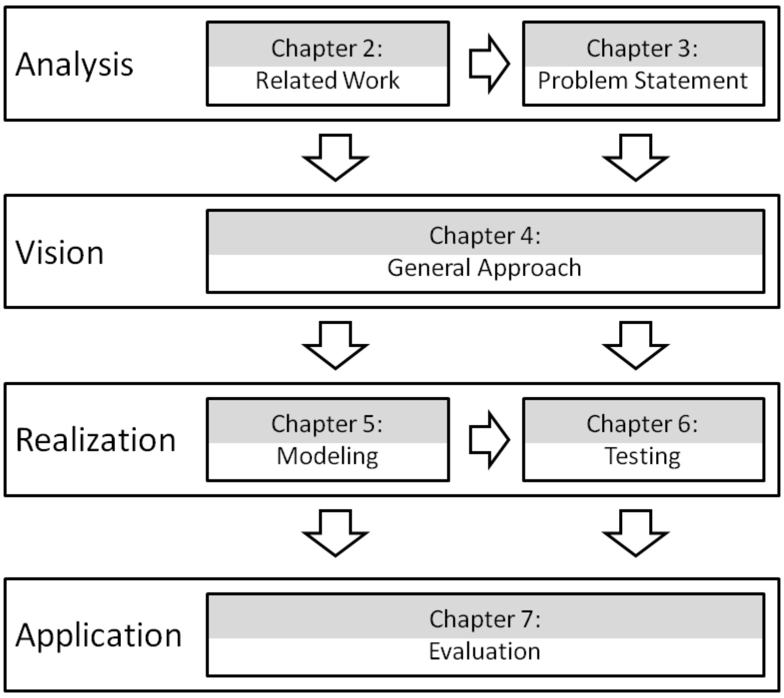


Figure 1.1: High-level structure of the thesis

suites as a motivation for the research activities that are described in the succeeding chapters.

- *Vision.* Chapter 4 describes an idealized development approach for SOA that illustrates, how the existing challenges from Chapter 3 can be overcome in general and which of the available state of the art, described in Chapter 2 can be utilized.

- *Realization.* Chapter 5 and Chapter 6 are explaining concrete solutions for the modeling and testing phase of the envisioned SOA development process of Chapter 4. Chapter 5 introduces the choreography language MCM, including syntax and semantics, and describes the available tool support. Chapter 6 explains, how MCM instances are used to derive test cases and presents a test framework that implements the testing by integrating complementary test generators. As described in Section 1.4, these chapters encapsulate the core scientific contributions of this thesis.

- *Application.* Chapter 7 finally explains the utilization of the conducted work in an industrial setting. It portrays a case study of the developed tools, describes the derived information and interprets the outcome.

1.4 Scientific Contributions

In this section, the relevance of the dissertation is explained. Based on the publications, listed and described below, the contributions to the state of the art are shown in the second part. The section is concluded by naming related activities.

Publication Description. The dissertation is based on various scientific publications, describe below:

[1] discusses modeling, generation and provision of test data for model-based testing of ERP software and identify several challenges in this area.

[2] clarifies the four main challenges regarding the provision of test data for automatic testing of ERP software: system test data supply, system test data stability, input test data constraints and test data correlation, and discusses several possible solutions.

[3] examines test objectives and test coverage criteria driving the model-based test generation in the context of integration testing for service-oriented architectures.

[4] analyses the SOA testing stack and the different objectives of its layers. It further explains why traditional and currently discussed testing techniques alone are not sufficient to cover all relevant testing layers.

[5] suggests a holistic design and development method combining test-driven and model-driven development for SOA architectures, where test-driven development is used on component level and model-based testing on integration and system level.

[6] identifies general requirements for choreography languages that can be used for automatic verification and validation activities. It further motivates a discussion about applicable techniques for service choreography modeling and whether existing choreography languages cover the identified needs.

[7] shapes various interpretations of service choreographies, which were left unspecified in state of the art choreography approaches and introduces the message choreography modeling (MCM) environment incorporating these contributions, including a detailed description of the syntax and semantics of MCM.

[8] present an overview of the developed model-based integration testing (MBIT) approach based on MCM and explains how it fits into the SAP testing methodology for SOA.

[9] introduces a model-checking based test generation for service choreographies based on MCMs. These are translated into EventB and processed by the model checker ProB.

[10] describes a four-step approach for service integration testing and a heuristics based test generation for service choreographies based on MCMs. These are translated into executable UML models using Java as action language and used for test generation and model debugging by a model execution engine.

Contributions. Like the thesis structure, also the scientific contributions of the dissertation can be divided into the four parts, illustrated in Figure 1.1. In the following, for each part the innovative core is described.

- *Analysis.* In this first part of the thesis, service integration has been identified as an untackled challenge of SOA [4]. Further requirements have been described that enable quality aware choreography modeling to answer this challenge in the SOA life cycle [6]. Also the test data types for enterprise SOA has been classified [1] and the associated test data provision challenges have been identified [2].
- *Vision.* The thesis incorporates a vision of a holistic SOA development [5] and shapes a proposal of a concrete development process [8].
- *Realization.* This part of the thesis describes a domain-specific service choreography language called MCM, which is suitable for a model-driven development of SOA, including a dedicated viewpoint semantics and consistency definition [7]. Further, the utilization of choreography models for SOA integration testing has been enabled by defining concrete objectives [3] and proposing a general testing process [10] and frameworkthat integrates mature test generation approaches. Further a model checking-based test generation algorithm is introduced that is compatible to MCM's viewpoint semantics [9].
- *Application.* The final part of the thesis shows the integration of the realized modeling and MBIT concepts in an industrial environment [8]. It further gives concrete evidence of their applicability in practice.

Related Activities. The PhD candidate served as an editor for the 2^{nd} workshop on Model-based Testing in Practice (MoTiP'09 [11]) and contributed to the research projects MODELPLEX[1] and DEPLOY[2]. MODELPLEX is an EU FP6 project that has the objective to develop an open

[1]Full titel: MODELing solutions for comPLEX systems, online at: `http://www.modelplex.org`

[2]Full titel: industrial DEPLOYment of system engineering methods providing high dependability and productivity, online at: `http://www.deploy-project.eu`

solution for complex systems engineering, improve quality and productivity, lead its industrialization and ensure its successful adoption by the industry. DEPLOY is an EU FP7 project that aims to make major advances in engineering methods for dependable systems through the deployment of formal engineering methods.

Chapter 2

Related Work

In this chapter an overview of the research fields and activities that are encapsulating the dissertation is given. It starts with an introduction to software testing in general. Afterwards, the challenges and activities regarding the development of SOA are explained. Finally, the related work on SOA testing is discussed more detailed.

2.1 Testing

Testing is an important way to realize software quality. In the following, a high-level classification of testing is given in the overview section. Afterwards, test coverage criteria are introduced as measures to determine the significance of tests. Finally, a summary of automated test generation techniques is presented.

2.1.1 Overview

From an organizational standpoint, the aim of testing is to lift a software up to a certain level of quality and raise the customers confidence in its functioning [PVDBJvV04]. Considering the testers motivation, testing is regarded as the process of executing a program with the intent of finding errors [MS04]. According to [Tre04] (see Figure 2.1) software testing can be can be classified according to its characteristics, scale and accessibility.

Characteristics. Testing is always performed with a certain test objective, i.e. showing that the system is complying to some characteristics. According to [GVV08] these can be partitioned into functional and non-functional characteristics, each demanding unique testing approaches.

- *Functional testing* determines whether the envisioned functionality is realized by the given implementation.

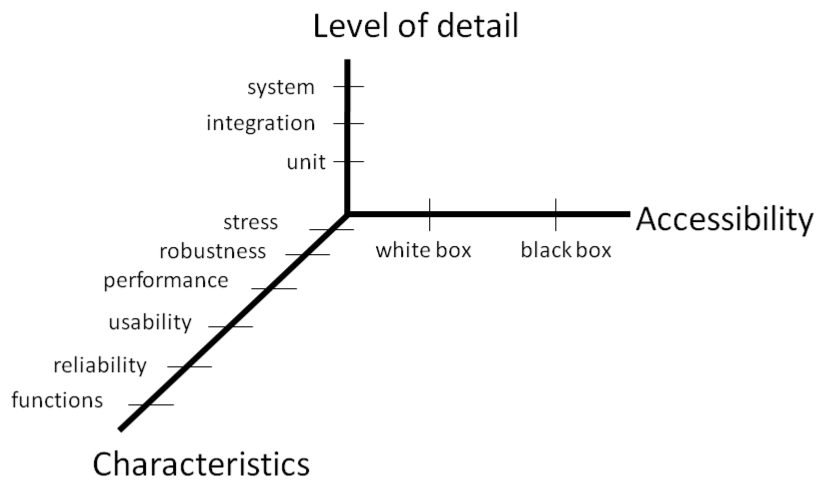


Figure 2.1: A common classification of software testing

- *Non-functional testing* is considering all those quality attributes that are not belonging to functional testing. There are various forms of non functional testing, e.g. performance testing, load testing, stress testing, usability testing, maintainability testing, reliability testing and portability testing. The testing of security aspects as well as interoperability testing are usually also regarded as non-functional, even though they are classified as functional testing in the ISO/IEC 9126 standard [ISO01].

Scale. Traditionally, three different testing layers have been advertised to enforce functional correctness of monolithic software systems (see for instance [Jor08]

- *Unit Testing.* This is carried out alongside the development of single software units like methods, procedures or classes, which are then tested in isolation.

- *Integration Testing.* This deals with the testing of aggregated functionality like clusters of classes or sub-systems.

- *System Testing.* This comprises the fully integrated application, usually using its externally exposed interfaces.

It has been argued convincingly in [ABR$^+$07] that everything apart from unit testing is a form of integration testing. However it still makes sense to distinguish between the different levels of integration testing, as long as they demand distinct strategies and techniques (like integration testing and system testing in the traditional approach).

Accessibility. The chosen approaches and techniques for software testing depend on the type of information that is available or intended to be utilized [UL07]. In general, two options exist:

- *White-Box Testing.* These testing techniques are utilizing the implementation code of the system under test for the design of the tests.
- *Black-Box Testing.* In contrast to white-box testing, the test design is based purely on externally available information, while the internal structure of the system under test is unknown or ignored. Externally available information might be the interface description, models that are specifying the intended behavior or textual requirements.

Because the wide range of testing characteristics cannot be tackled in this thesis, in the following the scope is limited to functional testing.

2.1.2 Test Coverage Criteria

To assess the significance of a test, coverage criteria can be used. Consequently, apart from just measuring the test coverage, testing techniques have been developed to directly guide the test generation in order to achieve a targeted degree of coverage. In the following, the most common categories of coverage criteria are introduced, including an indication of test generation techniques that address them.

Requirement coverage If each software requirement has been addressed at least by one test case, requirement coverage is achieved [Jor08]. Because software requirements are usually not formalized, the test designer's interpretation of requirements plays a vital role for the later software quality. Also the quality of the requirements itself (e.g. the granularity or clarity) influences the potential of fault detection. Therefore the main objective of requirement coverage is not fault detection but software validation that proves some desired features for the software consumer [Bin99]. However, requirements can also be annotated to code (e.g. as assert statements [BCZ05]) as well as to behavioral and static models (e.g. as properties [FHP02]). In these cases automatic model- or code-based test generation techniques (discussed below) can be utilized to achieve a certain requirement coverage. Otherwise manual work has to be carried out to link requirements with test steps, cases or the test suite [BJL$^+$05].

Data coverage To achieve a certain data coverage during test execution, combinations of input data have to be applied for test runs. Covering all possible combinations usually is infeasible, because if at least one variable has an infinite domain (e.g. a real number) the input domain cannot be covered exhaustively. But also a combination of large finite input domains or a high number of variables prohibit exhaustive coverage. To reduce the input variants, equivalence classes have been introduced, partitioning the input domain into sets of values that all produce a similar system behavior [WJ91]. Different data coverage criteria can be deduced from equivalence classes. Examples are boundary value coverage (testing the input values at the boundary of the data set) and statistical data coverage (testing with the most probable values of each data set) [MS04]. To reduce the amount of tests for every input value combination further, pairwise data coverage has been proposed [CDPP96].

Code coverage The automatic generation of tests by analyzing source code has been studied very extensively in the past. The so called white-box testing techniques aim at code coverage features like statements, conditions, decisions, and predicates. The general idea is to identify execution paths through a program and find the enabling input test data. In the industry test data selection usually is a manual process, but research in recent years has investigated strategies to support that extremely costly and difficult task [McM04]. Usually code coverage is transformed to coverage criteria of the application's control flow graph [ZHM97].

Model Coverage Model-based testing is a collection of different techniques that utilize mainly behavioral models in the testing process. Similar to code-based coverage, model-based test generation aims to cover certain entities of a model (e.g. states, transitions, predicates in conditions). In [UPL06] a classification of MBT techniques, distinguishing between the used model, the test generation technique and the test execution has been described in some detail. As explained in [UL07] model-based testing can be introduced by stepwise automating the activities of the classical manual testing process. Intermediate stages of this automation are Capture/Replay Testing, Script-based Testing, and Keyword-driven Testing. Analysis of the positive effects of MBT on software quality and cost reduction can be found in various publications, e.g. [UL07, BBJ^{+}03, AD97].

2.1.3 Automated Test Generation Techniques

In the following, state-of-the-art test generation techniques will be introduced that address the above test coverage types. Because automatic test generation techniques depend on formal specifications (either code or models) the overlap of techniques for the described types of coverage is quite high. Therefore, instead of classifying testing techniques according to the targeted coverage criteria, they can be grouped according to the used input specification: into structural and behavioral test generation techniques. In general, achieving a certain coverage for all of the above described criteria can be provided by either structural or behavioral techniques. However, data coverage is usually associated with structural techniques, while code coverage is commonly tackled by behavioral test generation.

Structural Test Generation

Structural information about a system can be given in various forms. Most common are interface descriptions (e.g. headers of methods and procedures on code-level or UML class diagrams on modeling level) that are specifying the domain of the input and output data. Data coverage can be addressed using MBT, by automatically generating input data from the specified domains as discussed above. Other approaches provide extra information about the correlation and constraints between input data values. For example the classification tree editor (CTE) [DDB$^+$05] can be used to identify, visualize and link input data values. Applying CTE, a test generator can compute a minimal test suite to achieve pair-wise or all-feasible-combination coverage.

Behavioral Test Generation

Like from code, control flow graphs can also be extracted from behavior models (e.g. state machines or activity diagrams) [OLAA03]. Model-based coverage criteria and test generation approaches are therefore closely related to code-based techniques. Various case studies (cf. [LY96]) describe implementations of different coverage criteria for model-based test generation. Similar to the automatic test generation on code level, not the definition of coverage criteria and processing of execution paths through the model itself, but the assignment of test data to actually execute them is perceived as the key issue. Therefore some work has already been carried out to utilize known test data generation methods on code level (e.g. symbolic execution) for MBT. In the following an overview of some of the most prominent code-based test generation techniques is given and linked to related MBT approaches.

Symbolic execution was investigated in the 70ties. It does not execute the whole program, but evaluates the assignment of variables for a chosen path through the code. During this processing, the path enabling constraints are determined for each variable. This can be done either by forward or backward traversal, with the difference that backward traversal does not need storage for intermediate symbolic expressions, while forward traversal allows early detection of infeasible paths [Kin76]. The application of symbolic execution however is limited to rather simple code because the needed constraint solving techniques require huge computational power and can hardly cope with dynamic structure like lists or trees [Edv99]. In the domain of MBT, symbolic execution techniques benefit from a higher level of abstraction and therefore have been applied successfully in various cases (cf. [SBLR07, FTW$^+$06, KPV03].

Domain reduction is a way to tackle the complexity problem of symbolic execution. It takes the input data constraints computed by symbolic execution and reduces the variable domain stepwise. In parallel also the constraints are simplified using the domain information. If no further reduction can be made, a random value is assigned to the input variable with the smallest domain, in order to restart the process [DO91]. Dynamic domain reduction is a further enhancement, reducing the input domain already during symbolic execution [OJP99]. Constraint solvers that are using domain reduction techniques have also been utilized for MBT (cf. [AAF$^+$04]).

Actual execution is a dynamic algorithm that uses randomly generated data and monitors the resulting computation. If the intended path is left, backtracking is used to re-execute the last branch. As backtracking might accidentally change the program flow at an earlier point and much iteration is needed until suitable input is found, this method has not found broad acceptance. Therefore a hybrid method combining actual with symbolic execution has been developed [GMS98]. Also for MBT actual execution has been adopted. E.g. [HN04] reports on a model execution engine that is utilized for test generation.

Local search is another representative of the dynamic test generation algorithms. It is enhancing the actual execution by evaluating the branch predicate at the point where the execution leaves the desired path. A branch distance is computed, describing how close the branch predicate was to the desired value [Kor90]. Model-checkers, which are used quite frequently for MBT [FG09], have to adopt certain search techniques as well, in order to explore the state space in an efficient way. The application of pruning, as described in [EKRV06], is an example for local guidance.

The chaining approach in contrast to the previous search techniques, only aims to find any path to a specified goal in the source code. It hence relaxes the criteria of finding input data for a specific path, given the fact that for statement coverage of the code, it is sufficient to find any path to an uncovered statement [FK96]. The special characteristic of this technique is to identify a chain of nodes (branches in the code) that are vital to reach the goal. If no input value set for the chain can be produced, additional nodes are added. In this way, loops can be addressed more effectively [McM04]. It can be assumed that the chaining approach is applicable to MBT as well, even though no evidence can be given.

Simulated annealing is used to loosen the restrictions of local search algorithms by allowing the search for data values to be more flexible. The values are allowed to "jump away" from the perceived goal computed by the distance function, but decreasing the amplitude steadily during computation. The aim behind it is to decrease the possibility of being trapped in a local minimum, as it might easily happen for search based algorithms. An application of simulated annealing for MBT has been reported in [PLP04].

Genetic algorithms are providing another way to loosen the restrictions of local search. They are inspired by the evolutionary biology and produce search results in a recursive approach. From a given set of data (initial population) the most appropriate items are selected according to a predefined fitness function. These items are used for the breeding of a new population by applying the concepts of crossover (combining items) and mutation (altering items). The selection and breeding is continued until the termination condition, e.g. finding a result or reaching a time limit, is reached. The technique has been used for code-based [PHP99] as well as for model-based test generation [GBL07, LI07].

Testability transformations was proposed to improve the performance of heuristics-based (dynamic) techniques like genetic algorithms that usually suffer from structural impediments like flag variables [HHH+04]. The basic idea is to transform the program code such that the heuristics, used to find a specific value, are guided towards the intended goal. As advertised in [Har08] testability transformation could also be used to improve model-based search techniques.

2.2 SOA

As introduced in Section 1.1, SOA provides methods and frameworks to compose single services in order to realize complex software systems. In the following, the challenges that are associated with SOA are described. Afterwards current development approaches are introduced. Finally the most challenging field of service composition is discussed in more detail.

2.2.1 Challenges

While SOA is regarded as the next evolutionary step to tackle the ever growing complexity of software, it also generates unique challenges. According to [PTDL07] these (yet open) challenges can be clustered into four different areas:

- *Service Foundations.* Despite the distributed nature of SOA the common infrastructure (e.g. the middle ware) has to provide for dynamic reconfiguration facilities, end-to-end security, data and process integration, and service discovery.

- *Service Composition.* In order enable dynamicity and adaptability, SOA propagates a loose coupling of services. This however demands for a sophisticated composability analysis, self-configuration of services, and QoS-awareness.
- *Service Management and Monitoring.* Developers and providers must be able to assess the status of a composed system in a dynamic environment. Therefor a self-configuring and self-adapting management is needed that continuously monitors the application in order to detect malfunctioning and external threats and applies self healing if discovering a disruption.
- *Service Design and Development.* In order to provide flexible software solutions, a holistic engineering methodology, flexible gap-analysis for business processes, service versioning and governance applicable to SOA is needed.

The conducted work, described in the thesis, focuses on aspects of the service design and development as well as service composition. Therefore, efforts to address the challenges of these areas are described in the following subsections.

2.2.2 Service Design and Development

The growing size and complexity of problems that are tackled by software also increases the complexity of the software development process and hence the industry seeks for well defined development models that enforce efficiency and high quality. To identify, define, and formalize development activities in the service oriented software engineering (SOSE), naturally the existing work from the area of component based software engineering (CBSE) has been adopted [Vli08]. In the following, the most prominent approaches are introduced

The Rational Unified Process (RUP) [Kru00] is a development process based on UML that has been introduced for CBSE. Because of the good tool support it found wide adoption in the industry. In order to make it applicable for SOA development, [BJLP05] adjusted the existing milestone definitions and introduced additional guidelines to address service composition and infrastructure.

The Service Development Life Cycle (SDLC), described in [Pap07], provides another methodology for an iterative SOA development approach. It incorporates various models, standards and best practices for the eight described SOA development phases Planning, Analysis, Design, Construction, Testing, Provisioning, Deployment, Execution, and Monitoring.

The Service Oriented Modeling and Architecture (SOMA) approach [AGA$^+$08], in contrast to RUP for SOA and SDLC, focuses on how business processes can be translated effectively into service-based applications by bridging from requirements to design. Core activities are the service identification, service specification and service realization.

All of the mentioned approaches are addressing diverse aspects of non-adaptive SOA applications. In fact, they are perceived as state-of-the-art if applied in combination [DND09]. Thereby SDLC provides a foundational framework, while RUP offers robust industrial processes and tools, and SOMA refines the analysis and design phase.

2.2.3 Service Composition

When designing service-based systems (SBS), the desired functionality of the software is partitioned into units, much like in component-based systems (CBS). However SOA becomes different on the integration level. Unlike components in a CBS, services are loosely coupled by message exchange and hence special care has to be applied to provide for their faultless interaction. The fundamentals of SOA - decomposition of systems into services - have been provided by already established concepts like the Common Object Request Broker Architecture (CORBA) [OMG04] and the Distributed Component Object Model (DCOM) [Mic96]. The application of web-service-like interaction however adds the advantage of interconnection in an object-model-independent way and hence makes interaction between services much easier.

XML [W3C08] emerged as the standard format for information exchange between web services, utilizing XML schema to define message types and using communication protocols such as the Simple Object Access Protocol (SOAP) [W3C07] for message transmission. The Web Service Description Language (WSDL) [W3C01] is the de facto standard for describing interfaces of web services, e.g. including the public ports, provided service operations and associated message formats. However it does not specify any behavioral information.

Service Composition Overview Building on top of the above described standards, there are two concepts for describing the service composition for SBS, *service choreography* and *services orchestration* [MM04]. Adopting the definition of [Pel03, BDO05], both forms of composition deal with the collaboration of multiple services that are each possibly under the control of a different party.

However, service orchestration on the one hand refers to an executable process composed by theses services that is under control of a single party. Service orchestration specifications hence always represents the perspective of this controlling party. Service choreography on the other hand is characterized by a collaboration of multiple parties without a global controller. Instead, interaction patterns for each party are defined that result in a successful collaboration.

To illustrate the difference between orchestration and choreography with a real-life example, the control structures of an classical orchestra and a ballet ensemble is used. While the musicians in an orchestra are controlled by a conductor, the dancers in the ballet are collaborating by taking their part in a previously agreed routine.

Because both service composition approaches depend on the collaboration of multiple parties, they are demanding formal ways of specification, such that participants can understand and agree on their role. In the following, current approaches to specify service orchestration and service choreographies are introduced.

Service Orchestration As explained above, a service orchestration model defines the order and the conditions of service invocations by a controller. In general orchestration models may be based on any process modeling languages, such as UML activity diagrams, Petri-nets, state charts, rule-based orchestration, activity hierarchies and π-calculus [DS05].

In the industrial world, the idea to describe business process that are spanning over multiple web services was picked up by Microsofts XLANG initiative [Tha01] and IBMs Web Services Flow Language (WSFL)[Ley01] in 2001. These two initial approaches were then combined to form the Business Process Execution Language for Web Services (BPEL4WS) specification [CGK+02] in 2002. Later, BPEL4WS was re-factored into the WS-BPEL [OAS07a] and submitted for standardization by a consortium including the market leading enterprise software vendors IBM, Microsoft, SAP and Oracle.

WS-BPEL is aiming at the definition of executable business processes by concatenating activities that are associated with operations on web-services. Also constructs for conditional branching, parallelism and loops are provided. In this way multiple envisioned message exchange sequences between multiple parties can be specified in one model. Numerous commercial tools (e.g. IBM's *WebSphere Process Server*, Oracle's *BPEL Process Manager*, SAP's *Exchange Infrastructure*, Parasoft's *BPEL Maestro*, Microsoft's *BizTalk Server*) as well as various open source tools (e.g. *Apache ODE*, *bexee*, *OSWorkflow*) exist that are able to execute WS-BPEL processes [MXS09]. Roughly, all of these execution engines take the role of the orchestration controller that receives and forwards messages between the participants according to the specification.

In order to support the modeling, common interaction patterns between orchestration participants have been identified [VdATHKB03]. These have also been used to evaluate the expressiveness of other service composition languages and drive the enhancement of standards [VdADtH03].

As the industry already picked up and productized the classical engineering techniques for WS-BPEL, current research activities are advancing in the direction of formal verification. The common goal is, to provide means for static analysis of WS-BPEL specifications that are able to detect undesirable situations, such as the unreachability of activities, unconsumable messages or multiple activities competing for the same message (cf. [OVvdA$^+$07, FBS04]).

Since its arrival, BPEL has been dominating the market for service orchestration. The biggest competing approach, the XML Process Definition Language (XPDL) [WfM08] is supplying an execution semantics for the graphical OMG standard Business Process Modeling Notation (BPMN) [OMG08a]. Similar to BPEL, there are various commercial and open-source tools available to both model and execute XPDL processes (e.g. Adobe's *LiveCycle Workflow*, Software AG's *Crossvision BPM*, IBM's *FileNet Business Process Manager 4.0*). The mentioned BPMN itself is not an orchestration language as it abstracts from the execution semantics. Instead, it is used to describe business processes independent of the actual realization.

Service Choreography According to the W3C Web Service Glossary [W3C04a] "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". As mentioned above, the main characteristic of choreography is that no centralized control exists. Choreography models describe the communication between a set of loosely coupled components and in that sense defines the allowed ordering of message exchanges between these components. The communication is observed from a global perspective, while the local behavior of the involved components is not considered as far as it is not affecting the communication.

Several choreography modeling languages have emerged in the last few years. Some of the more prominent languages are the Web Service - Choreography Description Language (WS-CDL) [W3C04b], BPEL4Chor [DKLW07], and Let's Dance [ZBDtH06]. They vary in several regards such as abstraction level, formal grounding, target users, etc.

While WS-CDL is a choreography language that targets the implementation level and directly builds on WSDL. In contrast BPEL4Chor and Let's Dance focus on high-level service interaction modeling in early design phases and target business analysts as key users. As mentioned above, the OMG's current version BPMN 1.2 [OMG08a] is a language to describe business processes rather than targeting choreography modeling. However the recent draft for BPMN 2.0 [OMG08b] will most probably include choreography modeling that aims to be understandable by business users and technical developers alike.

The current approaches to choreography modeling are still quite immature compared to orchestration modeling. There is little tool support for any mentioned language. BPEL4Chor and Let's Dance are research initiatives that solely provided prototypes for modeling, while WS-CDL can claim at least two vendors (Pi4 Tech and WS-CDL+). Nevertheless, especially WS-CDL has been subject to research recently. Especially the formal verification of choreography properties like local enforceability [DW07], deadlocks and unconsumable messages [KP06] has been considered, but also the conformance of a participants' behavior to a given choreography [DD04].

2.3 SOA Quality

The last section dealt with an introduction to SOA development and modeling. In the following, the SOA specific quality aspects will be discussed. First, general quality assurance approaches will be laid out. Afterwards the focus is narrowed to testing, starting with the identification of challenges for SOA, before the differentiating aspects of the SOA testing stack are compared with the traditional testing layers introduced in Section 2.1.1. Finally current approaches to the most challenging SOA testing layer - service integration testing - is explained.

2.3.1 SOA Quality Assurance

Activities that are aiming to assure software quality for service-based systems can be distinguished into three different categories [BD07]:

- *Static analysis.* The examination of software (and modeling) artifacts in order to determine specific properties or to prove that certain predefined properties hold [Ost96].
- *Testing.* The execution of software with the intent of finding errors [MS04].
- *Monitoring.* The determination, whether the current execution of software preserves specified properties [DGR04].

Recent research in the SOA community has addressed all of these activities in numerous ways. Therefore the EU-funded project S-CUBE[1] was started with the goal to consolidate this disjunct work into a common approach. In an analysis of SOA quality assurance techniques [SC08a] it was concluded that there is already fundamental knowledge in all these activities, either developed particularly for SOA or adapted from traditional software

[1]Full title: Software Services and Systems Network, online at: `http://www.s-cube-network.eu`

engineering. Therefore the challenge they identified, is to combine the results of these approaches and pair them with engineering principles, techniques and methods.

This thesis is largely focused on the testing of SOA, while only touching static analysis and monitoring. Therefore in the following, an overview of the SOA specific aspects and research activities on testing is given. For a more detailed discussion about testing approaches in general please refer to Chapter 2.1.

2.3.2 Challenges of SOA Testing

When new programming paradigms, such as the ones associated with SOA, are emerging, naturally they arise the question whether there is a need for new testing methods or whether existing approaches can be adapted. More concrete, it has to be determined whether approaches and techniques, developed for traditional monolithic systems, distributed systems, component-based systems, and web applications can be adapted to service-based systems. In order to provide an answer, the particularities and challenges of SOA testing have to be analyzed.

The authors of [CP09] identified the following key distinguishing factors for SOA that generate unique challenges for the testing activities:

- *Lack of code access.* For users, services are just interfaces as they neither have knowledge about the structure of the code nor the possibility to observe its execution. These limitations are preventing any form of white-box testing for users.

- *Dynamicity and adaptiveness.* For traditional systems one is always able to determine at least the set of possible targets for a call [MRR05]. This is not true for SOA where the work flow of abstract services might be bound to concrete services retrieved from one or more registries during execution. Consequently, a thorough integration testing is hindered.

- *Lack of control.* Services are deployed independently and might be evolved without informing the consumers. Therefore service can unexpectedly change their behavior or miss service level agreements. As a result, system integrators cannot decide on a migration strategy including regression testing of the system.

- *Lack of trust.* As decisions for choosing a certain service solely depend on information that the provider is giving out, the risk exists that service providers are negatively influencing users with incorrect or inaccurate information. This limited trust in the service provider's information makes the test design more complex.

- *Cost of testing.* As test invocation by users may cause cost or other undesirable effects (e.g. an experience of a denial-of-service attack) on the provider side, extensive or repeated testing might not be feasible.

A slightly different approach has been taken in [GGN09], where the challenges of SOA testing are clustered to different items. As described below, it can be shown however that there is a mapping between these collected challenges and consequently no additional aspects can be derived.

- *Stakeholder separation.* It deals with all aspects of the challenges arising by the separation of SOA user, service aggregator and service provider and hence incorporates the previously mentioned challenges *lack of code access*, *lack of control*, and *lack of trust.*

- *Service integration.* It deals with the problems of test isolation [BAHS07] (i.e. separating testing from execution activities) and test awareness in SOA systems and hence incorporates the previous challenges *lack of code access*, *lack of control*, and *cost of testing.*

- *Service versioning and migration.* It deals with the different aspects of change management for services and hence corresponds to the previous challenge *lack of control.*

- *Service binding and reconfiguration.* It deals with the aspects of runtime binding and change management for SOA applications and hence corresponds to the previous challenge *dynamicity and adaptiveness.*

Considering the challenges derived from [CP09], some general implications can be derived. It seems reasonable that the first three items - namely *lack of code access*, *dynamicity and adaptiveness*, and *lack of control* - are dealing with technical challenges while the following items - *lack of trust* and *cost of testing* - may have to be solved on the management level.

For addressing the management group it may be necessary to provide means of interaction between stakeholders, in order to share information and rights. Whether these means of interaction have to provide for anonymity, as it is presumed in [GGN09] will be seen. However, the conservatism of major business companies and the consequent request for knowing and trusting business partners will probably lead to different solutions [O'L00].

The technical group clearly states that black-box testing techniques will have to be applied, as code access in a SOA environment is limited. Dynamicity and adaptiveness itself can only be achieved by providing detailed information about interfaces (see Section 2.2.3). Otherwise (in-)compatibilities cannot be detected and/or handled automatically. Therefore it can be assumed that model-based testing will be have a much greater impact in the testing process than in traditional industrial development setups, where modeling is carried out rather sporadically. The lack of control over parts

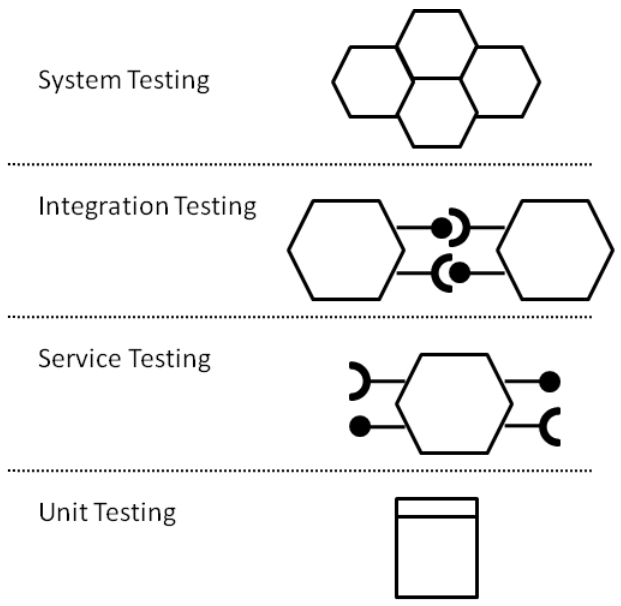


Figure 2.2: SOA testing layers

of the system implies that tests will have to be carried out not only in the development of a service-based application, but also regularly after deployment. Therefore having automatic regression tests in place is a natural conclusion [AKK$^+$05].

2.3.3 The SOA testing stack

After having discussed the general challenges of SOA testing, a closer look will be taken on the particular testing activities, in order to see which of them is affected in what way. In [UL07], the traditional testing layers (see Section 2.1.1) have been adjusted towards the support of CBS. As the general idea of partitioning applications into logical units is somehow similar to the SOA approach of encapsulating related functional units in a service, the definition of SOA testing layers can be done analogous to the one for CBS. Consequently four distinct testing layers (illustrated in 2.2) can be distinguished, namely unit testing, service testing, integration testing, and system testing. In the following each layer is introduced.

Unit Testing. Unit testing is the best understood testing layer in research and practice. In contrast to all other testing layers, unit testing focuses on getting confidence in the functional correctness and hence in the correct implementation of the algorithms. As mentioned above, it deals with single software units in isolation. During unit testing the execution context of the software unit under test is mocked. Therefore, it can be carried out in

SOA systems just like in any CBS implementation or any other software architecture, using all available tools and techniques. A good overview of such techniques can be found in [ABR$^+$07].

Service Testing. Also service testing for SOA is analog to the testing of components in the CBS world to some extend. The general focus of service testing is less on the correct implementation of algorithms but on the integration of the functional units inside the (service) component and on the fulfillment of the contractual obligations of the component's interfaces. Applying the classical definition of testing layers, service testing is part of integration testing as introduced in Section 2.1.

In the SOA world, the static aspects of service's interfaces are defined in WSDL and basically correspond to the UML component or class diagrams that are used to describe the components in the CBS world. Therefore similar testing methods as for component testing of CBSs can be used. Many of them have been described in [GTW03]. Corresponding to Section 2.3.2 the high degree of modeling in the SOA setup enables the application of MBT. Most of the approaches in the service testing literature (see [CP09] for a survey) are adding state machines (cf. [SP06, KKK$^+$06, FTdV06]), Petri-Nets (cf. [DYZ06, WBLH07]) or UML profiles for SOA to WSDL interface descriptions and applying state-of-the-art MBT approaches like constraint solving or model checking.

Integration Testing. As mentioned before, the loose coupling of service components is one of the distinguishing factors of SOA. In contrast to the CBS approach, integration testing cannot rely on homogeneous components with tightly connected interfaces. Instead the adaptability and distribution of SOA demands additional considerations for integration testing. Service components are connected by messaging channels with individual properties. The standard approach to define the minimal required transmission properties for a channel, which is connecting two service components, is by using to the Web Service Reliable Messaging (WS-RM) standard [OAS07b].

WS-RM is integrated into WSDL and incorporates the following delivery assurances, which can be associated with a channel:

- *At Least Once (ALO)* - each message is delivered at least once,
- *At Most Once (AMO)* - each message is delivered at most once,
- *Exactly Once (EO)* - each message is delivered exactly once,
- *In Order (IO)* - all messages are delivered in order (which can be combined with any of the previous delivery assurances).

Even though all the described WS-RM delivery assurances can be used to define channel requirements, currently only EO and EOIO are relevant, because ALO, ALOIO, AMO and AMOIO are not implemented by any commercial SOA platform. In practice this means that any WS-RM assignment will be realized by a stronger delivery assurance (either EO or EOIO).

Therefore, in contrast to message loss and message duplicates, the effects of message racing and its implications have to be considered during system development and should be tested thoroughly. Message racing in this context refers to situations where messages are not received in the same order as they are sent. As service integration testing is a key area of this thesis, it will be handled in more detail in Section 2.3.4.

System Testing. In the SOA world, system testing can be defined analog to the classical definition from Section 2.1.1. As the faultless interplay of the services can be assured on the integration testing level, in practice system testing is based on high-level usage scenarios and business requirements that have been defined by business analysts or customers. UI-based testing is therefore most appropriate to carry out the tests, as the system should be validated as a whole and only using access points that are available to the prospect user. Most commercial testing tools focus on UI level tests and offer certain degrees of automation [SG08].

2.3.4 Service Integration Testing

As explained, service integration testing is the most challenging layer within the SOA testing stack. Even though it is not thoroughly studied, some of the mature concepts for protocol testing in the 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 [Org94] and ETSI [ETS95]. A good summary is given in [Lai02].

Three different kinds of protocol tests can be distinguished:

- *Unit tests* are carried out during the development phase of a component by developers. They are similar to classic software unit tests.
- *Conformance tests* 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.
- *Interoperability tests* address the interoperability between components and different implementation of the same protocol. The aim is hence to guarantee the interoperability of interacting components that emanate from different distributors and implementers.

Comparing this classification to the SOA testing stack of the previous section, it becomes clear that service integration testing is related to protocol interoperability testing approaches. Under the assumption of hard-coded service integration, CBS approaches (cf. [RSM09, ABR$^+$07, GOC06]) to integration testing are applicable. These are mainly using UML-like static and behavioral models (class diagrams, state machines, collaboration and interaction diagrams) to generate tests. However their tests are generated and executed separately for each partner and hence they are different from an approach that tests the actual communication between two service components. Also the effects of asynchronous communication and multiple channels are not considered. Instead the send and receive events of a message are combined to an atomic unit.

In contrast, the approach of [PMBF05] utilizes global (sequence charts) and local behavior description (state machine) for deriving integration tests using a model-checking-based test generation. However, synchronous communication is assumed for this approach, too. Unfortunately, in most SOA systems the assumption does not hold, as service compositions are meant to be loosely coupled.

Also integration testing approaches that rely on service orchestration mechanisms and generate tests from BPEL (cf. [DYZ06, Lüb07]) are unsuited, because service orchestration as such depends on the availability of a global source of control while the envisioned SOA applications assume a shared control as they incorporate external services and aim at enabling business to business processes [WM06].

Service integration testing using choreography models is still a little covered subject. The recent surveys on SOA testing [CP09, SC08b] did not identify relevant work.

Chapter 3

Problem Statement

As described in Section 2.3.2, the development of SOA applications imposes some unique challenges to software testing. It has also been mentioned that these challenges can be separated into technical and management challenges. In the following, the technical challenges are further refined, in order to enable an assignment of concrete research activities for solving them. First, the distribution and dynamicity of the system state is described, then the integration of loosely coupled services is considered. The chapter is concluded by a short summary section, that explains how the discussed items are addressed in the following chapters. Beforehand a running example is introduced. It is not only used to illustrate the discussed problems, but also throughout the remaining thesis.

Section 3.3 is based on [1] and [2] that present challenges associated with the SOA system state in the domain of ERP software. The findings of Section 3.2 have been published in [6] and [4].

3.1 A simple buyer-seller example

In this section a running example from the enterprise world is introduced, describing a simplified communication between the service components of a buyer and a seller. An illustrating sketch of the communication is presented in Figure 3.1. The example will be used throughout the remaining thesis.

When a buyer is interested in placing a sales order it starts a conversation with the respective seller, by sending a `Request` message that provides the details of the order. This message will be answered by the seller using an `Offer` message with information about the price and terms of delivery for the desired goods. Afterwards the buyer has the choice to either accept or decline the offer. In the first case, it sends an `Order` message that successfully concludes the communication and triggers the execution of the production and/or delivery process at the seller side. In the other case, it sends a `Cancel` message that rolls back the previous communication and releases

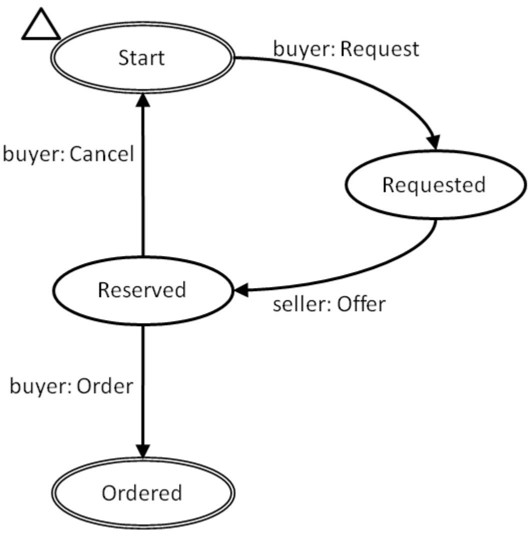


Figure 3.1: A sketched protocol of the running example

the reserved resources at the seller. In this case the protocol allows the buyer to restart the negotiation with a new request.

For the communication between buyer and seller a synchronous channel for the `Request` and `Offer` messages and a reliable asynchronous channel that does not guarantee to preserve the message order for the `Cancel` and `Order` messages will be assumed. In Section 2.3.3 the assignment of such channel properties has been discussed in general. The motivation for this specific channel assignment will be explained in Section 3.2.1.

3.2 Complex Interaction Patterns

As described in Section 2.3.4, service integration testing for SOA is not well covered by current research, while major difference between component-based and SOA testing can be found on the level of integration testing and consequently prevent the application of established CBS tools.

In the first part of this section more details on the particularities of SOA service integration are given, while the second part reasons about the inapplicability of current modeling approaches for controlling the quality of service integration in a model-driven development approach.

3.2.1 SOA Service Integration

In a component-based implementation, the communication is in most cases handled by synchronous calls between the components. Synchronous communication means that the initiator is blocked from further computation until the requested component is providing the desired answer. The SOA approach however demands a loose coupling that allows more flexibility and better distribution of the components. Therefore asynchronous channels are used in addition to synchronous ones.

Asynchronous channels provide different reliability degrees according to the WS-RM standard [OAS07b], for instance that every message is received exactly once (EO) or even exactly once in order (EOIO) (for details see Section 2.3.3). Both types of channels have mechanisms to ensure that each sent message is received once, thus preventing message loss and multiple receiving of the same message. The differentiating feature is that messages on an EOIO channel are always received in the same order they are sent, messages on the EO channel may overtake each other.

In practice changes of the message order happen when dynamic routing strategies are applied to the messages or when data corruption occurred, thus forcing a message to be sent again. On EOIO channels, message racing is usually prevented by re-sorting the messages at the receiver side. In the following, the necessary considerations for assigning channels to a service communication are described.

Synchronous vs. asynchronous channels. Synchronous channels are usually used when the sender of a request message demands an immediate response, because its consequent actions depend on it. The blocking of the sender until the response message arrives is therefore acceptable. In the running example of Section 3.1, if the buyer requests an offer from the seller, the response messages is crucial for the buyer, because the consequent decision about accepting or canceling the offer obviously depends on it.

In contrast, asynchronous channels are preferred over synchronous ones if the sender is not dependent on an immediate response. On one hand asynchronous communication does not block the sender until it finally receives the response, on the other side it gives the receiver the freedom to delay the computation of the incoming messages in favor of more urgent tasks that demand low latency. In the running example, a buyer would probably not need an immediate assurance that its `Order` message has been processed as long as the seller guarantees the availability of a communicated offer. Also a `Cancel` could be sent asynchronously by the buyer, as its future computation does not depend on the response of the seller.

The discussion shows that the decision about using synchronous or asynchronous communication should be (and in practice is) taken individually on interaction level rather than unified for the whole SOA application. More detailed discussions on the comparison and combined usage of synchronous vs. asynchronous SOA communication can be found in [EAA$^+$04, Sol06].

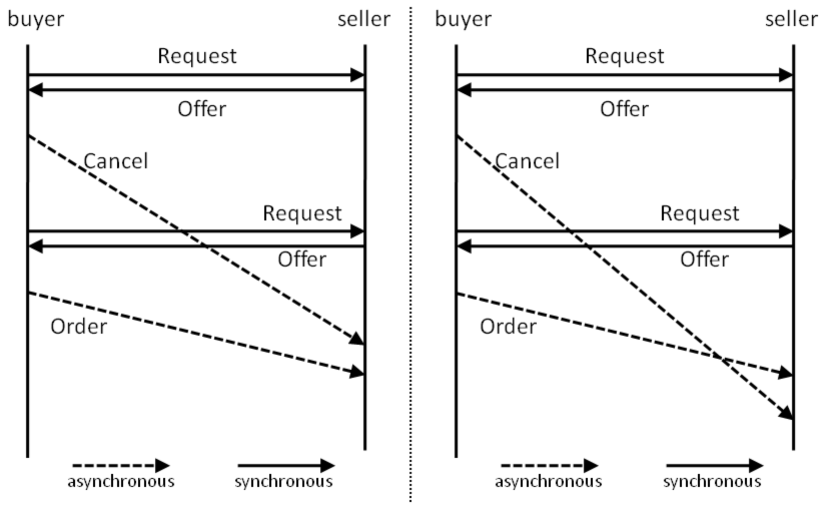


Figure 3.2: Example scenarios of asynchronous communication

Exactly Once vs. Exactly Once in Order. In the case of asynchronous communication, EO channels have the advantage compared to EOIO that their protocol overhead is much smaller and hence the latency is reduced a lot, especially in environments with bad transmission quality. However, EO communication should only be applied if the final result of computation is independent of the order in which the messages are received. In the running example, the order in which `Order` and `Cancel` messages arrive and are computed at the seller might be crucial. In the scenarios given in Figure 3.2 the buyer sends two requests during the interaction. While both depicted scenarios are valid for an asynchronous EO channel between buyer and seller, the scenario on the right cannot be observed in the case of an EOIO channel because the message racing of `Cancel` and `Order` is prevented.

In the described case, an EO channel could lead to problems because the seller might be unable to determine, which of the two previously sent offers has been canceled. However, if the `Cancel` and `Order` messages contain information about the corresponding offer, the seller will be able to take the right decision even if the messages arrive in changed order. Hence an EO channel with lower latency could be used in this case.

Message racing challenge. As argued in Section 2.3.4, the integration testing of SOA demands special attention as the classical approaches are not applicable “off the shelf”. Further, the particularities of SOA service integration have been explained above. In the following, the challenges of message racing for SOA integration testing will be discussed.

Three reasons can be identified, why it can be impossible to predict in which order messages are received in SOA systems:

- *Usage of an EO channel.* The effect of message racing on EO channels has already been explained above. In the running example of the communication between a buyer and a seller, the seller may send `Cancel` and `Order` messages on an EO channel which may lead to the changed receive order depicted on the right of Figure 3.2.

- *Multiple channels.* As discussed, synchronous communication is usually much faster than asynchronous. Therefore messages sent over different channels experience different latency and hence may be received in different order. In Figure 3.2 the second synchronous `Request` is sent later than the asynchronous `Cancel` but is received first.

- *Concurrent communication.* In many business scenarios, situations occur where both participants are allowed to send messages concurrently. In the running example this would be the case, if the buyer is allowed to send a `Cancel` before receiving the `Offer` from the seller. In this case, the buyer and the seller could act simultaneously and hence no prediction about the order of received messages could be made.

These three reasons for non-determinism in the SOA message delivery imply that robustness against message racing has to be considered in the system design and also checked during integration testing. It can even be argued that a system design using synchronous communication to ban message racing will only avoid the non-determinism to a certain extend as the effects of concurrent communication cannot be addressed. However this contradicts the paradigms of SOA and also affects its performance, as explained above. An integration testing approach for SOA therefore clearly has to address the faults related to message racing. Only then it will possible to build confidence in the correct functioning of the system.

3.2.2 SOA Choreography Modeling

As described in the previous subsection, quality control for SOA service integration has to examine and assure that the effects of message racing have been addressed in the development. Because of the limited access to code in SOA environments, integration should be handled in a model-driven way and therefore has to be accompanied with appropriate means of choreography modeling that has to fulfill the following objectives:

- *Comprehensibility.* Choreography modeling should support process integration experts, developers, and testers to get an unambiguous common picture of interaction of communicating service components. This is a crucial goal for software development with globally distributed development teams.

- *Verifiability.* Choreography models should be suitable for applying static verification techniques to discover inconsistencies in the design of the communication itself and in correlation with the behavioral descriptions of the involved service components. This ensures the discovery of problematic design decision before the actual implementation and hence avoids expensive corrections in later development stages.

- *Suitability for automatic test generation.* Choreography models should enable the derivation of integration tests using model-based testing techniques. After some initial effort for building test adapters, MBT promises an optimized test generation with a controlled test coverage and a high degree of automation.

In this respect, the presented protocol of the running example (see Figure 3.1) might be precise enough for a high-level business view of the intended process, however some semantical subtleties remain unclear. For example it has to be made explicit, whether this description specifies a subset of the intended behavior, which may allow additional transitions (e.g. observing a new `Request` message in the state `Reserved`), or a maximal allowed behavior, such that conforming implementations are allowed to leave out some functionality (e.g. the buyer might be allowed to skip sending a `Cancel` message in the case of rejecting an offer). Moreover, the semantics of message sending and receiving has to be clearly defined, based on the specific channel assumptions. In the running example two communication channels are used. Therefore, it might be observed that a `Cancel` message is delivered to the seller only after it received the `Request` message of a new negotiation process, even though these messages were sent in the opposite order. Consequently the protocol depicted in Figure 3.1 only applies if it is assumed that the transitions symbolize the sending of messages.

Requirements for a choreography language. Taking the initial thoughts and the general requirements from above as motivation, in the following specific requirements for a choreography language that supports the targeted model-driven approach are discussed.

- *Detailed message description.* Determinism is a prerequisite for most MBT approaches. In the context of SOA, the allowed communication sequences may vary depending on certain data values in the exchanged messages. In the running example this can be exemplified as follows. When presuming that the `Order` and `Cancel` messages have the same message type and can only be distinguished by the containing data (e.g. by a flag variable), the modeling language has to provide a way to express such constraints on the data instance level.

- *Infinite state space.* If for example a buyer asynchronously sends `Request` messages with the intention that each should be confirmed by a seller eventually, using either `Order` and `Cancel` messages, the choreography model has to distinguish between those communication states where confirmations are still expected and those where each `Request` has been answered. This cannot be specified by a finite automaton or a regular language, according to a classical result in formal languages theory. Instead unbounded variables (in the given case a counter that adds up the `Request` messages and subtracts the confirmations), implying an infinite state space, have to be available for choreography modeling.

- *Interaction termination.* For a choreography modeling approach that enables test generation an important prerequisite is to define those states, in which the communication is allowed to terminate, as they are implying states of data consistency. A termination state should not be defined as preventing any further communication (e.g. by disallowing outgoing transitions). When considering the negotiation between the buyer and the seller the `Start` state is a state where (e.g. after the receiving of a `Cancel` the communication might terminate or might be continued by the buyer. As discussed below, most of the current choreography languages are using an incompatible notion of termination, while the given example is in harmony classical notion of accepting/final states from finite automata theory.

- *Channel modeling.* Choreography modeling has to reflect the heterogeneous and distributed nature of SOA. In the running example distinct communication channel for the `Request` and `Cancel` message imply that the implementation of the seller component will have to be more robust, as it has to consider receiving a second `Request` message before a (deprecated) `Cancel` message arrives.

- *Explicit message send and receive.* Describing a send event together with its corresponding receive event as an atomic action restricts a choreography model significantly, as it prohibits to specify "send after receive" constraints. Considering an atomic modeling approach, for example the following "send after receive" constraint: 'the seller shall not be allowed to send an `Offer` after receiving a `Cancel` message' cannot be distinguished from the following stronger "send after send" constraint: 'the seller shall not be allowed to send an `Offer` after the buyer sent a `Cancel` message'. Therefore, for an unambiguous interpretation of the model the stronger "send after send" constraint would have to be applied. As this constraint can only be enforced, if both participants synchronize their communication, it contradicts the SOA paradigm of loosely coupled services and hence an explicit distinction between message send and receive is necessary.

- *Global and local views.* In order to design, verify and test a SOA application, next to a global model, also the envisioned local behavioral models of the participating components have to be given, as they specify the corresponding implementations. Keeping all these perspectives consistent is a major challenge of choreography modeling.

- *Pairwise choreographies.* Most interaction processes in SOA based systems span over multiple components. It can be noticed however that in the vast majority of cases no information is lost when projecting such multi-party choreography to pairwise choreographies, as the underlying processes are utilizing the components sequentially. Modeling pairwise instead of multi-party choreographies helps to reduce the modeling complexity.

- *State-based modeling.* Two major directions can be followed for choreography modeling: an activity-based or a state-based one. In the activity-based approach, the interactions between the parties and their ordering is the primary focus. In the state-based approach, the states of the choreography are modeled as first-class entities together with the interactions, which are then modeled as transitions between states. Since activity-based models may become cluttered by variables for bookkeeping of the choreography state and the message contents, a state-based approach is advocated.

Applicability of existing languages. Having derived concrete requirements for choreography languages that support a model-driven development approach including automated quality control, current choreography languages (described in Section 2.2.3) can be evaluated.

As mentioned, WS-CDL is a choreography language that targets the implementation level and builds on WSDL. It misses the explicit notion of termination, which is an important ingredient for test generation. Termination states in WS-CDL are denoted by the absence of outgoing transitions. Consequently the `Start` state from the running example could not be modeled as terminating in WS-CDL without leading to non-determinism. WS-CDL further only describes send events globally and hence does not reflect message racing directly.

The recent draft for BPMN 2.0 explicitly includes choreography modeling that should be understandable by business users and technical developers alike. In its current state, BPMN 2.0 has a too restrictive notion of termination. Even though (unlike WS-CDL) end states are defined explicitly, they are not allowed to contain outgoing transitions and thus lead to the same problems described above. Further it abstracts from channel information and treats message send and receive event as atomic unions. Moreover

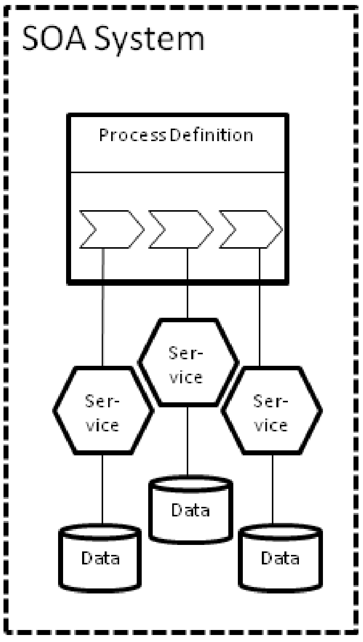


Figure 3.3: Decomposition of SOA

BPMN 2.0 has a large number of (non-trivial) modeling artifacts that make the modeling process complex and the learning curve steep.

BPEL4Chor and Let's Dance focus on high-level service interaction modeling in early design phases and target business analysts. Although the core of these languages is formal, guards and conditions can only be defined in natural language, which makes them inappropriate for verification or automatic testing approaches. Like WS-CDL, they do not have explicit notion of termination. Due to the assumed send viewpoint they are not able to reflect message racing. Additionally, Let's Dance even does not support the modeling of partner views.

3.3 SOA System State

The state of any software application is determined by its current processing step and by the stored data that the system is able to access. More precisely, the current processing step of a system describes the state in an abstract way and thus defines possible system responses in general. In the running example the buyer is initially only able to trigger a `Request`, later either a `Order` or `Cancel`. A process description deliberately abstracts from details like the internal data.

The internal data affects the state because it might constrain the possible system responses. In the running example, the decision to order or cancel might implicitly depend on the currently available funds in the buyers account. As the actual processing step of a system is represented by state variables, the state of a software application is in general determined by its current data assignment.

To make software testing productive, the ability to observe the systems data (and hence the system state) at testing time is regarded as a must. Without such information the reasoning about whether the system responded correctly to a given test stimulus seems to be impossible. As illustrated in Figure 3.3 decomposition of components, as demanded by the SOA concept, however leads to a decomposition of data, too. Consequently the state of a SOA application is hardly observable any more. Verifying SOA based systems therefore forces test concepts to consider unobservable system states, while still providing the means of reasoning about test results.

Another important paradigm in the theory of testing is the reproducibility of tests. For a stateless system or a system with a built-in method to return to the initial state (e.g. a calculator) this requirement can be met easily. As SOA applications are highly complex in functionality and data, such a requirement on testing usually cannot be met. 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. Further especially enterprise software usually intentionally prevents the reset of data to meet legal requirements. However, as each test run changes the system state, a rerun of the same test case will most likely find the system in a different condition.

State changes of software systems do not depend on actions alone but are connected to input data (transactional data) and system data (master data and state variables). When providing test data for complex systems such as SOA applications, various challenges have to be considered. These are illustrated in Figure 3.4.

Concerning the test input, two main categories for data constraints have to be satisfied. First of all, the test input data has to be compatible to the test case it is applied to (e.g. conforming to the input format of the test steps). Secondly, the test input data often has to reference available and appropriate system data. In Figure 3.4 these two categories are referred to as *System Data Dependent Constraints* and *System Data Independent Constraints*.

The other challenges depicted in Figure 3.4 deal with the provision of the data that has to be present inside the system during test execution. Again, these challenges challenges can be divided in two parts. The first, namely *System Data Supply*, contains the issues of inserting consistent data into the system. The second, namely *System Data Stability*, comprises the challenges of keeping the system data stable despite test execution.

In the following subsections, the challenges for handling system data and input data are described in more detail.

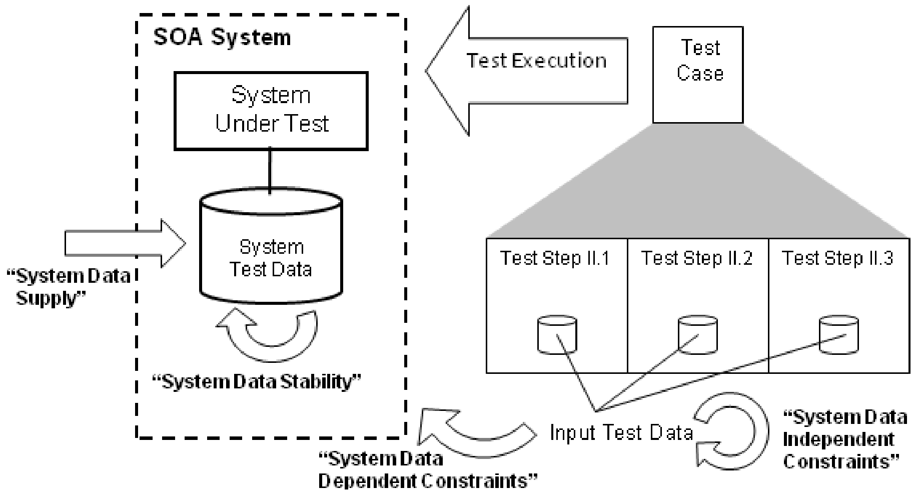


Figure 3.4: Challenges related to SOA test data

3.3.1 System Data Challenges

System data is a necessary ingredient for testing since internal data is the base of any data intensive system like ERP and will most likely be processed during any execution. Two special subjects regarding system data for regression testing have to be considered: *system data supply* and *system data stability*. In the following, detailed descriptions for each are given.

System Data Supply. In ERP systems, providing master data as system data for later interactions is an important user scenario. For testing, providing an initial system data could be either part of each test relying on system data or could be done during a general testing preparation, where the data is either inserted directly or via the developed system.

Provision per test case. In the first case, all tests should be able to run on empty systems, initializing and storing needed system data in the preamble phase. Such approach is unfortunately infeasible for ERP testing. Even the small example above is suitable to shows the practical weaknesses. From the buyers perspective, to order some goods from a seller, contact information as well as billing information have to be entered to the system. The billing information itself demands other information like the business unit to which it is assigned. This unit needs information about its manager and the company it belongs to, among others. Further iterations lead to the creation of a vast amount of master data even for a simple test run. Other processes (e.g. creating a report about all sent sales orders) additionally need saved transactional data inside the system. There are usually numerous internal data dependencies such that most test cases would be forced to set up a large amount of master data in the preamble phase in order to be executable.

Admitting that the effort of providing system data in test case preambles is unmanageable, ERP testing demands the insertion of common test data to the system during the testing preparation, which then can be used during the test execution.

Direct provision. The first possible test data preparation approach is to write the data directly into the database, but this is difficult, as it demands to either manually or automatically enforce system data consistency during this process. For the manual task, the complexity of data relations is too high, whereas for the automated task, consistent data insertion would mean to re-implement the expensive system data constraint checking and solving mechanisms that is part of the system under test.

System-based provision. A second more realistic solution is to fill the empty system with common test data by using the application and hence the implemented mechanisms enforcing system data consistency. The procedure itself therefore can be seen as a set of fundamental test cases (that can consequently be modeled and generated using MBT). Even though this approach seems to be much more feasible, problems still arise. First, using an untested system to generate a master data stack is error prone and hence the quality of master data will be poor due to faulty insertion by the system itself. Furthermore, in early development stages mechanisms needed for data insertion might not be fully implemented.

Despite the mentioned problems, providing common test data as described in the system-based test preparation approach is the most practicable solution so far.

System Data Stability. In order to enable efficient testing, the common test data in the system has to be kept unchanged. This is not only connected to the requirement of regression testing: 'always execute a test case on the same system state', but also a prerequisite for executing multiple test cases that depend on the agreed system state. If there is a common set of system test data which, as argued above, should be provided initially, it should not be changed by any test. However, this is a very expensive requirement for testing in ERP systems, because the changing of system data is part of the common ERP functionality. System data will even be consumed by some tests. In the given example the buyer is allocating and blocking some budget each time a sales order is sent. Consequently, the system data will be changed and eventually the funds are not sufficient any more for further test activities. Other tests might alter common master data unintentionally due to implementation faults in either the SUT or the test case itself.

As explained, altered common test data proves to be problematic for other test cases depending on them. Investigations on whether a failed test was caused by a faulty implementation or altered master data is hard to conduct as a failed test even might occur randomly e.g. depending on the execution order of test cases. Apart from the described technical difficulties, test cases implying a fault because of altered system data also negatively affect the tester's motivation.

Read-only data access. System data access rules that are preventing the alteration of common test data may be a solution, but enforcing write protections during test execution will also cause a difference in behavior of the SUT compared to the delivered system. Hence positive test results are not guaranteeing a correct computation in the delivered system any more.

Late binding. Another possibility could be to bind the concrete system data to abstract test cases during runtime. In this case, rules (e.g. defined in OCL) could be generated together with the test cases, allowing the test execution system to search for compliant system data and to assign it just before or even during test execution. To determine the current system data state and binding suitable data to the abstract test cases is however very complex and time intensive in practice. Furthermore, observing the whole system state is (if ever) not possible until very late stages, thus preventing such a testing strategy during most of the development period.

Data cloning. A third promising strategy to supply test data stability could be the cloning of master data tables in an initial system state (prior to test execution) to ease and speed up regular data resets, e.g. once a week. Shorter periods are usually impracticable because the copying costs time in which development and testing has to pause. Further, it usually still demands additional manual work. However directly copying master data will always result in the loss of stored transactional data, as former references and relations will be destroyed. Also structural changes of the master data (e.g. adding a field for the gender at the personal data), which are carried out frequently during development phases can result in the invalidation of the master data clone. Automatic reset and re-provision of system data as described in the previous section then is the only way to solve the problem. Nevertheless the practice shows that, even when it is nearly fully automated, system data reset demands about half a week of downtime at SAP. Therefore the usage is very limited while test data stability remains a difficult problem.

3.3.2 Input Data Challenges

Until now, only the test data inside the system has been discussed, but in order to test applications also data from the outside has to be provided for the test runs. For ERP systems, this input data will be mostly transactional but in order to test master data modification both transactional and master data might have to be added to test cases. Similar to system data, the general feature that makes it hard to deal with input data, is the complexity of its associated constraints.

In the context of ERP system testing, the input data relations can be

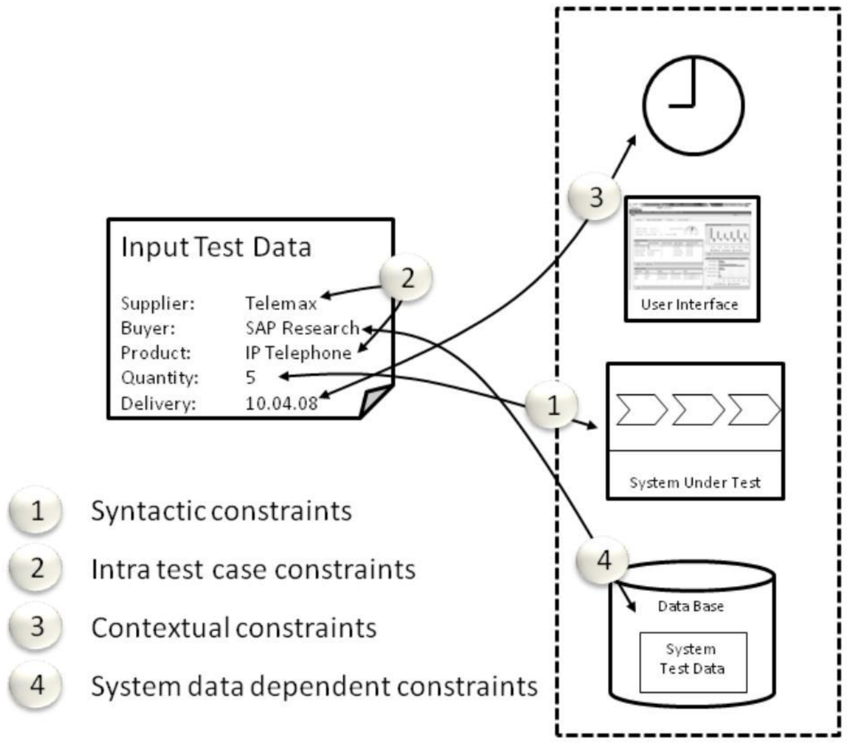


Figure 3.5: Illustration of input data constraints in an ERP system

classified in two groups: system data independent constraints describing the correlations of input data inside a test case, which are unrelated to the system data and input data constraints describing the relation between system and input data. These constraints are illustrated in Figure 3.5 and discussed in the following.

System Data Independent Constraints. System data independent constraints can be divided into the following categories:

- *Syntactical input data constraints.* Every ERP system has syntactical constraints on input data, concerning for instance data types and ranges. In the test case in Figure 3.5 a positive system response depends on the usage of the correct integer range for the quantity (i.e. positive values). In contrast a value outside the range should result in a system error message.

- *Intra test case constraints.* Not only in MBT but in every black box testing approach, test cases describe an interaction sequence with the SUT using only interfaces, which are accessible from the outside. The correct reaction of the system might not only depend on syntactically correct input values, but also on the semantical relation between the data used for different steps of the test sequence. In the example from Figure 3 the system should either react with a success notification or an error message depending on the fact whether the supplier Telemax is able to provide IP Telephones.

- *Contextual input data constraints.* Also the application context might enforce constraints on the test data. In Figure 3.5 the validity of the used delivery data depends on the current system time. Other contextual constraints are for example input data restrictions depending on user roles or business configurations.

System Data Dependent Constraints. The more complicated constraints are those relating input data to system data. Depending on the system data observability, it might even be impossible to determine in advance whether a certain input value should trigger a positive or negative system response, and hence the satisfaction of constraints might become nondeterministic. In the test case of Figure 3.5 for each identifier (Supplier, Buyer, Product) a valid master data entry has to be present inside the system data to be able to successfully process the sales order. However if it is not possible to observe system data during test execution, an unambiguous test oracle for the system response is impossible to provide. Also the absence of specific system data belongs to this category as master data inside the system often has the restriction to be unique. Hence input of already existing master data might result in different system behavior than the input of unique data. This issue especially becomes prominent in regression tests, where the provision of unique input test data can be problematic.

3.4 Summary

In this chapter two open fields of research in the domain of SOA have been described. The first topic (explained in Section 3.2) deals with the challenges of defining service communication protocols that allow the application of model-based testing. The second one (described in Section 3.3) explains the challenges connected to the provision of test data.

While both of these topics have to be addressed in order to enable a model-driven engineering and especially a fully automated testing approach, it has to be conceded that the necessary effort to address both would exceed the scope of one dissertation. Further, the generation of test cases (abstracting from the necessary test data) is a prerequisite for automatic test data generation. Therefore the aim of the conducted work has been limited to elaborate solutions for a model-driven service integration, including the test case generation for service choreographies, but excluding the automated provision of test data. In other words, the test data provision challenges described in Section 3.3 have not been addressed in general, but instead the described current approaches have been facilitated or enhanced.

Chapter 4

General Approach

As shown in the previous chapter, currently no choreography language is available that is fulfilling the requirements of a model-driven quality control for SOA service composition. Consequently, there is a lacking tool support for an unambiguous modeling as well as for the utilization of the content for model-based testing. This current situation can only be tackled by improving the current SOA development process.

In this thesis, a development process for SOA applications is advocated that combines model-driven and agile development. It will serve as a foundational framework to which the main research results of the dissertation are contributing.

Section 4.1 introduces the envisioned approach. Sections 4.2 and 4.3 discuss important considerations of the incorporated modeling and testing phases. Finally, in Section 4.4 the given approach is summarized and related to the problem statement of Section 3.2.

4.1 Overview

As in every customer oriented scenario, a SOA development process should start with the definition of user and market requirements. For Enterprise SOA applications especially functional requirements are described by business processes, which have to be supported. As illustrated in Figure 4.1, 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 (MDD) and 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.

Model-driven Development. By performing the design steps of MDD, the initial requirements are gradually refined into development models. These

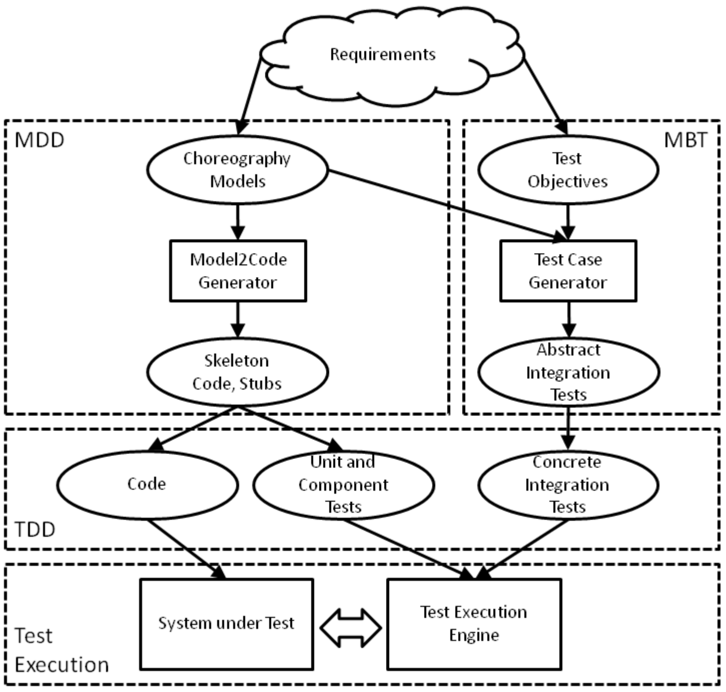


Figure 4.1: Envisioned development process for SOA

comprise of structural models, identifying and connecting the service components, and behavioral models of the business process flow. In order to avoid ambiguity, the description of the desired service communication should be created using a formal specification. Concrete considerations regarding a suitable formal specification are discussed in Section 4.2.

Having an unambiguous definition of the communication protocol should allow applying automatic model verification and validation techniques. For example the absence of deadlocks, livelocks, unconsumable messages or local enforceability can be proven at this early development stage. This effectively ensures that an infeasible design decision is identified and corrected before implementation has started and a necessary correction becomes costly.

The validation and verification activities described above, are directed towards incrementally leading to a good and adequate model. According to the experience gathered at SAP, modeling starts with a first sketch of the communication behavior. By using a simulation tool, discrepancies between the intuitive understanding and the modeled behavior can be uncovered and hence corrected within the model. Providing systematic checks, e.g. by utilizing a model checker or theorem prover, flaws in the model can be detected systematically. In this case, the modeler should be guided, e.g. by providing a trace to the situation in which the error occurred. This enables the modeler to adjust or refine the model accordingly.

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 (e.g. by [TPT09, Uhl08, FR07, HT06]) for the industrial application of MDD concepts for code generation, such as:

- Lacking tool support for model-level debugging.
- Lacking user expertise for required formalisms.
- 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. Therefore, as illustrated in Figure 4.1, the development models are instead 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. Taking advantage of the unambiguity of the development models, the programming can be carried out in parallel by autonomous teams. In addition to the general advantage of shortening the development process by parallelization, it further allows companies to work with distributed development teams.

Agile development techniques like Scrum [SB01], Test-Driven Development (TDD) [AJ02], and Extreme Programming [BA04] are more and more regarded as key practices of software engineering for systems of extreme size and complexity [Coc07]. While waterfall-like development process models bear immense risks, as no subsequent process phase can correct all errors from the phases executed before, iterative development models with close feedback loops help to mitigate these risks. Having developers and stakeholders to plan and review development efforts frequently creates transparency and allows for correcting implementation mistakes early in the process.

For example, SAP is using Scrum in combination with model-driven development since 2004, while TDD was recently introduced to its technology development area. The about 80 Scrum projects finished before end of 2007 showed significantly better results compared to those using processes inspired by the waterfall and V-models. Also, first pilots for TDD delivered a much lower defect rate, more comprehensible and thus more maintainable code.

In more detail, an agile development process will continue as follows. After automatically generating code stubs from the structural models, developers are starting to create tests for the functions they are going to implement. In the context of SAP, these tests mainly cover the *unit testing* and *service component testing* layers of the SOA testing stack, as introduced in

Section 2.3.3. In this way, the developers are able to validate their own code automatically by running these tests. Provided that the testing is successful, refactoring takes place to increase code readability, followed by another round of testing to make sure the refactoring changes did not affect the behavior. These process steps are repeated recursively for each bit of added functionality.

Model-based Testing. In the envisioned process, integration testing is carried out in parallel to the development of the software components. A recent study [Mur09] 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.

As discussed in Section 2.3.4, MBT approaches are able 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.

As explained, the main objective of using MBT is to facilitate state-of-the-art test generation techniques for the special purpose of overcoming the SOA integration testing challenges described in Section 3.2. However, the utilization of MBT to a specific domain (i.e. service integration testing) demands some upfront considerations, like the definition of test objectives. In Section 4.3, these considerations are described.

Test Execution. Similar to 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 hinder industrial application of fully automatic test case generation. Additionally, especially the test data challenges discussed in Section 3.3 have to be solved. As explained above, addressing these challenges have been excluded from the scope of the dissertation.

Therefore, it will be the responsibility of the testers to provide appropriate test data and hence the approach will leverage their experience. 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. It is also quite common that reference test data is

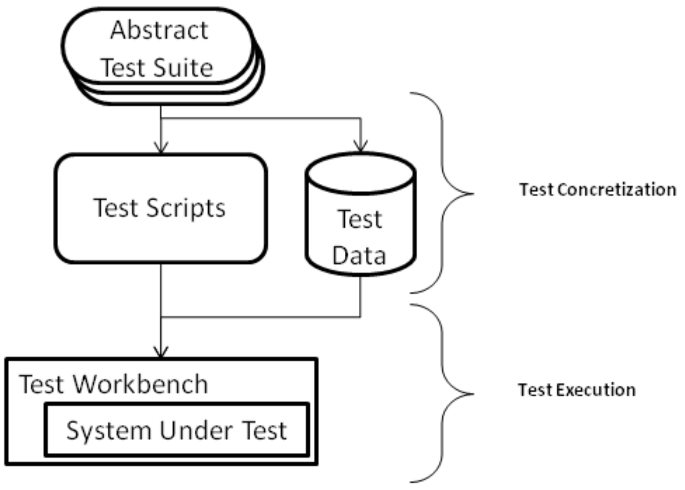


Figure 4.2: Overview of the test concretization and execution

provided by customers or internal departments, as additional information to the requirement specification. 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.

A simplified illustration of the activities related to the test concretization and execution is given in Figure 4.2. According to the nomenclature of [UL07, ch. 8], a mixed approach is used for the test concretization. The manual refinement of the abstract test cases follows the keyword-driven testing principles. Keyword-driven testing (or action-word testing) [BJPW01] 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 [UL07].

In the context of this dissertation, the used test language builds upon SAP's eCATT test script language [HLT07] and was designed to address the requirements of integration testing at a higher level of abstraction. It contains constructs for creating and modifying local business objects, triggering the sending of messages between the business components via the available enterprise services, and checking the values of internal component states against the expected values, in order to decide the failure or success of a test.

To minimize the manual effort for the test concretization, the generated abstract test cases are transformed in a modular way. Each test step is transformed into a separate reusable script, while for each test case a master script is generated that calls the reusable scripts in the appropriate order. In this way, a high reuse is enforced that results in less effort and enables parallelization of the manual work, which is a big advantage for integration

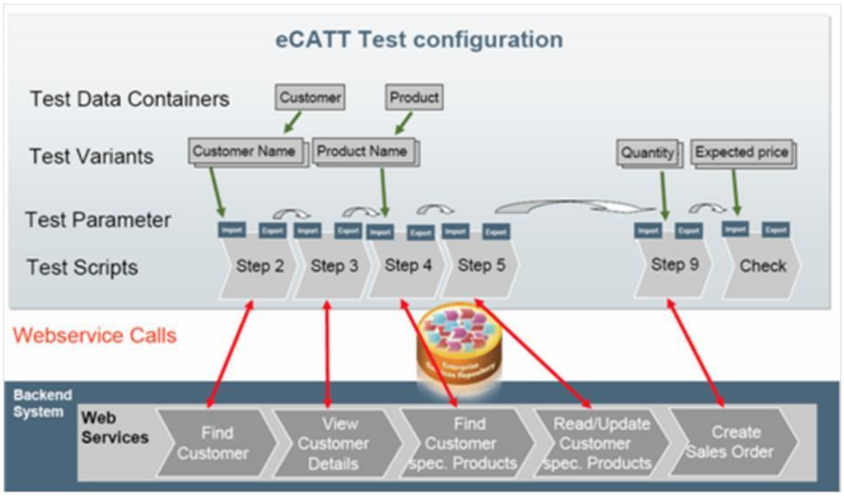


Figure 4.3: Service integration testing implemented using SAPs eCATT framework

testing with different development areas concerned.

The subsequent test execution activities are well understood in practice and thus trigger little research interest. Mature test management systems, supporting the whole testing process starting from the test planning, test execution until the final test reporting are available for most software domains [SG08].

In the context of this thesis, the test execution environment is provided by the SAP Test Workbench and SAP Solution Manager. 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 4.3 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.

4.2 Choreography Modeling

The challenges of service integration, described in Section 3.2, are manifold. Currently no choreography language that is suitable to support model-driven quality control is available. Consequently, there also is a lacking tool support for an unambiguous modeling. Chapter 5 describes the Message Choreography Modeling (MCM) language that has been developed to address this lack. MCM has been created based on the requirements for choreography modeling. This section discusses some fundamental design decisions that ultimately guided the development of MCM.

First of all, the usage of domain-specific languages in SOA development and especially in choreography modeling is motivated. Afterwards, possible viewpoints of choreography models are described. Finally, the implications of using a modeling language for both model verification and test generation are discussed.

4.2.1 Domain-specific Modeling

The key message of Section 3.2.2 was that currently available choreography modeling languages are not fulfilling the requirements of the described SOA development and especially service integration testing approach based on choreography models.

It can be argued that holistic modeling approaches like UML or Petri-nets are able to match the introduced requirements. In case of UML the richness and partial ambiguity however limits its application. Current scientific approaches (cf. [SGS04, KKKR05]) further do not consider asynchronous communication between service components.

Also, Petri-nets do not support service composition modeling as such. Therefore, their semantics have to be refined in order to be applicable. Again, current approaches to choreography modeling (cf. [ZCCK04]) do not consider asynchronous communication, but could be enhanced in the future.

The above mentioned shortcomings of modeling approaches, based on general concepts like communicating state machines or Petri-nets, however does not automatically imply the utilization of a domain-specific modeling approach. Combined with the fact that current DSLs for choreography modeling are also inapplicable, it shows that the gathered requirements have not been considered as a whole so far.

In general, modeling approaches like communicating state machines are feasible for choreography modeling. However, DSLs fundamentally raise the level of abstraction, while at the same time narrowing down the design space significantly to domain-specific concepts. The result of this specificity leads to a reduction of complexity and an increase in productivity of $5 - 10$ times as proven by various case studies [WL99, Saf07, KT00].

Further, domain-specific modeling is not limited to domain-specific tools. Instead, by providing model transformations, general solutions and standard tools can be integrated. In fact, this is the chosen approach for the dissertation (see Section 5.4).

4.2.2 Choreography Viewpoints

In Section 3.2.2, it has been explained that unambiguity of choreography models depends on a clear definition of the semantics regarding message send and receive. When specifying the communication between service components, it is therefore important to specify the viewpoint of the assumed observer, describing it. Various observation viewpoints can be defined. The choice of a suitable viewpoint for choreography modeling depends on the intended use of the model. As described in Section 3.2.2, the definition of a choreography viewpoint is usually neglected in the literature. Most approaches implicitly assume a global send viewpoint (i.e. the choreography model describes all acceptable observations of message send events).

The example in Section 3.1 a protocol of the communication between a buyer and a seller was given. By combining it with the given channel information, Figure 4.4 depicts all possible sequences of send and receive events of the communication that are no longer than 12 steps. Events are defined by a string containing the name of the owning component (i.e. buyer or seller), a symbol indicating the even type (! for send event, ? for receive event) and the associated message name. Consequently, the initial event *Buyer*!*Request*can be described by "the buyer is sending a `Request` message". Some of the events are surrounded by a double line. When these events are observed, the communication between seller and buyer is allowed to terminate, leaving both of them in a synchronized and consistent state, while the choreography model is in an accepting state and no message is pending.

For the definition of viewpoints, a non-participating observer of a service communication with the capability to observe all message send and receive events on all channels without delays will be assumed. In principle, viewpoints are models representing the observed event sequences.

Global send viewpoint A choreography model with a global send viewpoint contains a description of all sequences of send events that the above defined observer is able to monitor. Consequently, Figure 3.1 could be seen as the global send viewpoint model of the running example, by interpreting the transition labels as send events. As mentioned, most choreography languages are based on a global send viewpoint.

Global receive viewpoint The global receive viewpoint is an alternative to the above described send viewpoint. In this case, the choreography model describes all sequences of receive events that the above introduced observer is able to monitor.

A receive viewpoint for the running example is depicted in Figure 4.5. Compared to the send viewpoint, four additional transitions occur. Three of them are labeled as receive events of deprecated `Cancel` messages. Deprecated in this context means that due to message racing, the seller already receives another `Request` message before the `Cancel` message of the previous request is received. In these cases the delayed `cancel` messages are assumed to be deprecated. Consequently, "normal" and "deprecated" `Cancel` mes-

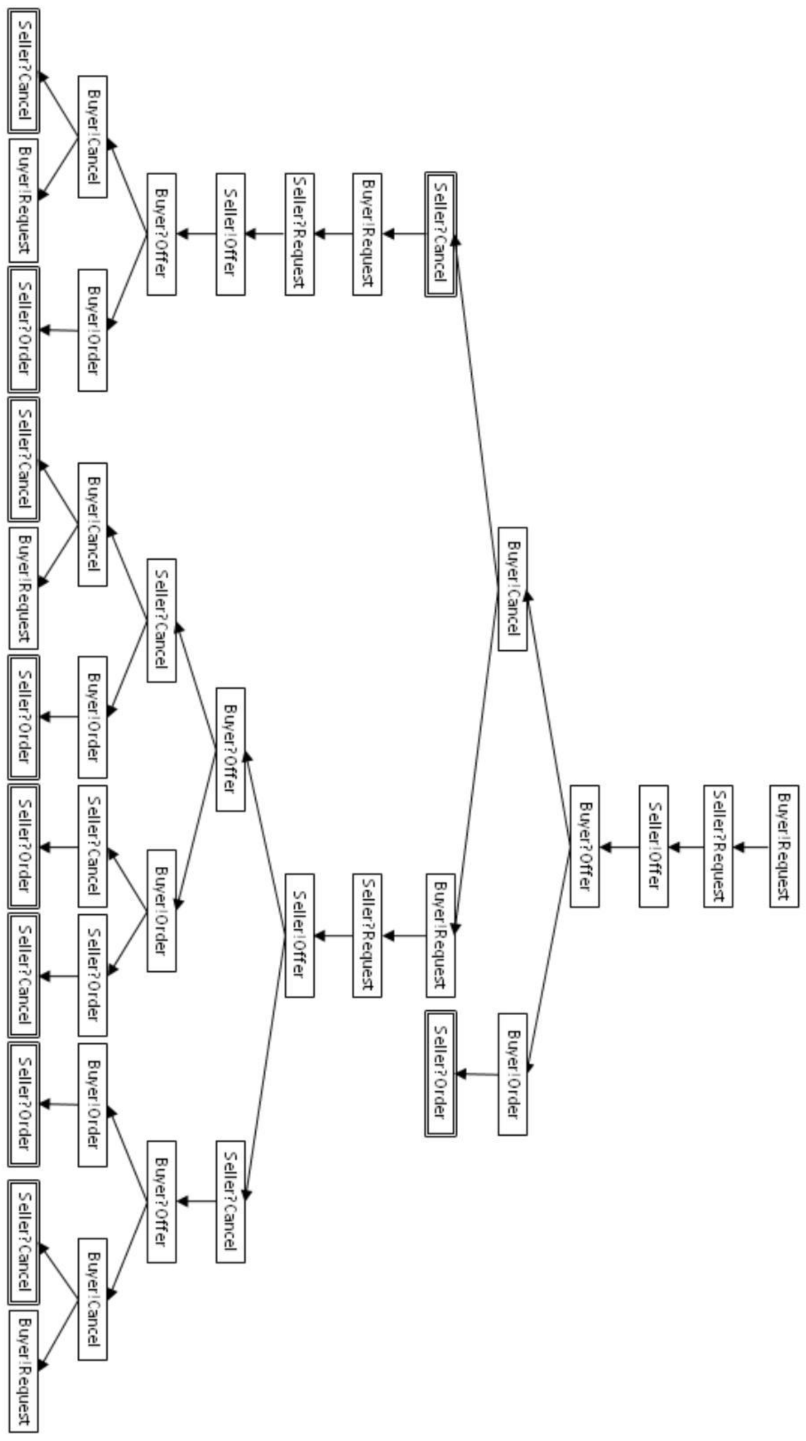


Figure 4.4: Message event sequences for the running example

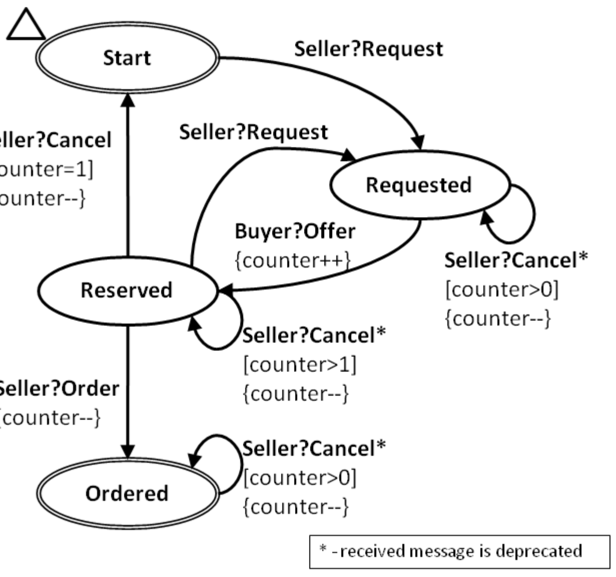


Figure 4.5: The global receive viewpoint for the running example

sages are technically identical. Further, an additional transition now connects the `Reserved` and the `Requested` state, because in cases of delayed `Cancel` messages, the buyer will receive another `Request` instead.

For defining the receive viewpoint of the running example, it is also necessary to add constraints and side effects to the choreography model, because the potentially infinite number of delayed `Cancel` messages[1] prohibit the use of finite state machine notations. In Figure 4.5 guards are defined inside [ ], side effects inside { } using pseudo code. Further, it is assumed that the referenced counter variable is an unbounded integer with 0 as initial value. The counter is used to keep track of the number of pending buyer response messages (either `Order` or `Cancel`), and hence is increased each time an `Offer` is received. Note that the usage of variables, guards and side effects is a general requirement for choreography modeling (see Section 3.2.2) and not restricted to receive viewpoints.

Global viewpoint discussions As explained in Section 3.2.1 an aim of service integration testing is to check the proper handling of message racing. The above example shows that a global send viewpoint is not reflecting message racing as deprecated `Cancel` messages are not visible. In contrast, a global receive viewpoint describes all possible sequences of receive events and hence explicitly distinguishes sequences with preserved message order

[1]This might happen, if the buyer constantly cancels the offers of the seller and afterwards sends new synchronous requests fast enough to block the seller from processing the asynchronous cancellations.

from those including message racing.

The introduced send and receive viewpoint models can be interpreted as describing projections of possible message event sequences in a service communication to sequences of either send or receive events only. Hence, another possible global viewpoint could be defined as including the original sequences of message send and receive events. This viewpoint also explicitly distinguishes event sequences including message racing. However, even for the relatively simple running example, the model gets cluttered because of the additional states that have to be included between send and receive events. Apart from decreasing readability, the higher complexity also negatively affects the application of MBT-tools in the envisioned service integration testing. Therefore, a global receive viewpoint will be utilized for MCM, as described in Section 5.3.

Local viewpoints Apart from global viewpoints, service communications can also be seen from the local perspective of the involved components. In this case, the choreography model describes all sequences of send and receive events that the above described observer is able to monitor for a specific component only.

The availability of such a local viewpoint is quite important in practice. While a global viewpoint is a concise description of the interaction protocol, local viewpoints describe a component's communication behavior and hence are an excellent starting point for the implementation and verification (i.e. service component testing, according to the definition from Section 2.3.3) of the service components. Therefore, incorporating both local and global viewpoints in one choreography modeling approach is considered an advantage and is included in choreography modeling standards such as WS-CDL [W3C04b] or BPMN [OMG08a]. Consequently, also MCM will incorporate local viewpoint models.

Figure 4.6 depicts a set of local viewpoint models for the buyer and seller component of the running example. It can be noticed that the structure of the seller model has an equal structure compared to the global viewpoint model depicted in Figure 3.1. In fact, even the constraints on the transitions are equal. Like the global receive viewpoint model, the model of the seller component incorporates four extra receive events (compared to the global send viewpoint), because it has to consider the potential message racing between `Cancel` and `Request` messages that the buyer is sending on different channels.

The buyer component model is a structural copy of the global receive viewpoint, too. However, some of the transitions are erased, such that the buyer is allowed to send `Cancel` messages in the state `Reserved` and `Request` messages in the state `Start` only. Not deleting the mentioned send transitions in the buyer would result in the same receive viewpoint model. How-

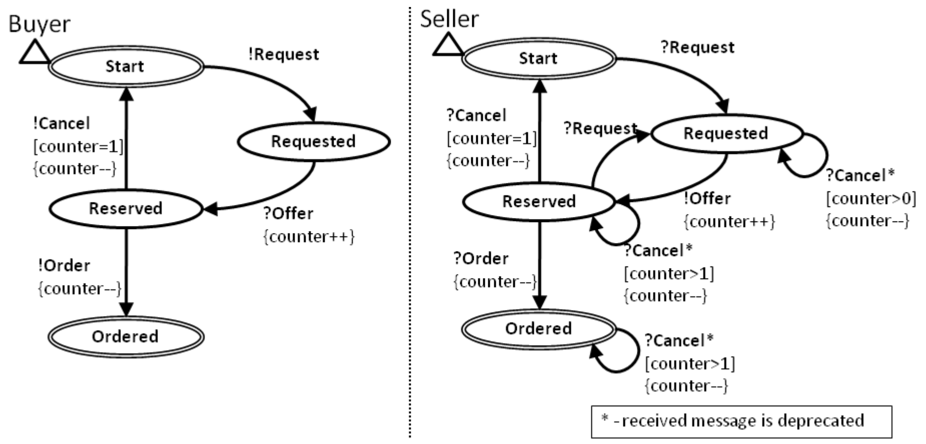


Figure 4.6: Local viewpoint models for the running example

ever, considering the description of the running example this seems to be reasonable.

It can be noticed that the guards and side effects of the buyer do not constrain the behavior and so its local viewpoint model appears to be a structural copy of the global send viewpoint. This is caused by the fact that the seller component is only reacting synchronously, while the buyer is initiating the conversation, but this is not a general case.

4.2.3 Consistency

As explained in the last section, a suitable choreography modeling language has to provide the means for describing the local components and a global receive viewpoint. Keeping the local and global perspectives consistent is a major challenge of choreography modeling. Considering the described development approach, also the consistency between the requirements of the user and the choreography models, and the consistency between the choreography and the implementation of the service components has to be provided. As depicted in Figure 4.7, the main goal of ensuring consistency between these abstraction layers is of course the enforcement of consistency between the requirements and the implementation itself.

There are various ways to define the consistency relation of models, describing concurrent interactions, based on concurrency semantics (e.g. simulation [Par81] or trace semantics [Hoa78]). A good introduction, including a classification of the different methods can be found in [vG90]. According to this classification, defining consistency based on traces means to compare sequences of observations (in contrast to the mapping of modeling elements, as demanded by simulation) and thus seems to be highly suitable for the given purpose of comparing different definitions of service communication.

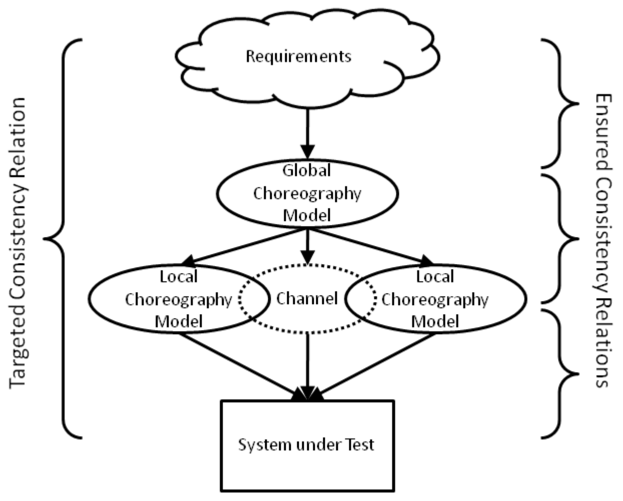


Figure 4.7: Consistency relations in Choreography Modeling

Consequently, in the following the discussion on consistency will be based on trace semantics.

Consistency between Requirements and Implementation Requirements are descriptions of a systems behavior that are usually not formalized in practice. The implementation of a system represents a concrete, executable instance that fulfills the requirements. Judging, whether an implementation fulfills the requirements is commonly done by testing. Utilizing the requirements in a testing process implies that all the specified behavior is reflected in the implementation. From a theoretical perspective, all abstractly defined traces of the requirements have to be realized by the system. In terms of trace semantics, the implementation has to include all traces of the requirements.

The envisioned development approach is further utilizing formal verification to enforce certain properties (e.g. absence of deadlocks) on the implementation. The common approach is separated into two parts: proving the desired properties on a high level of abstraction and proving refinement [vGG90] for the lower abstraction layers, including the implementation. In terms of trace semantics, the requirements have to specify (include) all traces of the implementation.

Jointly instrumenting formal verification and MBT techniques therefore would imply traces equivalence between the requirements and the system under test and hence also pairwise trace equivalence for all abstraction layers in between [vGG90] (see Figure 4.7). In practice, this obviously has to be restricted to the requirement specific abstraction domain, in this case component communication. In the following an overview of the applied methods for ensuring consistency are given. In Section 5.5 their realization will be described in more detail.

Consistency between Requirements and Choreography Model As described in Section 4.1, requirements are not formalized in practice and hence applying formal methods and MBT on this level is impossible. Instead, the choreography models are the envisioned artifacts to enable these quality assuring activities. Therefore, the consistency enforcement between choreography models and requirements is a manual task. However, industrial model-based approaches map requirements to behavioral model elements (e.g. tagging states with requirement IDs) in order to automatically analyze requirement coverage [UL07]. For MCM, a simulation tool is provided that enables to check whether the choreography model captures what has been informally described in requirements.

Consistency between Global and Local Viewpoints For enforcing consistency between global and local viewpoints two possible solutions exist [DW07]: a generative approach where the local views are generated from the global ones, or a checking approach where global and local models are created separately and then verified whether they are consistent with each other. While the first ensures that global and local views are always consistent, it makes changes to the local models considerably more difficult, since these would be overridden by re-generation from the global model. The latter approach allows for such “asymmetric” changes, but requires manual effort to update the global view when changes to the local models are made. For MCM, a mixed approach is utilized. It will be described in Section 5.5.

Consistency between Choreography Model and Implementation For each service component, involved in the choreography, more detailed development artifacts such as implementation code or other models may exist. In comparison to the local views of choreographies, they also take into account behavior that is not related to the communication. Such components are usually described with the help of models, which specify contained attributes (and their types) and state transition diagrams, which describe the effect of actions (such as service calls) on the internal state of components. As described, utilizing MBT aims at ensuring trace inclusion from choreography models to the implementation. How trace inclusion in the opposite direction is enforced for MCM, will be described in Section 5.5.

4.3 Model-based Integration Testing

According to the general approach given in Section 4.1, after having obtained a formal representation of a service choreography, MBT techniques can be deployed to derive test suites for integration testing. In the following, some necessary decisions are described that have to be made in order to adapt MBT to the specific requirements of service integration testing, starting with the identification of the targeted errors, followed by a discussion on suitable coverage criteria and the definition of industrial requirements for the test design.

4.3.1 Error assumption

According to [Wey98], integration testing can be seen as testing of an assembly of components that were already individually tested. As described in Section 2.3.3, this includes the testing of the functional units that a service components consists of (unit testing), as well as the testing of the interfaces that a service component exposes (service component testing). Service integration testing further does not aim to uncover faults in the messaging infrastructure, but assumes that the defined reliable messaging properties are realized correctly.

As explained above, consistency between requirements and the SUT demands that every trace in a component's local viewpoint model has to be executable by the implemented service component. Assuming the functional correctness of the participating service components and the infrastructure, the main error sources in the communication of service components can be defined in relation to the protocol fault classification of [Kön03]:

- *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.

4.3.2 Coverage

To enable effective model-based service integration testing, the targeted coverage of the choreography model has to be chosen carefully. Testing each component individually using local test coverage criteria is of course not sufficient in determining whether two components are able to operate with each other under the agreed circumstances. Only the application of a global test concept can provide that.

In [3], possible coverage criteria for state-based service choreography descriptions have been discussed that can be used to drive service integration testing. Also some hints were given on how to choose them accordingly, depending on effort and fault assumptions. For the envisioned testing approach, transition coverage has been identified as best fit, because it already uncovers a significant amount of integration faults with relatively small efforts [UL07]. For example in the approach of [ABR$^+$07], transition coverage of a global communication model was able to detect about 90% of integration related faults. This is also backed by the analysis in [3], showing that apart from sneak paths, illegal message faults and trap doors[2], transition coverage is able to expose the mentioned communication related faults to a great extent.

According to the discussion in Section 3.2.1, integration tests should ensure that message racing has been treated correctly during software development. By covering all transitions of a global receive viewpoint of the choreography model, it can be secured that a message will be handled by its receiving component in any allowed situation.

For the automatic test generation from a choreography model, the composed system that incorporates information from the two local viewpoints and the channel model (relating the send and receive events of the service components) could be used. However, various cases studies (cf. [CSH03]) show that state space explosion, as it might be introduced through the unbounded channel model, is a major stumbling block when applying automatic test generation to industrial settings.

Transferring this knowledge to service integration testing, it becomes clear that utilizing the composed system for the test generation might become problematic. On the other hand, utilizing the global receive viewpoint for the test generation offers a reduced complexity. However, it demands an investigation on the relations of the global and local test coverage. The result is that none of them implies the other, as explained below.

Global transition coverage. Not surprisingly, transition coverage of the global receive viewpoint does not guarantee transition coverage of the local viewpoint models. A counterexample is given in Figure 4.8 where service

[2]All these faults are subject to negative testing.

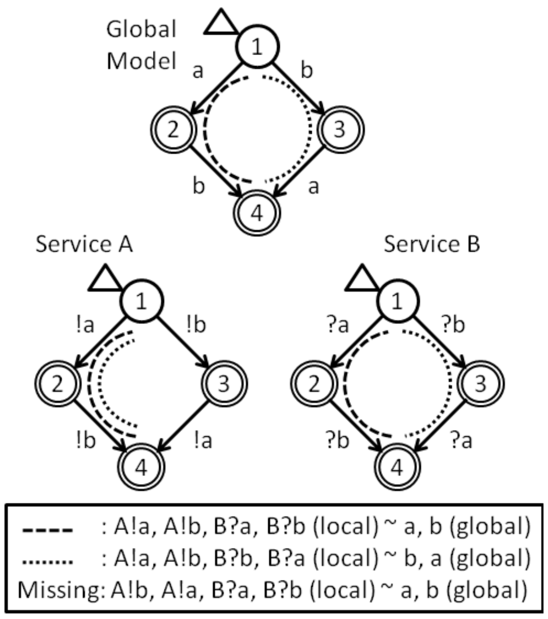

Global Model
1
a
b
2
3
b
a
4
Service A
!a
!b
!b
!a
Service B
?a
?b
?b
?a
---- : A!a, A!b, B?a, B?b (local) ~ a, b (global)
.... : A!a, A!b, B?b, B?a (local) ~ b, a (global)
Missing: A!b, A!a, B?a, B?b (local) ~ a, b (global)

Figure 4.8: Assuming an EO channel, global coverage does not imply local coverage

A sends the messages `a` and `b` to service *B* in arbitrary order over a single EO channel. The global model depicts the global receive viewpoint for the choreography. In each model state `1` is the initial state, while all other states are accepting states.

As illustrated, when generating a test suite that covers every transition in the global model, the result could be a test suite consisting of the two test cases $\{(\mathtt{a}, \mathtt{b}), (\mathtt{b}, \mathtt{a})\}$ characterized by the differently dotted lines. In the figure, for each test case a valid sequence of the message events of *A* and *B* is given. However, though equivalent to the global test cases, these two sequences are not covering every event of the local viewpoint models. Two send events of the service *A* are not covered.

Local transition coverage. Less intuitive is the fact that transition coverage of the local viewpoint models also does not guarantee transition coverage of the global receive viewpoint. The counterexample depicted in Figure 4.9, again shows the global and local models of two services *A* and *B*. In each model, state `1` is the initial state, while all other states are accepting states. Each service is able to initiate a conversation by sending the message `a` or `b` respectively. Both services are allowed to send their message, as long as they have not received anything from their partner. A test suite that covers every local transition, is illustrated by dotted lines. The three test case, $\{(A!a, B?a), (B!b, A?b), (A!a, B!b, B?a, A?b)\}$, are covering the local

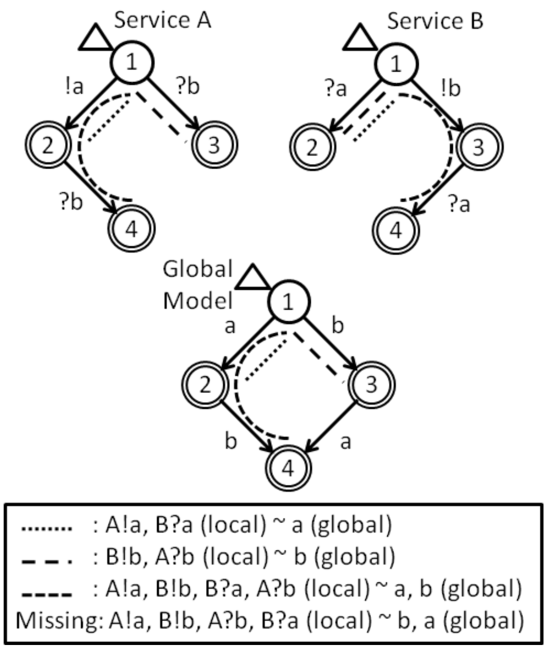

Service A
!a
?b
?b
Service B
?a
!b
?a
Global Model
a
b
b
a
1
2
3
4
: A!a, B?a (local) ~ a (global)
: B!b, A?b (local) ~ b (global)
: A!a, B!b, B?a, A?b (local) ~ a, b (global)
Missing: A!a, B!b, A?b, B?a (local) ~ b, a (global)

Figure 4.9: Assuming an EO channel, local coverage does not imply global coverage

models of service A and service B but not the global model. It has to be added, that the given example is describing a consistent choreography, as every local trace can be realized by a global one and, more important here, also each global trace can be realized locally. As given in the figure, the missing global trace $(\mathtt{b}, \mathtt{a})$ is realized by $(A!a, B!b, A?b, B?a)$.

Although transition coverage of global model does not cover all send and receive events of the local model, it does however cover the receive events. This is due to the the assumed receive semantics of the global model. As explained above, this is a main goal for service integration testing, because it is able to reveal all cases of message racing as well as other transition related faults. Consequently, transition coverage of the global viewpoint is targeted for the remaining test generation approach, exploiting the positive effect that the state space that needs to be explored is significantly lower compared to the composed system.

4.3.3 Industrial Requirements

Important from an industrial perspective is that the test generation approach further aims to be optimal regarding the minimization of the effort in the subsequent test phases, namely test concretization (e.g., provisioning of test data), test execution, and test analysis. Based on practical experience of the testing process at SAP [8], it can be concluded that the test optimization should be driven by the following objectives sorted from highest to lowest priority:

1. *Each test case should start in the initial state and end in an accepting state*: Bringing a complex system in the needed state using test preambles is complicated and time consuming. Stopping a test while the system is not in an accepting state leads to problems with inconsistent data that might hamper consequent test executions. Even though such a procedure does not solve the test data stability challenge described in Section 3.3, it significantly reduces the manual effort associated with test data corrections.

2. *The length of the longest generated test case should be minimal*: The longer a test case gets[3], the harder it is to maintain, e.g. because of lower complexity, less debugging effort and better error reproducibility and fault isolation. Therefore, especially for generated tests, a top priority is to carefully control path lengths. Consequently, the test suite with the stated property is chosen, even if there are other test suites with a lower overall size[4].

3. *Message racing in the test suite should be minimal*: As explained, testing the effects that message racing has on the interaction is an important part of each test suite. Consequently, each transition of the global receive viewpoint has to be covered and therefore all possible cases of message racing are contained. However, tests are mostly carried out in rather idealistic environments where messages are received in the same order they have been sent, independent of the chosen channel. Therefore, during test execution, message racing has to be emulated on the channel in a controlled way, usually leading to much higher effort. Therefore, message racing should not be emulated more often than necessary.

4. *The number of test steps should be minimal*: As the test effort generally increases with the overall size of the test suite, it should be as low as possible.

4.4 Summary

In this chapter, a development approach for a model-driven SOA development has been described that also incorporates agile methods on service component level. As explained, this approach is the contextual framework

[3]The length is determined by the number of transitions it executes.
[4]Sum of all executed transitions in the test suite.

of the dissertation. The incorporated choreography modeling, model verification and model-based testing activities are motivated by the requirements that have been described in Section 3.2.

In Section 4.2 it has been explained that the requirements for a choreography modeling language should be addressed by designing a custom DSL, including a global receive viewpoint and a local viewpoint per component. These design decisions are based on the requirements *Explicit message send and receive*, *Global and local views*, and *Pairwise choreographies.* How the other introduced requirements, i.e. *Detailed message description*, *Infinite state space*, *Interaction termination*, *Channel modeling*, and *State-based modeling* is addressed by the developed choreography language MCM, will be explained in Chapter 5. Further, it has been laid out in Section 4.2.3 that incorporating model-based testing and formal verification for ensuring software quality demands trace equivalence between the different abstraction layers.

In Section 4.3, the error assumptions that drives the model-based service integration testing have been described. Further, it was deduced that transition coverage of the global choreography viewpoint would uncover most of the related faults. A special test objective that has been underpinned by the discussions of Section 3.2, was to check that message racing has been considered during software development. As discussed in Section 4.3, when applying transition coverage on the global receive viewpoint model for test generation, the missing or wrong implementation of receive events can be detected as transition faults.

Chapter 5

Message Choreography Model

According to the W3C's Web Service Glossary [W3C04a], "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". In this chapter, the choreography modeling language MCM is described. The development of MCM is one of the major scientific contributions of this dissertation and has been conducted with the aim to enable and support the model-driven development process described in Chapter 4. The underlying ideas have been published in [7].

In the following, the general concept of MCM is described using the running example from Section 3.1 for illustration. Afterwards, the MCM syntax and semantics are presented in detail. Finally, the tool support for MCM is introduced that enables the modeling and realizes the consistency enforcement for MCM, described in Section 4.2.

5.1 MCM Overview

Choreography modeling languages complement the structural information of the communicating components (e.g. service interface descriptions and message types) with information about the message exchange between them. In Section 3.2 the objectives for choreography modeling and the resulting requirements for a modeling language have been introduced. Further, it was concluded that recent choreography languages were not conforming to the given requirements. Based on these requirements, in Section 4.2, important design considerations for choreography modeling have been explained, leading to the decision to design a domain-specific modeling language, which is incorporating a global receive viewpoint and local component viewpoints. In this section, the Message Choreography Model (MCM) language is described that incorporates the given design decisions and addresses the aforementioned requirements. It consists of different modeling artifacts, each defining distinctive aspects of service choreographies:

- *Global Choreography Model.* The global choreography model (GCM) is an extended finite state machine (EFSM), which specifies a high-level view of the conversation between service components. Its purpose is to define every allowed sequence of observed message receive events.
- *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. As explained in Section 4.2.2, these structural copies can be restricted manually, by deactivating some of the local transitions.
- *Channel Model.* The channel model (CM) describes the characteristics of the communication channel on which messages are exchanged between the service components. For services such characteristics are formalized by WS-RM standard [OAS07b] in practice and describe the channel's reliability guaranties (see Section 2.3.3).

Figure 5.1 shows, how the example from Section 3.1 can be modeled, using the Eclipse-based MCM editor (described in Section 5.5). The GCM at the top of Figure 5.1 represents the global receive viewpoint that has been discussed in Section 4.2.2. The arrows labeled with an envelope depict the interactions `Request`, `Offer`, `Cancel`, `Order`, and `Cancel(deprecated)` which are ordered with the help of the states `Start`, `Requested`, `Reserved`, and `Ordered`. The states `Ordered` and `Start` are so-called accepting states (thus connected with the filled circle). In these states, the communication between the partners is allowed to terminate.

Similar to the description in Section 4.2.2, interactions labeled with `Cancel(deprecated)`, describe the receive of `Cancel` messages by the seller that happened after receiving a new `Request` message due to message racing. Consequently, `Cancel` and `Cancel(deprecated)` messages are physically identical. Only by considering the context in which they are received, they can be distinguished.

To allow the distinction and hence to keep the model deterministic, a set variable called `ID_SET` is declared and initialized with $\emptyset$. It stores the transaction IDs from the header of `Offer` messages that have not yet been addressed by `Cancel`, `Cancel(deprecated)` or `Order` messages (the headers of these messages also store the ids). Whenever a `Offer` interaction takes place, an assignment

$$\texttt{ID\_SET} := \texttt{ID\_SET} \cup \{msg.\texttt{Header.ID}\}$$

is executed, referring to the ID stored in the header of the `Request` message. To distinguish between a deprecated and an actual `Cancel` in state

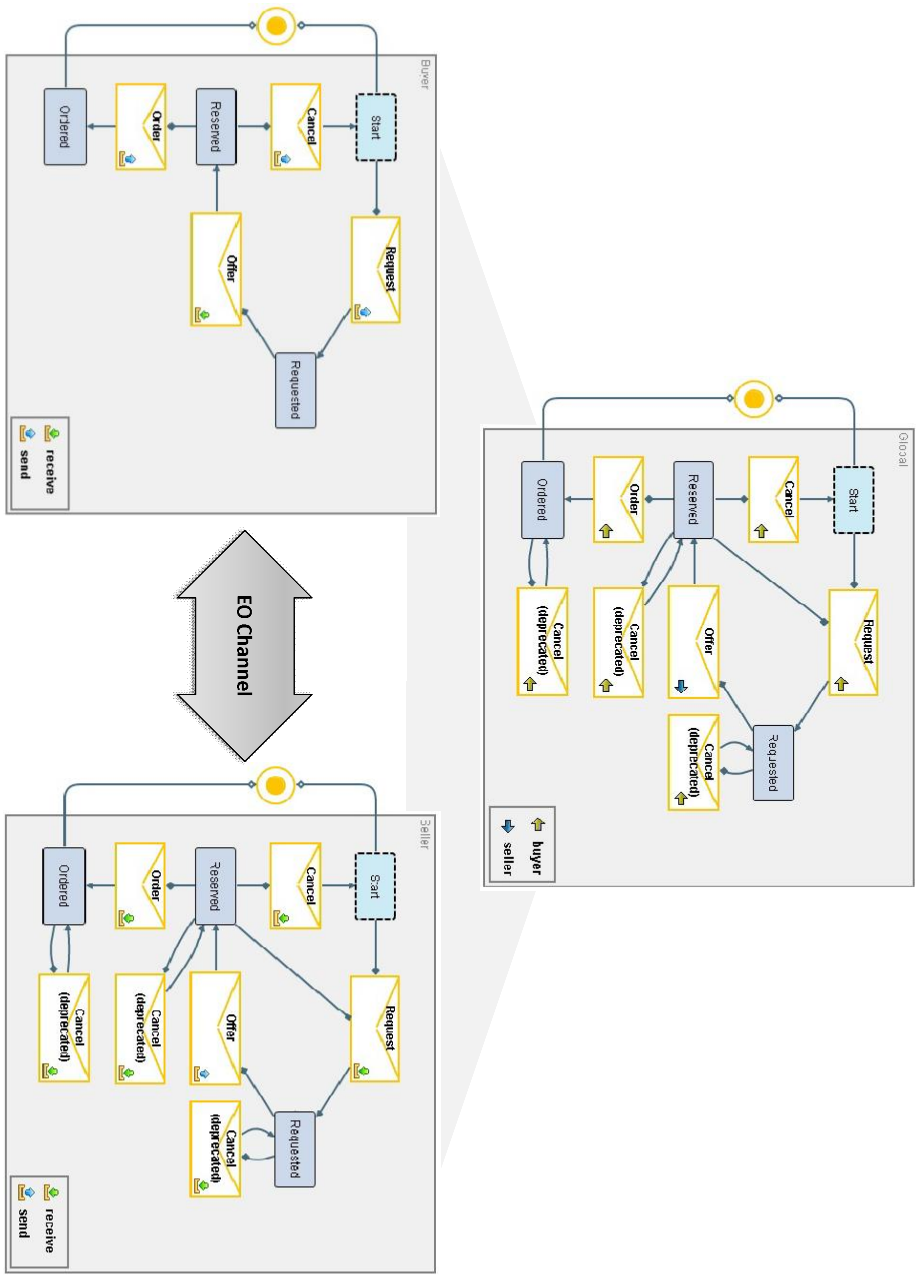


Figure 5.1: GCM (top) of the choreography and LPMs of the buyer (left) and the seller (right)

`Reserved`, for the interaction `Cancel` an additional guard

$$\texttt{ID\_SET}\backslash\{msg.\texttt{Header.ID}\}) = \emptyset \land msg.\texttt{Header.ID} \in \texttt{ID\_SET}$$

can be modeled in MCM, while for `Cancel(deprecated)` the guard

$$\texttt{ID\_SET}\backslash\{msg.\texttt{Header.ID}\}) \neq \emptyset \land msg.\texttt{Header.ID} \in \texttt{ID\_SET}$$

has been added. Note, that this assignment of guards and side effects is more sophisticated than the described assignment in Section 4.2.2. In Section 5.2 the formal syntax and the complete set of guards and assignments for the example is described.

The LPM of the buyer of the running example is depicted in the lower left part of Figure 5.1. It is a structural copy of the GCM, but the interaction symbols now represent either send or receive events of the buyer. Moreover some send events are "inhibited" by special local constraints. As described in Section 4.2.2 it is inhibited that a `Cancel(deprecated)` is ever sent and that a `Request` is sent in the `Reserved` state. Therefore, in the figure these send-events have been erased. However, due to possible message overtaking on an EO channel that is assumed in the running example, receiving a deprecated `Cancel` is possible on the seller side. The LPM of the seller is depicted in the lower right part of Figure 5.1.

5.2 MCM Syntax

In this section, the abstract syntax of MCM is presented. As explained in Section 3.2, limiting the choreography modeling to pairwise communications is reducing the overall complexity, without sacrificing valuable information in the vast majority of cases. Therefore, in the remaining discussion MCM is assumed to describe the choreography of exactly two participating components.

Consequently, a message choreography model $M = (G, L_1, L_2, C)$ consists of a GCM G, two LPMs L_1 and L_2 and a CM C. G, L_1, and L_2 are extended finite state machines, i.e. they incorporate unbounded variables as well as guards and actions at their transitions, which may reference these variables. In the following, L_1, L_2, and C are referred to as *composed system*. For expressing guards and actions, a constraint language is needed. It is described below, followed by the definition of the global and local models.

Constraint Language. As explained in Section 2.2 in the SOA domain, XML documents are utilized for messaging. Consequently, messages are characterized by some hierarchical record data type (or schema) representing the message type. An important feature of the set of terms *Term* of the constraint language is to reference elements of messages that are structured this way. For example, $msg.\texttt{Header.ID} \in Term$ points to the `ID` of the

`Header` element of the message referenced by the *msg* variable. The terms further contain the global variables (defined for the extended state machine, see below) and constants (including e.g. $0, 1, 2, \ldots, \emptyset$, etc.). *Term* is closed under application of arithmetic or set-theoretic operators. Hence, in a natural way, a simple type system can be built into *Term* so that syntactically illegal function applications can be excluded. For example, assuming that `ID_SET` is a variable, $\texttt{ID\_SET} \setminus msg.\texttt{Header.ID} \in \mathit{Term}$.

The set *Form* of formulas of the constraint language is then the set of first order formulas over *Term* and the predicates $=, <, >, \subseteq, \in$. This is necessary, because often all sub-nodes within a message should satisfy a certain condition. In the given example

$$(\forall x : x \in \texttt{Order.items} \rightarrow x.\texttt{status} = \texttt{Released}) \in \mathit{Form}$$

the constraint defines that all items of an `Order` message should be released.

Global Choreography Model. The GCM G is an extended finite state machine, represented by the tuple (S, E, s_0, T, I) where:

1. S is a finite set of *states*;
2. $s_0 \in S$ is the *initial state*;
3. $T \subseteq S$ is a set of *accepting states*;
4. $I \subseteq \mathbb{P}(S) \times S$ is a finite set of *interactions* indicating the transitions of the state machine;
5. V is a finite set of *variables*, while for each interaction $i \in I$ there is a special variable $msg_i \in V$ referring to the message instance, exchanged during an interaction;
6. $v_0 : v \rightarrow \mathit{Term}$ is the *initial variable assignment* (see below).

An interaction $i = (\{s_n, ..., s_m\}, s_x)$ is therefore associated with a set of enabling states $\{s_n, ..., s_m\}$ and a successor state s_x. That means the interaction i can take place when the system is in one of the enabling states and will change the system state to the successor state. In Figure 5.1 interaction `Request` has the enabling states `Start` and `Reserved` as well as the successor state `Requested`.

With each interaction $i \in I$, a function $\mathit{sender}(i) \in \{P_1, P_2\}$ is associated that indicates which partner is responsible to send the message of this interaction. Each interaction i is associated with a guard $\mathit{pre}(i) \in \mathit{Form}$, which describes a condition under which the interaction can be observed and a side-effect $\mathit{act}(i) \in (V \rightarrow \mathit{Term})$, which describes assignments of variables from V to terms during the transition. Further, there is an initial assignment of terms to variables E.

Example 1 *As explained before, the GCM for the running example includes the set $V = ID\_SET$ and the following guards and actions:*

$$\begin{array}{rcl} pre(\texttt{Request}) & = & msg.\texttt{Header.ID} \notin \texttt{ID\_SET} \\ pre(\texttt{Order}) & = & msg.\texttt{Header.ID} \in \texttt{ID\_SET} \\ pre(\texttt{Cancel}) & = & \texttt{ID\_SET} \backslash msg.\texttt{Header.ID}) = \emptyset \\ & & \wedge\ msg.\texttt{Header.ID} \in \texttt{ID\_SET} \\ pre(\texttt{Cancel(depr.)}) & = & \texttt{ID\_SET} \backslash msg.\texttt{Header.ID} \neq \emptyset \\ & & \wedge\ msg.\texttt{Header.ID} \in \texttt{ID\_SET} \end{array}$$

$$\begin{array}{rcl} act(\texttt{Request})(\texttt{ID\_SET}) & = & \texttt{ID\_SET} \cup \{msg.\texttt{Header.ID}\} \\ act(\texttt{Order})(\texttt{ID\_SET}) & = & \texttt{ID\_SET} \backslash \{msg.\texttt{Header.ID}\} \\ act(\texttt{Cancel})(\texttt{ID\_SET}) & = & \emptyset \\ act(\texttt{Cancel(depr.)})(\texttt{ID\_SET}) & = & \texttt{ID\_SET} \backslash \{msg.\texttt{Header.ID}\} \end{array}$$

Local Partner Model. Like a GCM, a local partner model LPM L_j is an extended finite state machine with a finite set S_j of states, an *initial state* $s_0^j \in S$, and a set of accepting states $T_j \subseteq S$. In addition, it entails two finite disjoint sets $I_j^!$ and $I_j^?$ with $(I_j^! \cup I_j^?) \subseteq \mathbb{P}(S_j) \times S_j$ of send and (resp.) receive events forming the transitions of the state machine. As in the GCM, there is a finite set V_j of variables. Further, for each send/receive event $e \in E$, where $E = I_j^! \cup I_j^?$, there is a special variable $msg_j^e \in V_j$, referring to the message sent or received in e, as well as a message type $MT(e)$. Guards and side-effects are assigned to send and receive events as for interactions of the GCM.

A basic strategy to ensure consistency is to demand that for every send/receive event $!\,i$ or $?\,i$ of an LPM there is a corresponding interaction i with the same message type in the GCM. If it is a send event $!\,i$ in L_j, then $sender(MT(i)) = j$, if it is a receive event $?\,i$, then $sender(MT(i)) \neq j$. Different other ways to ensure consistency among GCM and LPMs will be discussed later in Section 5.5.

Example 2 *The lower part of Figure 5.1 shows the two LPMs L_1 (left) and L_2 (right) of the running example. For the interaction* `Request` *in GCM, there is* $!$`Request` *in L_1 and* $?$`Request` *in L_2. L_1 contains a set $V_1 = \{\texttt{ID\_SET1}\}$ and pre and act of the LPMs are copied accordingly from the GCM, e.g.:*

$$\begin{array}{rcl} pre(!\,\texttt{Request}) & = & msg_1.\texttt{Header.ID} \notin \texttt{ID\_SET}_1 \\ act(!\,\texttt{Request})(\texttt{ID\_SET}_1) & = & \texttt{ID\_SET}_1 \cup \{msg_1.\texttt{Header.ID}\} \end{array}$$

However, it is not correct to transform the interactions one-by-one into send/receive events. For instance, it should not be possible to send `Request` *in the state* `Reserved`. *Therefore,* $!\,\texttt{Request} = \{\texttt{Start} \mapsto \texttt{Requested}\}$ *while* $?\,\texttt{Request} = \{\texttt{Start} \mapsto \texttt{Requested}, \texttt{Reserved} \mapsto \texttt{Requested}\}$.

Channel Model. Given a set of message types MT used in the GCM, the channel model C is a total function from a sequence of messages (of types MT) to a sequence of messages (of types MT'). With $MT' \subseteq MT$ and a message sequence s, $\pi_{MT'}(s)$ denotes the projection of s to a sequence of messages of types MT'. Let $\pi_{MT'}(s)$ be canonically extended on the channel model. The channel model C is then based on assignments of disjoint subsets MT' of MT to channel reliability guarantees, which enforce that $\pi_{MT'}(C)$ satisfies certain properties. Reliability guarantees of the WS-RM standard described in Section 2.3.3 can be modeled as follows: *exactly once in order* (EOIO) where $\pi_{MT'}(C)$ is the identity function on interaction sequences and *exactly once* (EO) where $\pi_{MT'}(C)$ is a permutation on an interaction sequence (non-deterministic).

5.3 MCM Semantics

There are various ways to define an operational semantics for a GCM, LPM and a channel model. As explained in Section 4.2.3, a trace-based semantics will be used in this thesis. The description of this trace semantics is started by giving the relevant trace definitions for MCM. Afterwards, the relation of these traces is described.

5.3.1 Trace Definitions

MCM can be described as consisting of a GCM G and a composed system CS incorporating the two LPMs L_1 and L_2, and the channel model C. In the following, the trace semantics for G and CS will be given, followed by the description of the consistency relation between G and CS, which is based on trace equivalence according to the discussion in Section 4.2.3.

Traces of the GCM. For any GCM G, the traces of G, denoted by $Traces(G)$, is the set of sequences $(i_1, \ldots, i_n)$ of interactions, for which there exists a sequence $(s_0, \ldots, s_n)$ of states and a sequence of concrete variable assignments $v_0, \ldots, v_n$ such that s_0 and v_0 are initial, and for all $k = 1, \ldots, n$, s_{k-1} is an enabling state of i_k, s_k is the successor state of i_k, the guards of i_k are satisfied in v_{k-1}, the value of v_k equals the value of v_{k-1} except the updates $act_k(v_k)$, and s_n is an accepting state.

Traces of the CS. For the definition of traces of CS, first the semantics of LPM have to be fixed. By treating send and receive events similar to interactions of GCM, LPM's semantics is similar to the semantics of GCM.

CS consists of LPMs L_1 and L_2 and the channel model C. Its state space is defined by the notion of composed state $s_k^{CS} = (s_k^{L_1}, s_k^{L_2}, s_k^{C})$, consisting of a local state of each LPM and a state of the channel, described by a sequence of messages that are already sent but not yet received (that means they are on the channel). Therefore, while G has a finite number of states, CS may have an infinite number of states, if there are no (unnatural) restrictions on the channel size. The traces of CS, denoted by $Traces(CS)$, are defined by the sequences $(e_1, \ldots, e_n)$ of send or receive events e_k $(k = 1, \ldots, n)$ of the LPMs involved in the composed system, which satisfy the following property. There is a sequence $(s_0, \ldots, s_n)$ of composed states such that for all e_k (k=1,...,n), and its message type MT the following holds:

- In s_0 the channel is empty and L_1 and L_2 are in their initial states (i.e. composed initial state).
- The state of L_x in s_{k-1} is an enabling state of e_k, the state of L_x in s_k is the successor state of e_k, the guards of e_k are satisfied in v_{k-1} of L_x, the value of v_k equals the value of v_{k-1} except the updates $act_k(v_k)$, and the state and variable assignment of L_y $(x \neq y)$ in s_{k-1} equals the state and variable assignment of L_y in s_k.
- If e_k is a send event, then in s_k the channel sequence equals the channel sequence of e_{k-1} attached with a new message msg_k of type $MT(e_k)$, where msg_k satisfies the guards of e_k in $sender(MT(e_k))$.
- Let ch be the sequence of messages on the channel in s_{k-1}, with $C(ch) = (m_1, \ldots, m_q)$. If e_k is a receive event, then ch is not empty, m_q is of type $MT(e_k)$, m_q (interpreted as msg_k) satisfies the guards of e_k, and the channel sequence of s_k equals $(m_1, \ldots, m_{q-1})$.
- In s_n the associated states of L_1 and L_2 are accepting states and the channel is empty (i.e. composed accepting state).

5.3.2 Consistency Relation of GCM and CS

As explained, using the GCM for the test derivation (see Section 4.3) requires G to be an equivalence or under-approximation of CS, because otherwise globally generated test suites would contain traces without local equivalents (i.e. infeasible paths). On the other hand, allowing under-approximation prevents the application of formal mechanisms to prove conformance of CS to G (see [KRW09]) as it possibly omits allowed behavior. As concluded in Chapter 4, the targeted approach is therefore implying that trace equivalence of G and CS must be required.

Since G and CS have different alphabets, the alphabet of interactions used by G has to be mapped to the corresponding send and receive events in CS, in order to define a consistency relation between them. There are various applicable consistency relations between G and CS, depending on the viewpoint of the assumed global observer for G. A global observer in this respect, is an idealized entity that observes the message flow between the components associated with L_1 and L_2 and relates it to the interactions of G. In the following, the two global viewpoints, described in Section 4.2.2 are described.

Send-viewpoint. A GCM G describes a send-viewpoint of a composed system CS, if for each trace $(e_1, \dots, e_n)$ of CS there exists exactly one trace $(i_1, \dots, i_k)$ of G (and vice versa), such that if $(!\,e_1, \dots, !\,e_k)$ is the projection of $(e_1, \dots, e_n)$ to the send events, then $(MT(i_1), \dots, MT(i_k)) = (MT(!\,e_1), \dots MT(!\,e_k))$. In other words, G describes all sequences of send events that can be observed in a communication between L_1 and L_2.

Example 3 *If the traces of the composed system in Figure 5.1 are projected to send-events the following set, written as regular expression (as defined in [HU69]), is obtained:*

$$\{(\texttt{!Request !Offer !Cancel})^{*}\ (\texttt{!Request !Offer !Order})^{?}\}$$

In this case a send-viewpoint is constructed by simply taking the LPM of the buyer and transforming send/receive events into interactions.

Receive-viewpoint. G is a receive-viewpoint of CS if for each trace $(e_1, \dots, e_n)$ of CS there exists exactly one trace $(i_1, \dots, i_k)$ of G (and vice versa), such that if $(?\,e_1, \dots, ?\,e_k)$ is the projection of $(e_1, \dots, e_n)$ to the receive events, then $(MT(i_1), \dots, MT(i_k)) = (MT(?\,e_1), \dots, MT(?\,e_k))$. In other words, G describes all sequences of receive events that can be observed in a communication between L_1 and L_2. The receive-viewpoint thus reflects the possible loss of message order on the channel and is therefore best suited for integration testing.

Example 4 *The GCM in Figure 5.1 is a receive-viewpoint of the composed system with an EO channel in that figure, because it covers the projection of the traces from the LPMs to receive events.*

As explained in Section 4.2.2, the receive viewpoint is best suited for the derivation of integration tests, as it reflects message racing as experienced by the involved components.

5.4 MCM Transformation Semantics

In the previous section the trace semantics of MCM has been given. This trace semantics can be refined by describing a transformation of MCM artifacts to a modeling notation with a well-defined semantics. This is a common approach to define language semantics (cf. [Var02]). In this section, the transformation of MCM to the formal language Event-B [AH07], as published in [9], is presented.

Event-B fits quite naturally to MCM as interactions can be expressed seamlessly as events and the relationship between GCM and LPMs can be formulated as Event-B refinement. Apart from defining the MCM semantics, Event-B opens a variety of possibilities to analyze the model. These will be described in Section 5.5. In this section, a brief overview on Event-B is given. Afterwards, the translation of MCM artifacts to Event-B is introduced.

5.4.1 Introduction to Event-B.

Event-B [AH07] is an evolution of the B-Method [Abr96] that puts emphasis on a lean design. In particular, the core language of Event-B is (with a few exceptions) a subset of the language used in its predecessor. It distinguishes between static and dynamic properties of a system. While static properties are specified in a context, the dynamic properties are specified in a so-called machine. A context contains definitions of carrier sets, constants as well as a number of axioms. A machine basically consists of a finite set of variables and events. The variables form the state of the machine and can be restricted by invariants. The events describe transitions from one state into another state.

An event has the form

$$EVENT \mathrel{\widehat{=}} ANY\ t\ WHERE\ G(t,x)\ THEN\ S(x,t)\ END$$

It consists of a set of local variables t, a predicate G, called the guard and a substitution $S(x,t)$. The guard restricts possible values for t and x. If the guard of an event is false, the event cannot occur and is called disabled. The substitution S modifies the variables x. It can use the old values of x and the local variables t. For example, an event that takes two natural numbers a, b and adds the product ab to the state variable x could be written as

$$EVENT \mathrel{\widehat{=}} ANY\ a,b\ WHERE\ a \in \mathbb{N} \wedge b \in \mathbb{N}\ THEN\ x := x + a * b\ END$$

For events that do not require local variables, the abbreviated form

$$EVENT \mathrel{\widehat{=}} WHEN\ G(x)\ THEN\ S(x)\ END$$

can be used. The primary way to structure a development in Event-B is through incremental refinement preserving the system's safety and termination properties.

5.4.2 Design Considerations of the Transformation.

The transformation from MCM to Event-B contains a formal representation of both, the GCM and the CS, incorporating the two LPMs and the CM. Therefore, the subsequently described translation generates two Event-B machines, which use a common context. The *Global Model* describes the

GCM and the *Local Model* describes the composition (defined as in [But09]) of the two LPMs and the CM. Both machines describe the exchange of messages, the first in terms of observing a message, and the latter in terms of sending and receiving messages.

As messages with the same type and content may occur more than once, to each message a unique natural number is assigned, which is incremented when a new message is sent. Further to each message a type is assigned while it is possible to specify the content of the message as functions on the message. Because we aim at the use of a model checking technique the translation result is designed to be as deterministic as possible. We experimented with an assignment of types to messages which is non-deterministically initialized upfront; however this resulted in an indigestible state space for the model checker.

5.4.3 Transformation Description.

By defining a translation from the global and from the local MCM models into the two Event-B machines a precise semantics of MCM is obtained, which is presented in the following. The transformation is implemented and can thus be applied completely automatically.

Global Model. For each transition in the GCM exactly one event is generated. The states are represented by a global variable *status* with elements from a set type $s_1, \ldots, s_k$, with constants $s_1, \ldots, s_k$. It is initialized with $init \in S$. The basic translation of an Interaction $i \in I$ with $(s_1, \ldots, s_k,\ I,\ s_m) \in \Rightarrow$ is as follows:

$$\begin{aligned} i \mathrel{\widehat{=}} WHEN\ guard1 :\ & status = s_1\ \wedge\ \ldots\ \wedge\ status = s_k \\ & THEN\ act1 :\ status\ :=\ s_m\ END \end{aligned}$$

This basic translation must be augmented with preconditions and actions, associated with that interaction. Therefore, data types, constants, variables, terms and formulae used in MCM have to represent in terms of Event-B. This is done as follows. For each data type $t \in T$ a set is defined in the Event-B context, without explicit characterization of elements. These sets are named in Event-B according to their type name $name(t)$. For each complex data type $t = (f, t')$ we define a partial function $f :\ name(t) \nrightarrow name(t')$. f is initialized with $f := \emptyset$.

The constants and global variables are defined in a standard way. For each constant $c \in C_t$ an element is added to the set $name(t)$. For the interactions $I = \{i_1, \ldots, i_n\}$ we additionally define a set $MESSAGES = \{name(itype(i_1)), \ldots, name(itype(i_n))\}$.

Example 5 *Consider the interaction* `Request` *with*

$$pre(\texttt{Request}) = msg.\texttt{Header.ID} \notin \texttt{ID\_SET}$$

and

$$act(\texttt{Request})(\texttt{ID\_SET}) = \texttt{ID\_SET} \cup \{msg.\texttt{Header}.\texttt{ID}\}$$

of the running example. For it, the functions $Header : \mathbb{N} \twoheadrightarrow MessageHeader$ *and* $ID : MessageHeader \twoheadrightarrow InstanceID$ *(MessageHeader and InstanceID here are the corresponding names from name(T))are defined, as well as the local variables* $t1$ *and* $t2$ *in order to choose appropriate values to be assigned in the functions. Because* $ID\_SET \in T_{Set(InstanceId)}$ *an Event-B variable* ID_SET *of type* $\mathbb{P}(InstanceID)$ *is defined.*

$$
\begin{array}{l}
Request \mathrel{\widehat{=}} ANY\ t1\ t2\ WHERE \\
grd1 :\ status = Reserved\ \lor\ status = Start \\
grd2 : t1\ \in\ MessageHeader \\
grd3 : t2\ \in\ InstanceID \\
grd4 : t3\ \notin\ ID\_SET \\
grd5 : t1 \in dom(ID) \Rightarrow ID(t1) = t2 \\
THEN \\
act1 :\ status := Requested \\
act2 :\ Header(msg) := t1 \\
act3 :\ ID(t1) := t2 \\
act4 :\ type(msg) := Request \\
act5 :\ ID\_SET := ID\_SET \cup \{t3\} \\
act6 :\ msg := msg + 1 \\
END
\end{array}
$$

The guard $grd5$ *describes a consistency property: if the function is already defined on an element, then the value must be the corresponding term.*

For the accepting state $e_i \subseteq S$ a special event *terminate* with a guard $status = c1\ \lor\ \ldots\ \lor\ status = c_1$ (for all $c_i \in e_i$) and an action $acceptingstate := true$ is defined, where *acceptingstate* is a global variable. In each event from the translation of GCM an additional action $acceptingstate := false$ is added. As a result, *acceptingstate* equals true, iff the system state is an accepting state.

Local Model. In the local model, events representing sending and receiving of messages are generated. Depending on the viewpoint either the send or the receive event can be defined to be a refinement of the corresponding interaction in GCM.

By definition of LPMs, the variables from V and the status variable are duplicated (one for each partner). The variable *msg* is translated as for the GCM in order to keep the unique message enumeration. It is only used by send events, where it is set in the same way as in the GCM. In receive events, local variables (parameters) are used in order to obtain some message from a channel.

A channel is defined as a global variable of type $\mathbb{P}(\mathbb{N})$, denoting the set of messages being exchanged. It is initialized with $\emptyset$. Typically, there are two partners P_1 and P_2 and two sequencing contexts (EO and EOIO). In that case, four possible channels can be obtained in the model (two for each direction).

Example 6 *Below, a translation of the interaction* `Request` *from the LPMs for the partners buyer (B) and seller (S) of the example is shown. The duplicated variables can be distinguished by the corresponding prefixes. The channel from buyer to seller having the sequencing EO is denoted by* $channel\_BS\_EO$.

$$
\begin{array}{l}
send\_Request \mathrel{\widehat{=}} ANY\ t1\ t2\ t3\ WHERE \\
grd1:\ B\_status = Reserved\ \vee\ B\_status = Start \\
grd2:\ t1 \in MessageHeader \\
grd3:\ t2 \in InstanceID \\
grd4:\ t3 \notin B\_ID\_SET \\
grd5:\ t1 \in dom(ID) \Rightarrow ID(t1) = t2 \\
THEN \\
act1:\ B\_status := Requested \\
act2:\ Header(msg) := t1 \\
act3:\ ID(t1) := t2 \\
act4:\ type(msg) := Request \\
act5:\ B\_ID\_SET := B\_ID\_SET\ \cup\ \{t3\} \\
act6:\ channel\_BS\_EO := channel\_BS\_EO\ \cup\ \{msg\} \\
act7:\ msg := msg + 1 \\
END
\end{array}
$$

$$
\begin{array}{l}
receive\_Request \mathrel{\widehat{=}} m\ WHERE \\
grd1:\ S\_status = Reserved\ \lor\ S\_status = Start \\
grd2:\ m \in channel\_BS\_EO \\
grd3:\ type(m) = Request \\
grd4:\ m \in dom(Header) \\
grd5:\ Header(m) \in dom(ID) \\
grd6:\ ID(Header(m)) \notin S\_ID\_SET \\
THEN \\
act1:\ S\_status := Requested \\
act2:\ S\_ID\_SET := S_ID_SET\ \cup\ \{ID(Header(m))\} \\
act3:\ channel\_BS\_EO := channel\_BS\_EO \setminus \{m\} \\
END
\end{array}
$$

The translation of a send event is very similar to the translation of the corresponding event in GCM. In receive events, all function values are already set, so that the purpose is to find a suiting message m in the channel and "receive" it (i.e. delete it from the channel). If a sequencing context is EOIO then an additional guard, enforcing that the message m has the smallest number in the channel, is needed.

For inhibitor conditions $inhib(i) = C$ (with $i \in I$) a guard $status \notin C$ is added to the event $send\_i$. In the example, the guard $grd6 :\ B\_status \notin \{Reserved\}$ is added to $send\_Request$. It remains future work to optimize the translation by simplifying this and $grd1$ to $B\_status = Initial$.

Accepting states are treated similar to the translation of GCM, except that it is demanded additionally that $channel = \emptyset$ for all of them, because only if all channels are empty, the system can enter into an accepting state. For all other events of the translation from the LPM, an action $acceptingstate := false$ is added.

5.5 MCM Tool Support

In previous sections, the syntax and semantics of MCM has been given. This section introduces the tooling of MCM that has been implemented to guide the creation of MCM models and enforce its consistency.

As depicted in Figure 5.2, an Eclipse-based editor has been implemented that is utilizing the Meta-Object Facility (MOF) standard [OMG06] for the definition of the underlying MCM meta-model. Figure 5.1 is a collage of screenshots of the editor, showing the three different viewpoints (i.e. GCM, LPM_1 and LPM_2) of the MCM instance for the running example.

The automatic transformation of MCM to Event-B described in Section 5.4 is utilized to connect the MCM editor with the Rodin

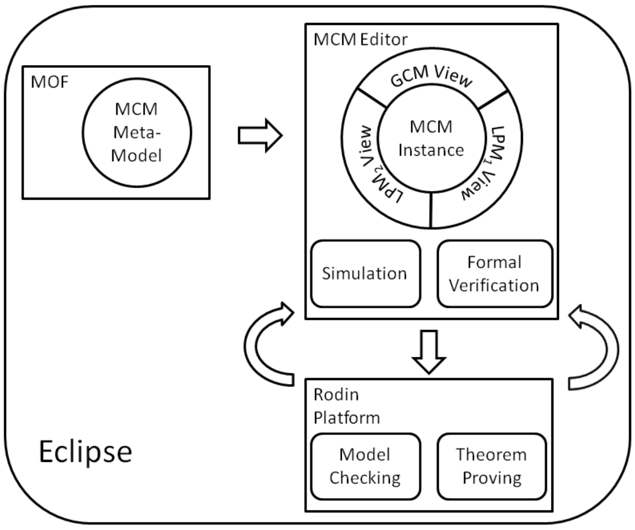


Figure 5.2: Overview of MCM modeling tools

platform [ABHV08]. It is also Eclipse-based and includes a set of formal verification tools. Currently, the model checker ProB [LB08] and the theorem provers of Atelier B [Ste01] are used to ensure MCM's consistency obligations. In the following, for each consistency relation (as introduced in Section 4.2.3) the tool-based realization is described.

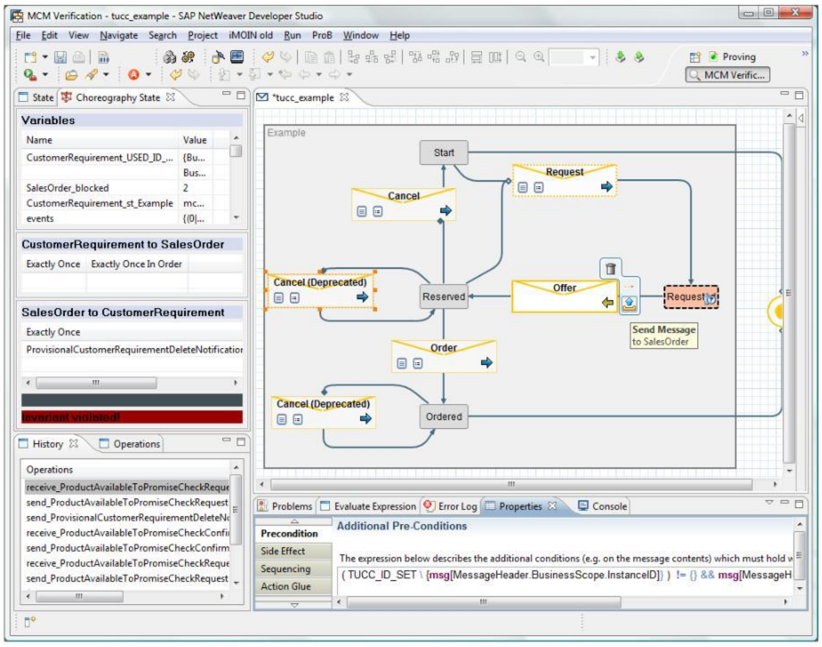
5.5.1 Ensuring Consistency between Requirements and MCM

In order to be sure that the MCM model corresponds to what the modeler intended to express and what has been informally described in requirements documents, a simulation tool has been added to the editor. By using the formal representation of MCM in Event-B, ProB is used as a model checker backend to interactively simulate sending and receiving of messages. In each state of the simulation the active states of the partners as well as of the channel. Currently enabled message send and receive events are presented to the modeler, who may select one of them manually, leading the simulation into the next state.

Figure 5.3 illustrates this. The highlighted state depicts the current state of the partners and the highlighted interaction can be clicked by the user to perform send/receive actions (here: sending `Offer` is possible). On the left hand side, the state of the channel and the history of events is depicted.

5.5.2 Ensuring Consistency between MCM Viewpoints

By deducing from the given definition of MCM, inconsistency of the models can have the following reasons:



Variables

Name	Value
CustomerRequirement_USED_ID_...	{Bu...
	Bus...
SalesOrder_blocked	2
CustomerRequirement_st_Example	mc...
events	{(0|...

CustomerRequirement to SalesOrder

Exactly Once	Exactly Once In Order

SalesOrder to CustomerRequirement

Exactly Once
ProvisionalCustomerRequirementDeleteNotificatior

```
Operations
receive_ProductAvailableToPromiseCheckReque
send_ProductAvailableToPromiseCheckRequest
send_ProvisionalCustomerRequirementDeleteNo
receive_ProductAvailableToPromiseCheckConfi
send_ProductAvailableToPromiseCheckConfirm
receive_ProductAvailableToPromiseCheckReque
send_ProductAvailableToPromiseCheckRequest
```

The expression below describes the additional conditions (e.g. on the message contents) which must hold w

```
( TUCC_ID_SET \ {msg[MessageHeader.BusinessScope.InstanceID]} ) != {} && msg[MessageH
```

Figure 5.3: Interactive simulation of MCM

- *Syntactical Inconsistency.* GCM and LPMs could syntactically not fit to each other. For instance there could be send events in LPM for which there are no interactions in GCM. Especially in an agile environment such simple consistency properties are in danger to be violated, e.g., when both levels are modified simultaneously and global and local views therefore get out of sync.

- *Semantical Inconsistency.* The composed system could not realize the behavior of GCM, i.e. CS and GCM are not receive-viewpoint trace equivalent. A special case is that for a GCM no trace equivalent CS exists at all. The choreography is then said to be not *locally enforceable.*

In general, for ensuring consistency between local and global views two competing, tool supported approaches exist: a generative approach where the LPMs are generated from the GCM, and a checking approach where global and local models are created separately, but having a consequent consistency check installed. While the first approach ensures that global and local views are always consistent, it makes changes to the local models considerably more difficult, since these would be overridden by re-generation from the global model. The latter approach allows for such "asymmetric" changes, but requires manual effort to update the global view when changes to the local models are made.

A pragmatic mix of these approaches however, seems most appropriate. The first of the issues mentioned above, in this case is addressed by maintaining the two LPMs as structural copies of the GCM, while allowing for additional guards on the LPM. This is implemented in the MCM editor by realizing the LPMs and the GCM as views on a common model instance. Trace equivalence can however not be ensured by syntactic means. Therefore model checking and theorem proving techniques have to be applied. In the MCM editor they are provided by connecting it to the Rodin platform. In the following, the implemented countermeasures to the given causes of inconsistency are described in detail.

Enforcing Syntactical Consistency. As described, the problem of keeping local and global views synchronized is solved by having a single meta-model for both GCM and LPM, while for each choreography, GCM as well as the two LPMs for each involved partner are views on one common instance of that meta-model.

For each partner, an LPM view is obtained from a GCM as follows. An interaction in the GCM is interpreted as a send or as a receive event in the LPM of the considered partner. A state in the GCM is represented by a corresponding state in the LPM, and constraints and effects are transferred accordingly.

Following this approach, the addition of a send-event to an LPMs automatically leads to the addition of an interaction in GCM and to the addition of a receive-event to the other partner's LPM. However, semantically the GCM states and the corresponding LPM states are different. The former denotes a globally observable state (based on the chosen message exchange viewpoint), while the latter denotes the latest information that the components have about the global state. These states are in general not equal because of the latency of message transmission.

By this technique, the most general LPMs, which might realize a choreography given by a GCM, are obtained. In order to allow for asymmetric resolution strategies, the LPMs can be augmented with additional guards. As discussed in Section 3.2, the added guards may only restrict that messages are sent and are further only visible in the particular LPM, but not in GCM or in the LPM of the other partner. The given implementation thus ensures syntactical consistency of GCM and LPMs during the whole modeling life-cycle.

Enforcing Semantical Consistency. Trace equivalence of the GCM and the CS can not be ensured by the above described syntactic means. Therefore, the MCM editor further provides checks which prove consistency, based on translating the choreography model to Event-B (see Section 5.4) and checking the obtained formal model with Rodin platform tools [ABHV08] and Rodin based plugins. To show trace equivalence in Event-B essentially boils down to showing the refinement of the Event-B machine generated for the combined system towards that of the GCM and vice versa.

Another important property is the absence of unconsumable messages. Whenever a message is being exchanged the receiver of the message must be ready to receive it. This property can be encoded as an invariant of the generated Event-B machine for the LPMs. Since proving these properties still requires a considerable amount of user interaction, also the model-checking tools provided by ProB [LB08] are utilized, which are not able to completely prove the refinement relation in general (because of the unlimited size of the channel), but give good feedback in cases where the model still contains errors.

A detailed description of the realized semantical consistency checks can be found in [KRW09].

To visualize these cases of modeling errors, the simulation plugin to the MCM editor has been enhanced such that it can represent the errors in the model by highlighting the states of both partners in the conflict situation and by re-playing the sequence of interactions leading to it.

5.5.3 Ensuring Consistency between MCM and Implementation

For each service component involved in the choreography (represented by LPMs) more detailed development artifacts, such as implementation code or other models exist. In comparison to LPMs they take into account the messaging behavior w.r.t. more than one partner component and internal actions that are unrelated to communication.

In the context of SAP, such components are usually described with the help of models. These specify contained attributes (and their types) and state transition diagrams, describing the effect of actions (such as service calls) on the component's state. By provision of a translation from these models to Event-B a formal representation can be obtained, which allows to show that they constitute a refinement of the LPM and thus preserve the properties specified in the LPM.

However, as LPM and component models operate on a separate state space. Therefore the modelers are required to add glue expressions to each of the states $s \in S_j$ of each LPM L_j. To guide this, the MCM editor is equipped with an expression language over the state of the business components. It is used to assign an expression $glue(s)$ to s.

Example 7 *In the running example, the* `Reserved` *state in the LPM of the Buyer may correspond to the status attribute* `OfferReceived` *in the buyer's purchase order business component. Thus it will be specified:*

$$glue(\texttt{Reserved}) \;=\; (\mathit{Status} : \texttt{PurchaseOrder.OfferReceived} = \mathit{true})$$

A detailed description of the formalization and transformation of development models, as well as the realization of the refinement checks is out of scope of this thesis. It will be available as a project deliverable [KWR10].

As described in Section 4.2.3, the targeted trace equivalence of LPMs and the corresponding components can only be achieved by combining the described formal checks of trace inclusion (i.e. refinement) of the implementation in MCM with trace inclusion of MCM in the implementation, which is ensured by testing. The details of this test approach are described separately in Chapter 6.

5.6 Summary

Chapter 4 introduced a development approach for a model-driven SOA development and motivated the design of a choreography modeling language. In this chapter, the choreography modeling language MCM has been described that is needed to realize the given development approach.

Section 5.1 introduces the MCM modeling artifacts for the global and local viewpoints (GCM and LPM), and the channel model. The formal definition of MCM Syntax and Semantics has been provided in Sections 5.2 and 5.3. Section 5.4 gives an account of a transformation of MCM to Event-B. This transformation is a prerequisite to ensure the various consistency obligations of MCM that have been introduced in Section 4.2.3. The tool-supported realization of these consistency obligations is explained in Section 5.5.

As MCM is based on the conceptual decisions described in Section 4.2, it already fulfills some of the requirements for choreography modeling that have been deduced in Section 3.2. The remaining requirements *Detailed message description*, *Infinite state space*, *Interaction termination*, *Channel modeling*, and *State-based modeling* have been satisfied by supplying a constraint language to describe message content, defining GCM and LPM as EFSMs with a classical notion of termination, and including a channel model for MCM.

Chapter 6

Test Generation

The envisioned SOA development approach given in Chapter 4, utilizes choreography models for model-based service integration testing. In Section 4.3, the general concepts of the intended MBT are explained. In this chapter, the realization of the testing approach is described in detail. It is based on the choreography language MCM that has been described in Chapter 5.

Section 6.1 introduces the developed test generation process. The testing framework that supports this process is described in Section 6.2. It integrates, various test generation methods. These are discussed in Sections 6.3, 6.4, and 6.5.

6.1 Test Generation Process

The core idea of 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. However, 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 modeled easily, as it is deeply rooted in the internal behavior of the components[1].

However, using MBT for service integration promises to reduce the manual effort by automatically generating suitable sets of test cases for desired coverage of the choreography model. As explained in Section 4.3, it will be aimed at transition coverage of the GCM. Therefore, the following four-step

[1]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.

approach for test generation has been deduced.

Step 1: Global test suite. A test generator generates a set of globally observable message sequences that covers each transition of the GCM. As explained in Section 4.3 all generated test cases have to start in the initial state and end in an accepting state of the GCM. The resulting test suite might vary depending on the utilized test generator.

Example 8 *Based on the GCM (global receive viewpoint) of the running example, depicted in Figure 4.5, an assumed test generator that computes terminating traces breadth first until full transition coverage is achieved, would derive the following traces.*

$$
\begin{aligned}
&< \texttt{Request}, \texttt{Offer}, \texttt{Order} >,\\
&< \texttt{Request}, \texttt{Offer}, \texttt{Cancel} >,\\
&< \texttt{Request}, \texttt{Offer}, \texttt{Cancel}, \texttt{Request}, \texttt{Offer}, \texttt{Order} >,\\
&< \texttt{Request}, \texttt{Offer}, \texttt{Cancel}, \texttt{Request}, \texttt{Offer}, \texttt{Cancel} >,\\
&< \texttt{Request}, \texttt{Offer}, \texttt{Request}, \texttt{Cancel}, \texttt{Offer}, \texttt{Order} >,\\
&< \texttt{Request}, \texttt{Offer}, \texttt{Request}, \texttt{Cancel}, \texttt{Offer}, \texttt{Cancel} >,\\
&< \texttt{Request}, \texttt{Offer}, \texttt{Request}, \texttt{Offer}, \texttt{Cancel}, \texttt{Order} >,\\
&< \texttt{Request}, \texttt{Offer}, \texttt{Request}, \texttt{Offer}, \texttt{Cancel}, \texttt{Cancel} >,\\
&< \texttt{Request}, \texttt{Offer}, \texttt{Request}, \texttt{Offer}, \texttt{Order}, \texttt{Cancel} > .
\end{aligned}
$$

Step 2: Local test suite. For each global test case a corresponding local trace (i.e. a traces of the composed system) is computed. This is necessary, because the global test cases only specify 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.

Example 9 *For the running example, the local test suite is given below. To keep the traces concise, the redundant*[2] *sender and receiver name for each event is omitted. Note that all the enumerated traces are incorporated in Figure 4.4.*

[2]For each message sender and receiver can be determined unambiguously.

$< !\texttt{Request}, ?\texttt{Request}, !\texttt{Offer}, ?\texttt{Offer}, !\texttt{Order}, ?\texttt{Order} >$,
$< !\texttt{Request}, ?\texttt{Request}, !\texttt{Offer}, ?\texttt{Offer}, !\texttt{Cancel}, ?\texttt{Cancel} >$,
$< !\texttt{Request}, ?\texttt{Request}, !\texttt{Offer}, ?\texttt{Offer}, !\texttt{Cancel}, ?\texttt{Cancel}$,
$!\texttt{Request}, ?\texttt{Request}, !\texttt{Offer}, ?\texttt{Offer}, !\texttt{Order}, ?\texttt{Order} >$,
$< !\texttt{Request}, ?\texttt{Request}, !\texttt{Offer}, ?\texttt{Offer}, !\texttt{Cancel}, ?\texttt{Cancel}$,
$!\texttt{Request}, ?\texttt{Request}, !\texttt{Offer}, ?\texttt{Offer}, !\texttt{Cancel}, ?\texttt{Cancel} >$,
$< !\texttt{Request}, ?\texttt{Request}, !\texttt{Offer}, ?\texttt{Offer}, !\texttt{Cancel}, !\texttt{Request}$,
$?\texttt{Request}, ?\texttt{Cancel}, !\texttt{Offer}, ?\texttt{Offer}, !\texttt{Order}, ?\texttt{Order} >$,
$< !\texttt{Request}, ?\texttt{Request}, !\texttt{Offer}, ?\texttt{Offer}, !\texttt{Cancel}, !\texttt{Request}$,
$?\texttt{Request}, ?\texttt{Cancel}, !\texttt{Offer}, ?\texttt{Offer}, !\texttt{Cancel}, ?\texttt{Cancel} >$,
$< !\texttt{Request}, ?\texttt{Request}, !\texttt{Offer}, ?\texttt{Offer}, !\texttt{Cancel}, !\texttt{Request}$,
$?\texttt{Request}, !\texttt{Offer}, ?\texttt{Offer}, ?\texttt{Cancel}, !\texttt{Order}, ?\texttt{Order} >$,
$< !\texttt{Request}, ?\texttt{Request}, !\texttt{Offer}, ?\texttt{Offer}, !\texttt{Cancel}, !\texttt{Request}$,
$?\texttt{Request}, !\texttt{Offer}, ?\texttt{Offer}, ?\texttt{Cancel}, !\texttt{Cancel}, ?\texttt{Cancel} >$,
$< !\texttt{Request}, ?\texttt{Request}, !\texttt{Offer}, ?\texttt{Offer}, !\texttt{Cancel}, !\texttt{Request}$,
$?\texttt{Request}, !\texttt{Offer}, ?\texttt{Offer}, !\texttt{Order}, ?\texttt{Order}, ?\texttt{Cancel} >$.

Step 3: Optimized local test suite. The produced test suite, incorporating all translated test cases, has to be optimized according to the test objectives described in Section 4.3. As mentioned the size of the test suite and hence the optimization effort might vary, depending on the utilized test generator.

Example 10 *An optimized test suite for the running example is given below:*

$< !\texttt{Request}, ?\texttt{Request}, !\texttt{Offer}, ?\texttt{Offer}, !\texttt{Cancel}, !\texttt{Request}$,
$?\texttt{Request}, !\texttt{Offer}, ?\texttt{Offer}, !\texttt{Order}, ?\texttt{Order}, ?\texttt{Cancel} >$,
$< !\texttt{Request}, ?\texttt{Request}, !\texttt{Offer}, ?\texttt{Offer}, !\texttt{Cancel}, !\texttt{Request}$,
$?\texttt{Request}, !\texttt{Offer}, ?\texttt{Offer}, !\texttt{Cancel}, ?\texttt{Cancel}, ?\texttt{Cancel} >$,
$< !\texttt{Request}, ?\texttt{Request}, !\texttt{Offer}, ?\texttt{Offer}, !\texttt{Cancel}, !\texttt{Request}$,
$?\texttt{Request}, ?\texttt{Cancel}, !\texttt{Offer}, ?\texttt{Offer}, !\texttt{Order}, ?\texttt{Order} >$.

Step 4: Executable test suite. The abstract test cases of the optimized test suite are translated into executable test suites. This step is semi-automatic. Test step skeletons as well as state checks using the glue

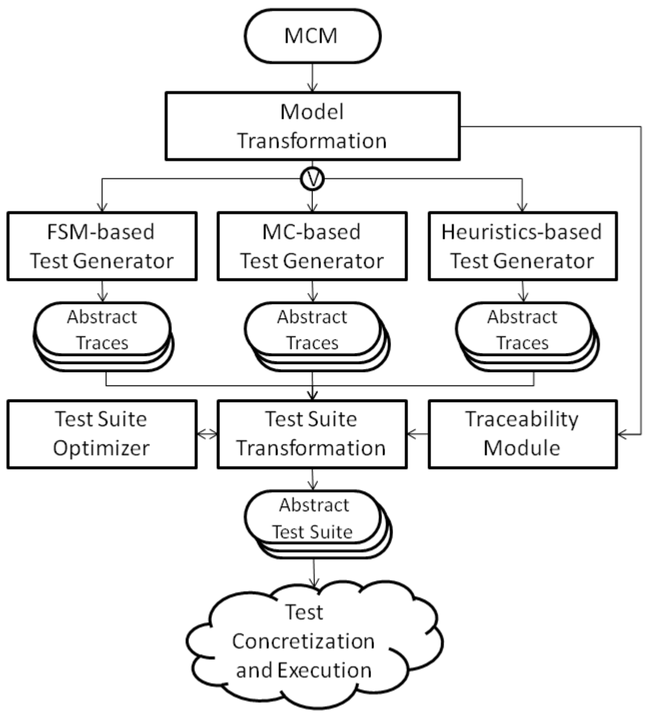


Figure 6.1: The test generation framework

information of the local components (see Example 7) can be automatically generated. As message triggers and concrete test data is not modeled in MCM, this information has to be added manually to the test sequences. Details of the test concretization and the subsequent test execution can be found in Section 4.1.

6.2 Test Generation Framework

Taking the envisioned testing process, given in the previous section, as a starting point, in the following, the test generation framework that implements this intended approach is described.

Various state-based MBT technologies and tools have been developed in the research community. Each of them has its strengths and weaknesses. The framework that has been developed for the model-based integration testing on the basis of MCM models allows to plug in different MBT test generators, in order to supply the most appropriate technique based on the actual context. All the described test generators are Eclipse-based and hence were integrated into the MCM editor similar to the Rodin tool set described in Section 5.5.

The testing framework's general layout is depicted in Figure 6.1. Below, an overview of the design and implementation of the framework is given, by describing each depicted artifact. The different, currently supported test generators are described in the subsequent Sections 6.3 to 6.5.

MCM. Prerequisite for the test generation process is the provision of MCM artifacts that are conforming to the specification given in Chapter 5.

Model Transformation. In order to supply existing advanced test generation techniques to the user of MCM, model transformations from MCM to the input format of these MBT tools are provided. In this way, the redevelopment of mature MBT implementations is avoided and the concept additionally allows future extensions. Details on the currently implemented transformations are given in the respective subsections describing the used test generators.

Test Generators. The test generators basically provide the execution of Step 1 of the previous subsection. Currently three different test generation techniques are supported: Finite state machine (FSM) based, model-checking (MC) based, and heuristics-based test generation.

While FSM-based generation is the fastest technique, it can only be applied, if the transitions of the GCM are independent of each other, i.e., no transition guard references data that can be altered by another transition. Otherwise the generated test suites may contain infeasible paths, i.e., paths violating the constraints specified in the model.

Model-checking guarantees to find an extended test suite that can be reduced to an optimal set of test cases regarding the previously introduced test objectives. However, faced with complex models it experiences the well-known state explosion problem and possibly needs infinite resources to produce a result.

To avoid unacceptably expensive computation of test suites, heuristic-based test generation can be used. By applying it, the runtime can be controlled very well. On the other hand, it cannot guarantee to produce an optimal solution.

The concrete implementations of each of the three methods are discussed in the consequent Subsections 6.3 to 6.5. The common strategy that should be applied when choosing an appropriate MBT algorithm is to use the FSM-approach in case the transitions of the GCM are independent of each other. If this is not the case, the MC-based approach may be applied. However, in cases where the MCM's complexity is too high and the MC-based approach does not terminate, a test suite could finally be obtained by using the heuristic-based approach. In Chapter 7, the three integrated test generators are compared, based on the given case study.

Abstract Traces. Though the output format of each test generator might differ, the result is generally a set of traces through the given GCM.

Test Suite Transformation. The test suite transformation has two tasks. First of all, in order to obtain a unified output in form of an abstract test suite, the given trace sets have to be transformed into an intermediate format. Secondly, Step 2 of the overall test process given in Section 6.1 has to be executed.

Test Suite Optimizer. The optimization of the test suite may take place either before or after mapping the global traces to local traces during test suite transformation. While the first option promises slightly better performance, it may not achieve minimal occurrence of message racing, which is one of the defined optimization criteria.

Traceability Module. In order to compute local sequences from the generated traces in the test suite transformation unit, a mechanism is needed that provides the link between the obtained abstract traces and their associated modeled entities.

Abstract Test Suite. The output of the test framework is an optimized abstract test suite. In order to preserve explicit links to the input model and make the test concretization process as easy as possible, it has a proprietary format.

6.3 FSM-based test generation

For the FSM-based approach, a custom solution is realized that is based on classical graph theory instead of integrating an existing test generator. The decision was made, because the necessary algorithms are well known and documented, and hence the implementation effort seemed to be lower compared with the integration effort of an external tool and the transformation effort for the input and output of the test generator. Instead, the implementation directly works on MCM instances and further generates the abstract test suite in an internal format.

The tool is realized utilizing a variation of the Rural Chinese Postman Tour, described in [SL96], that produces a minimal test suite by default. Thus, only the resulting traces through the GCM have to be mapped to the local paths only (Step 2 of previous subsection), whereas the test optimization (Step 3) can be omitted.

The applied algorithm is divided into six steps as illustrated in Figure 6.2 and is based on the classical idea of finding an Eulerian path through a graph. In the following a short description of each step is given.

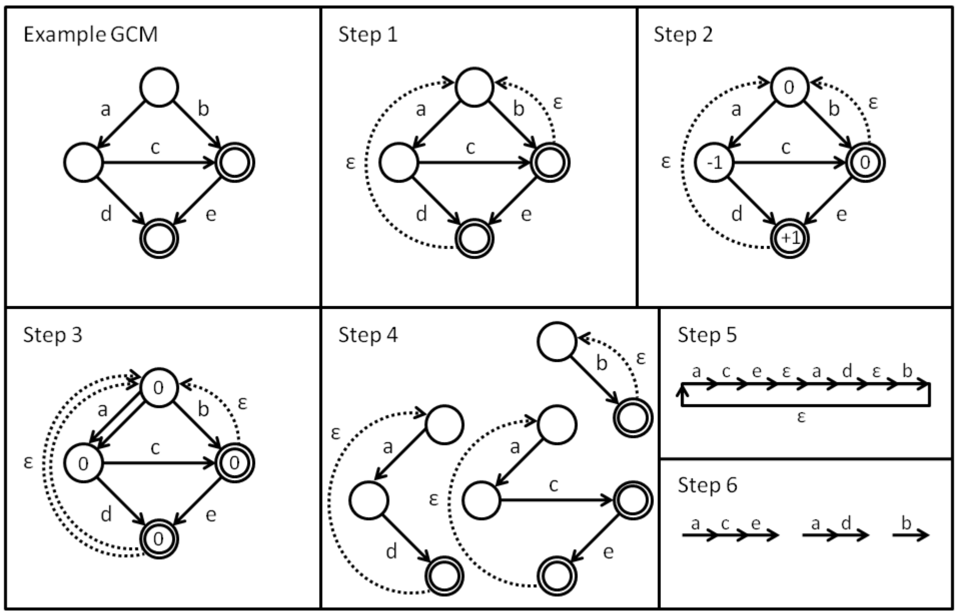


Figure 6.2: Steps of the test generation for finite state machines

- *Step 1: Connecting the graph.* Basis for the applied algorithm is a strongly connected model (i.e., each state is reachable from any state). As defined in Chapter 5, in a GCM each state is reachable from the initial state and each state reaches an accepting state. Hence, connecting the accepting states of the GCM with the initial state by adding ϵ transitions leads to a strongly connected GCM.

- *Step 2: Calculating the Eulerian weight.* The prerequisite for the existence of an Eulerian path through a directed graph is a balanced Eulerian weight for each node. The Eulerian weight for each state is calculated by adding the number of incoming transitions and subtracting the number of outgoing transitions.

- *Step 3: Balancing the Eulerian weight.* The balancing of the Eulerian weights without altering the described behavior can be obtained by duplicating existing transitions. This can be done by computing the shortest paths from states with positive weights to negative weights. ϵ transitions are neglected when calculating the length of a path.

- *Step 4: Computing the Eulerian cycles.* In a balanced graph, Eulerian cycles can be obtained by traversing the model randomly starting at any state. From this state the model is traversed randomly until the starting state is reached and thus a cycle is computed. During this random walk each traversed transition is marked, in order to avoid processing it twice. The whole procedure is repeated until all Eulerian cycles are found (i.e. every transition in the model has been marked).

- *Step 5: Computing the Eulerian path.* In order to compute the Eulerian path, the Eulerian cycles are combined pairwise by inserting one cycle into another (using a state that they both traverse) until all initial cycles are incorporated in one (Eulerian) cycle covering the whole model. By splitting this cycle once in the initial state, the Eulerian path is obtained.

- *Step 6: Computing the test suite.* Finally the obtained path is split into the abstract test cases by deleting the previously added ϵ transitions and collecting the resulting traces into a test suite.

As explained, the disadvantage of the FSM-based approach is that it does not take the transition constraints into account, which may lead to the generation of infeasible paths (i.e. test cases that are violating the transition constraints and hence cannot be executed). Considering the example from Section 3.1, this disadvantage becomes evident.

Example 11 *Applying the FSM-based test generator to the GCM of the running example (see Figure 4.5), one of the generated test suites*[3] *contains the following test cases:*

$$< \texttt{Request}, \texttt{Cancel}, \texttt{Offer}, \texttt{Order}, \texttt{Cancel} >,$$
$$< \texttt{Request}, \texttt{Offer}, \texttt{Cancel}, \texttt{Cancel} > .$$

Both test cases are violating the annotated constraints in the GCM of the given example. In the first test case, the with 0 *initialized counter variable is left unchanged by the first* `Request`*. In state* `Requested`*, which is reached subsequently, the* `Cancel` *transition demands a counter value to be greater than* 0 *and is therefore not reachable. In the second test case, the counter is again* 0 *after the* `Request`*. The subsequent* `Offer` *sets it to* 1*. Consequently, in the next state (*`Reserved`*) only the* `Cancel` *that leads to the state* `Start` *is active. However, in* `Start` *the second* `Cancel` *cannot be triggered. Both of the given test cases therefore are infeasible.*

To tackle this problem, an MC-based approach is deployed as described in the next section.

[3]Because there is more than one optimal solution, the generated test suite depends on the order in which the transitions are explored by the algorithm.

6.4 Model-checking based test generation

Model checking has been utilized for test generation in various approaches [FG09]. The current version of the framework incorporates a test generator based on the model checker ProB [LB08]. ProB is a validation tool set originally written for the B method. Its automated animation facilities allow users to animate and model-check their specifications which are valuable capabilities in the development of formal specifications. ProB has been adapted to support a number of formalisms such as Z, CSP, and CSP||B [BL05].

As ProB currently uses Event-B [AH07] as input language, it was necessary to provide a model transformation for MCM. For the details of this transformation please refer to [9].

The test generation algorithm that has been developed for the integration testing approach based on MCM is separated into three steps, which are corresponding to the steps of the general approach from Section 6.1. In the following for each step details about the implementation are given and the computed results for the running example from Section 3.1 are shown.

Step 1: Generation of an Initial Global Test Suite. As explained in Section 4.3, the test generation is to cover each transition of the global communication model, i.e., each interaction of the GCM. To satisfy the first and second objective given at the end of Subsection 4.3, ProB was extended to detect when transition coverage is obtained. This is gained by exploring the state space of the model, using a breadth-first strategy (corresponding to the second objective), stopping when full coverage is achieved by discovered, terminating traces(corresponding to the first objective).

The recognition of this termination has been ensured by adding a *history* variable to the GCM, storing the set of executed interactions and adding a corresponding end-interaction for every original interaction i, which can be triggered if being in a valid end state and if $i \in history$. Afterwards all traces that end in an accepting state are extracted from the explored state space to form the initial test suite.

Example 12 *From the example given in Section 3.1, the following initial set of test cases is obtained:*

$<$ `Request`, `Offer`, `Order` $>$,
$<$ `Request`, `Offer`, `Cancel` $>$,
$<$ `Request`, `Offer`, `Cancel`, `Request`, `Offer`, `Order` $>$,
$<$ `Request`, `Offer`, `Cancel`, `Request`, `Offer`, `Cancel` $>$,
$<$ `Request`, `Offer`, `Request`, `Cancel`, `Offer`, `Order` $>$,
$<$ `Request`, `Offer`, `Request`, `Cancel`, `Offer`, `Cancel` $>$,
$<$ `Request`, `Offer`, `Request`, `Offer`, `Order` $>$,
$<$ `Request`, `Offer`, `Request`, `Offer`, `Order`, `Cancel` $>$,
$<$ `Request`, `Offer`, `Request`, `Offer`, `Cancel` $>$,
$<$ `Request`, `Offer`, `Request`, `Offer`, `Cancel`, `Order` $>$,
$<$ `Request`, `Offer`, `Request`, `Offer`, `Cancel`, `Cancel` $>$.

Step 2: Mapping of Global to Local Paths. In order to obtain locally executable test cases, the global sequence of message observations for each path has to be mapped to the corresponding send and receive events of partners. As the GCM uses the receive semantics, the global observe sequences can be directly translated to sequences of receive events. Afterwards for each receive event a corresponding send event is generated and added to the path in such a way that the local behavior descriptions are not violated (see the corresponding *Step 2* of Section 6.1).

In most cases it is infeasible to exhaustively explore the full state space (as the state space of the CS is actually even considerably bigger) to find a suitable mapping from global to local traces. The problem could be encoded as an LTL formula, but this formula would be very big, with ensuing consequences for the complexity of model checking. A better solution, is to encode the desired LPM traces into a CSP [Hoa78] process. This process is synchronized with the Event-B model, using the technology of [BL05], suitably guiding the model checker.

The CSP Process is divided into two components. The first process encodes the desired trace of receive events, followed by an event on the goal channel, indicating to the model checker that this is a desired goal state.

Example 13 *For the last trace given above, it looks as follows:*

$$\begin{aligned} RECEIVER \;=\; & Seller?\texttt{Request} \rightarrow Buyer?\texttt{Offer} \rightarrow Seller?\texttt{Request} \\ & \rightarrow Buyer?\texttt{Offer} \rightarrow Seller?\texttt{Cancel} \rightarrow Seller?\texttt{Cancel} \\ & \rightarrow goal \rightarrow STOP. \end{aligned}$$

The second process encodes the sender events. While the number of send events for each event type is equal to the number of receive events, the order of them is unknown.

$$\begin{aligned}
&SENDER(n_1, n_2, n_3, n_4) \quad = \\
&\qquad n_1 > 0 \quad \& \quad Buyer!\texttt{Request} \rightarrow SENDER(n_1 - 1, n_2, n_3, n_4)\,[\,] \\
&\qquad n_2 > 0 \quad \& \quad Seller!\texttt{Confirm} \rightarrow SENDER(n_1, n_2 - 1, n_3, n_4)\,[\,] \\
&\qquad n_3 > 0 \quad \& \quad Buyer!\texttt{Cancel} \rightarrow SENDER(n_1, n_2, n_3 - 1, n_4)\,[\,] \\
&\qquad n_4 > 0 \quad \& \quad Buyer!\texttt{Order} \rightarrow SENDER(n_1, n_2, n_3, n_4 - 1).
\end{aligned}$$

The sender process is now interleaved with the receiver process.

$$MAIN \quad = \quad SENDER(2, 2, 2, 0) \;|||\; RECEIVER.$$

In that way, ProB will ensure that every event of the Event-B model synchronizes with an event of the CSP process (MAIN), guiding it and stopping when the CSP process can perform an event on the goal channel.

Step 3: Test Suite Reduction. Of course the resulting test suite has to be optimized. While the first and second objectives given in Section 4.3 have been satisfied by design, the reduction of test suite is still necessary. This can be achieved by using a greedy search or even a brute force algorithm that computes every possible combination of test cases and selects the optimal one according to the given objectives.

Example 14 *For the given initial test suite, an optimal test suite would incorporate the local equivalents of the following global paths:*

$$\begin{aligned}
&< \texttt{Request}, \texttt{Offer}, \texttt{Request}, \texttt{Offer}, \texttt{Order}, \texttt{Cancel} >, \\
&< \texttt{Request}, \texttt{Offer}, \texttt{Request}, \texttt{Offer}, \texttt{Cancel}, \texttt{Cancel} >, \\
&< \texttt{Request}, \texttt{Offer}, \texttt{Request}, \texttt{Cancel}, \texttt{Offer}, \texttt{Order} > .
\end{aligned}$$

6.5 Heuristics-based test generation

In cases where the FSM-based approach cannot be applied (because of interdependent interactions) and also the MC-based approach is infeasible (because of state explosion), still a possibility has to be supplied that allows generation of feasible test cases for covering as much of the GCM model

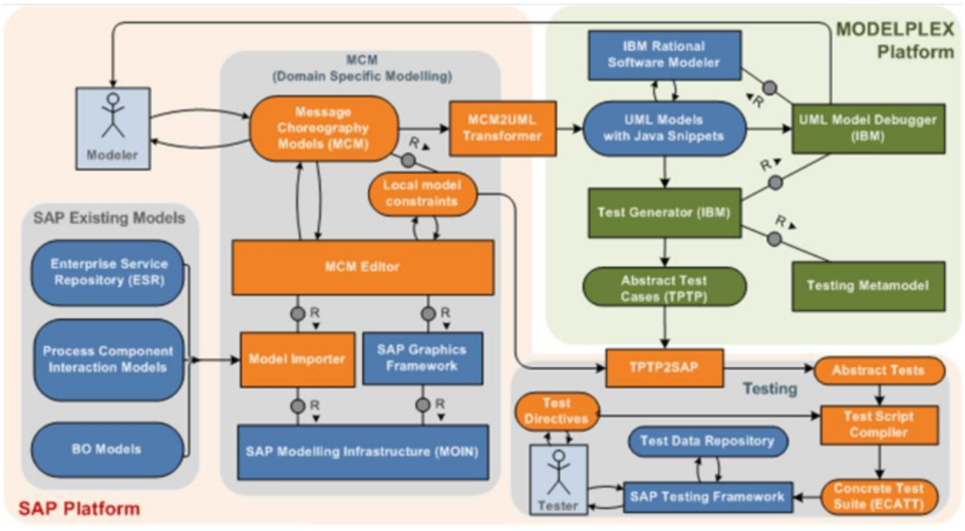


Figure 6.3: Integration of the IBM test generator

as possible. Therefore an engine is needed, which is able to execute randomly chosen (but guided) paths in the state space and record the input and expected outputs, according to the model.

Some experiments have been conducted with the UML model execution and test generation engines provided by the IBM research prototype [DK07], which is integrated into IBM Rational Software Modeler[4]. The prototype tool allows the execution of UML 2.0 models that are annotated with Java code on the transitions. In order to utilize the given tools, a transformation from MCM to UML with Java annotations had to be realized.

Figure 6.3 depicts the tool integration of the IBM prototype, which is part of the MODELPLEX platform[5]. In the middle, the MCM module including the MCM editor and a model importer from SAP's existing models is shown. The connection between MCM and UML is made via the MCM2UML and the TPTP2SAP transformer modules. Roughly explained, the transformation works as follows. The message types and service components are transfered into UML class and component diagrams, and the GCM is translated into a UML state machine, where the interaction constraints and side effects are implemented using Java snippets. More details on the transformation can be found in [10].

IBM's heuristics-based test generator utilizes the model execution engine by sending inputs and observing outputs. The inputs are recorded as test sequences of stimuli applied to the SUT while the outputs are also recorded as expected outcomes. In other words, the model is used as a test oracle

[4]http://www.ibm.com/software/awdtools/modeler/swmodeler

[5]http://www.modelplex-platform.com

predicting the correct behavior, while the test generator heuristically tries to cover for instance the inputs of different available services. The main advantage of the tool is that it actually executes the model and therefore is able to determine all enabled transitions at each state for a given context on-the-fly. Thus the generated paths are always feasible. However, this approach cannot guarantee complete transition coverage, since it applies some sort of guided randomness and further does not explicitly allow to target transitions.

Unfortunately, the heuristics that the IBM test generator is using, are quite premature. Because focus of the prototype was to comply with the UML standard instead of advanced heuristics, the choices of the tool are taken randomly at the moment.

Example 15 *To evaluate the prototype on the example given in Section 3.1, the test generator is asked for a test suite that covers all transitions or terminates after* 100 *steps. The following results can be obtained after* 10 *runs.*

- 9 *runs terminated, because a test suite with full transition coverage was found,* 1 *run terminated with* 87% *transition coverage.*
- *The average test suite contained* 47 *test steps, distributed over* 6 *to* 7 *test cases.*
- *The average execution time to generate a test suite was* 3 *seconds.*
- *After optimization, the average test suite contained* 23 *test steps, distributed over* 2.7 *test cases.*

The experiment above illustrates the main advantage of the heuristics-based test generation. It terminates and delivers a result, even if the test goal (here transition coverage) cannot be reached with reasonable effort. This is especially valuable, if the model contains unreachable transitions that would force an MC-based test generator to infinitely explore the state space without returning any result.

The output of the tool is a test suite in the Eclipse Test & Performance Tools Platform[6] (TPTP) format. Therefore, as explained in Section 6.1, further a transformation mechanism into SAP's internal test suite format had to be provided.

In general, the transformation to UML opens up the usage of other UML-based MBT tools (see a recent evaluation of such tools in [GNRS09]) like the Test Designer from Smartesting[7]. However, the current translation takes into account the special semantics of the executable UML models as

[6]`http://www.eclipse.org/tptp`

[7]`http://www.smartesting.com`

supported by the IBM tool. Other tools may use different UML semantics and also OCL as action language instead of Java, implying that adjusting the current transformation for different UML tools might not be trivial.

6.6 Summary

Chapter 4 introduced a model-driven SOA development approach and motivated the application of MBT to service integration testing based on choreography models. In this chapter, the necessary test generation process and its realization has been described.

As laid out in Chapter 3, the focus of the thesis is to derive test case for service integration testing from choreography models, while the generation of test data that is needed for the execution of those test cases was kept out of scope. In Section 4.3, the necessary conceptual decisions (i.e. the required coverage criterion and optimization parameters for the test suite) for utilizing MBT in the service integration generation have been described.

Section 6.1 stepwise explained the envisioned test generation process that derives an abstract test suite from the MCM modeling artifacts according to the mentioned conceptual decisions. The concretization and execution of test cases has been described in Section 4.3. The testing framework that has been created to enable the previously described automatic test generation process was introduced in Section 6.2. Its main purpose is to provide multiple alternative state of the art test generators that can be chosen according to the structure of the choreography model. The currently supported MBT tools were introduced in the Sections 6.3, 6.4, and 6.4. A comparison of these tools is provided as part of the case study, described in Chapter 7.

Chapter 7

Case Study

This chapter describes a case study that has been conducted to evaluate the developed modeling and model-based testing solution. It exemplifies the approach presented in earlier chapters. Chapter 4 described the SOA development approach that defines the context for the scientific contributions of the thesis and explained the core concepts of modeling and model-based testing of service choreographies. Chapter 5 introduced the proprietary language MCM that implements the given choreography modeling concepts. The testing framework that realizes the model-based service integration testing has been given in Chapter 6.

Aim of this case study was to gain evidence that the described SOA development approach based on MCM is applicable in an industrial setting. Further, its efficiency and effectiveness was investigated.

Section 7.1 gives information about the design and context of the case study. The case study execution, including the modeling and test generation, is described in Section 7.2. Section 7.3 explains the conclusions that can be deduced from the case study execution.

7.1 Case Study Design

In this section, the context of the conducted case study is given. It starts with a description of the context in which the case study is deployed. Afterwards the planned case study activities are given.

7.1.1 Setting

As described in Chapter 1, the dissertation 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],

[1]`http://www.sap.com/sme/solutions/businessmanagement/businessbydesign/index.epx`

SAP Business Suite[2]) and its SOA-based, open technology platform SAP NetWeaver [HKB+08]. In the following, the system, which is subject of the case study, is described. Afterwards, the users that were conducting the cases study are characterized.

System under Test The concepts that are described in this thesis have been developed in collaboration with the SAP product group that is responsible for the development of the SOA-based solution SAP Business ByDesign. It is created on top of SAP NetWeaver, which provides a SOA technology platform including a messaging infrastructure [KO09].

At the time the case study was conducted, Business ByDesign was quite mature in terms of functionality and quality and already released to selected customers. Now, Business ByDesign is freely available in the United States, Germany, France, the United Kingdom, China, and India.

The design of SAP Business ByDesign has been captured by various modeling artifacts, based on proprietary SAP languages [KP09]. For customers, high level behavior descriptions of the supported business processes exist in form of use case scenarios. 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 an documentation activities.

Case Study Participants Two groups of users have been involved in the conducted case study. In the following, their background and relevant qualification is given.

- *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 knowledge about MBT.

[2]`http://www.sap.com/solutions/business-suite/index.epx`

7.1.2 Approach

In the initial planning of the case study, four use cases were identified that were intended to follow the whole service integration process, as described in Chapter 4, 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.

In the following the modeling and testing activities of the case study are described.

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 would conduct 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 PhD candidate. Afterwards the models were planned to be validated by the pilot users in *another guided session of* 1 *hour*.

Testing. For each of the four use cases, one test suite was generated and concretized. Two times the FSM based and two times the model checking based test generator have been used. 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 messages.

The generated abstract test cases were used to automatically generate and load test scripts into the test environment of the development teams. Further, the corresponding MCM model and automatically generated UML message sequence charts for each test case have been supplied for each pilot. The testers were asked to concretize the generated test scripts autonomously and to execute them on the system under test.

Analysis As described, 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 PhD candidate allowed to gather some unbiased observations (e.g. execution time of the test generation, number of uncovered faults).

2. Participants were questioned in interviews. The semistructured approach was chosen, because this interview technique is mixing open-ended and specific questions and should be chosen according to [Sea99], if the responses might lead to further discussions that better captures information of unknown structure.

7.2 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.

7.2.1 Modeling

According to the plan described in Section 7.1, the creation of the pilot models was conducted in 2 *guided sessions that lasted about* 1 *hour*. The refinement and consolidation that was conducted by the PhD candidate in between accounts for *another* 2 *hours*.

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 GCM's receive semantics as well as the proprietary constraint language 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.

7.2.2 Test Generation

As described in Section 7.1, for each pilot an abstract test suite has been generated automatically. 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 suites was estimated to be* *between around* 4 *hours.* The consequent test execution did not uncover any fault in the development system[3].

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.

7.2.3 Comparison of Test Generators

As mentioned, the identification of the four suitable pilots out of about 200 existing service component pairs 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 (as explained in Section 6.2) and hence were suited for FSM-based test generation. In the following, one such pilot is used, as it allows to compare all available test generators from Chapter 6.

Figure 7.1 shows the anonymized GCM of the pilot that is chosen for the description of the case study execution. 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 task management, as implied by the naming of the modeling elements. In the following, the results of applying the different test generators from Chapter 6 are described.

FSM-based generation. Applying the FSM-based test generator (described in Section 6.3) to the given pilot, the following test cases are derived from the GCM. The computation is carried out in *less than* 1 *millisecond.*

$$\begin{aligned}&< \texttt{Create}, \texttt{Stop}, \texttt{Revoke}, \texttt{Release}, \texttt{Close}, \texttt{Restart}, \texttt{Block},\\ &\qquad \texttt{Unblock}, \texttt{Close} >,\\ &< \texttt{Create}, \texttt{Change}, \texttt{Delete} > .\end{aligned}$$

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

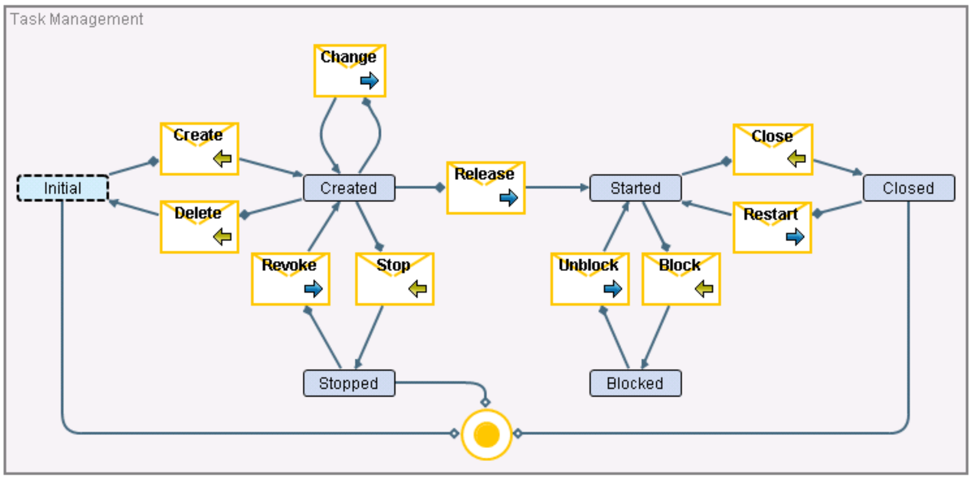


Figure 7.1: The GCM of an example pilot

As explained in Section 6.3, the implemented algorithm provides a test suite that covers the model with the minimal number of transitions. Therefore, a further optimization according to the objectives given in Section 4.3 is not possible. Consequently, the risk of applying this FSM-based test generation is that the generated test suite contains few but long test cases. The given test suite indicates this possible effect. As described in Section 4.3 long test cases are much harder to maintain and to debug in case of errors. As described in Section 6.3, FSM-based generation may generate infeasible test cases when applied to choreography models with dependent transitions. Therefore its use is limited.

MC-based generation. In contrast to the FSM-based test generator, the MC-based test generator (described in Section 6.4) provides a test suite that is optimized according to the objectives given in Section 4.3. The resulting test suite for the given pilot is given below and has been *produced in about* 0.4 *seconds*.

$<$ `Create`, `Release`, `Block`, `Unblock`, `Close` $>$,
$<$ `Create`, `Release`, `Close`, `Restart`, `Close` $>$,
$<$ `Create`, `Change`, `Stop`, `Revoke`, `Delete` $>$,

As a consequence of the applied optimization, 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, as described in Section 6.4. 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.

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 (similar to the example in Section 6.5), 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, as stated in Section 6.5, 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. As explained in Chapter 6, each test generation method has its strengths and weaknesses in certain context. The comparison of these methods, as presented above, was further able to confirm the proposed strategy for choosing between them: "Use the FSM-approach in case the transitions of the GCM are independent of each other. If this is not the case, the MC-based approach may be applied. However, in cases where the MCM's complexity is too high and the MC-based approach does not terminate, a test suite could finally be obtained by using the heuristic-based approach." As shown, in the third case optimal results cannot be guaranteed.

7.3 Case Study Evaluation

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 7.2, there has been positive feedback regarding the automated test generation and the utilized reuse concept of test scripts (described in Section 4.1). The time saving potential of the reuse concept was further demonstrated when concretizing all the generated test suites from Section 7.2.3. 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 mis-interpretations 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. However, these statements indicate that the testers greatly believe in the effectiveness of the approach.

7.4 Summary

In this chapter, a case study for the modeling and model-based testing of service choreographies was described. It was possible to show that the MCM-based approach for model-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, from the case study it was deduced that the approach saves resources, provides controlled coverage, and increases confidence in the implementation. Nevertheless, in order to compare the fault uncovering capabilities with the current testing strategies, additional evaluation has to be conducted in the future. Additionally, the integrated MBT tools of the testing framework have been compared, illustrated on one of the use cases.

Chapter 8

Conclusion

8.1 Summary

Applying a sound service integration strategy is a key factor for successful SOA projects. Aim of the thesis was to provide such a strategy, which is also concrete enough to be applicable in practice.

Therefore, in Chapter 3 the key challenges of SOA integration have been explained. Further, requirements for an approach that is able to answer these challenges were derived. A concrete approach that utilizes modeling of service choreographies and consequent model-based testing was introduced in Chapter 4.

While various mature techniques and tools exist that support model-driven software development including model-based testing, in the area of SOA service integration a modeling notation that allows to leverage such techniques was missing. Consequently, in this thesis the choreography modeling language MCM has been described in Chapter 5, which was developed driven by the specific requirements of a comprehensive quality control. This domain specific language is based on the unique concept of choreography viewpoints and supported by an Eclipse-based editor that integrates a rich tool set of state-of-the-art static verification techniques to ensure the creation of consistent and semantically correct models.

To enable the utilization of various existing model-based testing technologies according to their specific strengths, important considerations regarding the objectives and approach of service integration testing have been discussed. Consequently, a test generation framework has been provided in Chapter 6 that provides access to a set of complementary test generation tools. The framework is further integrated into the testing landscape at SAP.

In Chapter 7, a case study for the MCM-based SOA service integration has been described. The portrayed process incorporated the modeling of service choreographies as well as the test generation and execution in an industrial setting. In particular it could be shown that a SOA development process, incorporating behavioral specifications for model-based testing can be instrumented to guide and improve current industrial practice.

8.2 Outlook

Despite the successful derivation of test suites from service choreography models, the provision of appropriate test data remains a major challenge that has been solved rudimentary by leveraging the testers expert knowledge. Consequently, to explore the potential of improvements in the area of test data provisioning, which is with about 50% of the overall testing effort, by far the most time consuming test concretization activity, should therefore be the greatest target for consequent research.

As described in [2], the test data for enterprise applications is subject to various constraints. These are ranging from dependencies between different data inputs of a test case (e.g. during a test run an address string that has to be given in a test step may have to be conforming to the regulations for a previously chosen country) to dependencies between test data and system data (e.g. the given test input data has to reference a material with special attributes). Incorporating automatic test data generation into the described approach will therefore require a set of additional information and techniques like the formalization and specification of the mentioned data constraints, the supply and specification of system data access during a test run, and the utilization of advanced test data provisioning as well as online testing techniques.

Apart from this fundamental test data challenge, another aim is to further improve the current prototype. There are various missing features that have to be provided in order to transform the developed modeling tool and testing framework into a mature product. Such features are for example version control and change management of MCM models and the storage of modeled content inside a model repository. Further, the partial derivation of MCM models from existing modeling content as well as code can be tackled to ease the currently manual task.

References

[AAF+04] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A. Ziv. Genesys-pro: Innovations in test program generation for functional processor verification. *IEEE Design & Test of Computers*, 21(2):84–93, 2004.

[ABHV08] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, and Laurent Voisin. A roadmap for the rodin toolset. In *Proceedings of Abstract State Machines, B and Z*, page 347, 2008.

[Abr96] J.R. Abrial. *The B-book.* Cambridge University Press, 1996.

[ABR+07] Shaukat Ali, Lionel C. Briand, Muhammad Jaffar-Ur Rehman, Hajra Asghar, Muhammad Zohaib Z. Iqbal, and Aamer Nadeem. A state-based approach to integration testing based on UML models. *Information & Software Technology*, 49(11-12):1087–1106, 2007.

[AD97] Larry Apfelbaum and John Doyle. Model-based testing. In *Proceedings of the 10th International Software Quality Week (QW97)*, 1997.

[AGA+08] A. Arsanjani, S. Ghosh, A. Allam, T. Abdollah, S. Ganapathy, and K. Holley. SOMA: A method for developing service-oriented solutions. *IBM Systems Journal*, 47(3):377–396, 2008.

[AH07] Jean-Raymond Abrial and Stefan Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to Event-B. *Fundamenta Informaticae*, 77(1-2):1–28, 2007.

[AJ02] S.W. Ambler and R. Jeffries. *Agile modeling: effective practices for extreme programming and the unified process.* Wiley, 2002.

[AKK$^+$05] Manoj Acharya, Abhijit Kulkarni, Rajesh Kuppili, Rohit Mani, Nitin More, Srinivas Narayanan, Parthiv Patel, Kenneth Schuelke, and Subbu Subramanian. SOA in the real world - experiences. In *Service-Oriented Computing (ICSOC)*, volume 3826, pages 437–449, 2005.

[BA04] K. Beck and C. Andres. *Extreme programming explained: embrace change.* Addison-Wesley Professional, 2004.

[BAHS07] D. Brenner, C. Atkinson, O. Hummel, and D. Stoll. Strategies for the run-time testing of third party web services. In *IEEE International Conference on Service-Oriented Computing and Applications*, pages 114–121, 2007.

[BBJ$^+$03] Paul Baker, Paul Bristow, Clive Jervis, David King, and Bill Mitchell. Automatic generation of conformance tests from message sequence charts. *Telecommunications and beyond: The BroaderApplicability of SDL and MSC*, 2599/2003:170–198, 2003.

[BBMP08] Cesare Bartolini, Antonia Bertolino, Eda Marchetti, and Andrea Polini. Towards automated WSDL-based testing of web services. In *Int. Conf. on Service-oriented Computing (ICSOC'08)*, volume 5364 of *LNCS*, pages 524–529. Springer, 2008.

[BCZ05] D.M. Berry, BH Cheng, and J. Zhang. The four levels of requirements engineering for and in dynamic adaptive systems. In *11th International Workshop on Requirements Engineering Foundation for Software Quality (REFSQ)*, 2005.

[BD07] Luciano Baresi and Elisabetta Di Nitto. *Test and Analysis of Web Services.* Springer, 2007.

[BDG$^+$08] Paul Baker, Zhen Ru Dai, Jens Jens Grabowski, Øystein Haugen, Ina Schieferdecker, and Clay Williams. *Model-Driven Testing: Using the UML Testing Profile.* Springer, 2008.

[BDO05] Alistair Barros, Marlon Dumas, and Phillipa Oaks. A critical overview of the web services choreography description language. *Business Process Trends White Paper*, 2005.

[BHB$^+$07] Abbie Barbir, Chris Hobbs, Elisa Bertino, Frederick Hirsch, and Lorenzo Martino. Challenges of testing web services and security in SOA implementations. In *Test and Analysis of Web Services*, pages 395–440. Springer, 2007.

[Bin99] Robert Binder. *Testing object-oriented systems: models, patterns, and tools.* Addison-Wesley Professional, 1999.

[BJL+05] F. Bouquet, E. Jaffuel, B. Legeard, F. Peureux, and M. Utting. Requirements traceability in automated test generation: application to smart card software validation. In *A-MOST '05: Proceedings of the 1st international workshop on Advances in model-based testing*, pages 1–7. ACM, 2005.

[BJLP05] A. Brown, S.K. Johnston, G. Larsen, and J. Palistrant. SOA development using the ibm rational software development platform: a practical guide. *Rational Software*, 2005.

[BJPW01] Hans Buwalda, Dennis Janssen, Iris Pinkster, and Paul Watters. *Integrated test design and automation: Using the Test-frame method.* Addison-Wesley Professional, 2001.

[BL05] Michael Butler and Michael Leuschel. Combining CSP and b for specification and property verification. In *In Proceedings of Formal Methods*, pages 221–236. Springer, 2005.

[But09] M. Butler. Decomposition structures for Event-B. In *Proceedings of Integrated Formal Methods iFM2009*, volume 5423 of *LNCS*. Springer, 2009.

[CDPP96] D.M. Cohen, S.R. Dalal, J. Parelius, and G.C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, 13(5):83–88, 1996.

[CGK+02] Francisco Curbera, Yaron Goland, Johannes Klein, Frank Leymann, Dieter Roller, Satish Thatte, and Sanjiva Weerawarana. Business Process Execution Language for Web Services (BPEL4WS). `http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel1.pdf`, 2002. Version 1.0.

[Coc07] A. Cockburn. *Agile software development: the cooperative game.* Addison-Wesley, 2007.

[CP09] Gerardo Canfora and Massimiliano Di Penta. Service-oriented architectures testing: A survey. In *Software Engineering: International Summer Schools, ISSSE 2006-2008, Revised Tutorial Lectures*, pages 78–105. Springer-Verlag, 2009.

[CSH03] Ian Craggs, Manolis Sardis, and Thierry Heuillard. AGEDIS case studies: Model-based testing in industry. In *Proceedings of the 1st European Conference on Model Driven SoftwareEngineering*, pages 129–132. Imbus AG, 2003. Online at: http://www.agedis.de/documents/AGEDIS in Industry.PDF.

[DD04] Remco M. Dijkman and Marlon Dumas. Service-oriented design: A multi-viewpoint approach. *International Journal of Cooperative Information Systems*, 13(4):337–368, 2004.

[DDB^{+}05] Zhen Ru Dai, Peter H. Deussen, Maik Busch, Laurette Pianta Lacmene, Titus Ngwangwen, Jens Herrmann, and Michael Schmidt. Automatic test data generation for TTCN-3 using CTE. In *ICSSEA: International Conference Software & Systems Engineering and their Applications*, 2005.

[DGR04] N. Delgado, A.Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering*, pages 859–872, 2004.

[DK07] Dolev Dotan and Andrei Kirshin. Debugging and testing behavioral UML models. In *OOPSLA Companion 2007*, pages 838–839. ACM, 2007.

[DKLW07] Gero Decker, Oliver Kopp, Frank Leymann, and Mathias Weske. BPEL4Chor: Extending BPEL for modeling choreographies. In *IEEE International Conference on Web Services (ICWS 2007)*, pages 296–303. IEEE Computer Society, 2007.

[DND09] E. Di Nitto and S. Dustdar. Principles of engineering service oriented systems. In *Proceedings of the 31st International Conference on Software Engineering*, pages 461–462. IEEE Computer Society, 2009.

[DO91] Richard A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991.

[DS05] S. Dustdar and W. Schreiner. A survey on web services composition. *International Journal of Web and Grid Services*, 1(1):1–30, 2005.

[DSU90] A.T. Dahbura, K.K. Sabnani, and M.U. Uyar. Formal methods for generating protocol conformance test sequences. *Proceedings of the IEEE*, 78(8):1317–1326, Aug 1990.

[DW07] Gero Decker and Mathias Weske. Local enforceability in interaction Petri Nets. In *Proceedings of the 5th International Conference on Business Process Management (BPM'07)*, volume 4714 of *Lecture Notes in Computer Science*, pages 305–319. Springer, 2007.

[DYZ06] Wen-Li Dong, Hang Yu, and Yu-Bing Zhang. Testing BPEL-based web service composition using High-level Petri Nets. In *EDOC'06*, pages 441–444. IEEE Computer Society, 2006.

[EAA$^+$04] M. Endrei, J. Ang, A. Arsanjani, S. Chua, P. Comte, P. Krogdahl, M. Luo, and T. Newling. *Patterns: service-oriented architecture and web services.* IBM Corp., International Technical Support Organization, 2004.

[Edv99] Jon Edvardsson. A survey on automatic test data generation. In *Proceedings of the Second Conference on Computer Science and Engineering in Linkping*, pages 21–28. ECSEL, October 1999.

[EKRV06] Juhan-P. Ernits, Andres Kull, Kullo Raiend, and Jri Vain. Generating Tests from EFSM Models Using Guided Model Checking and Iterated Search Refinement. *Formal Approaches to Software Testing and Runtime Verification*, 4262:85–99, 2006.

[ETS95] ETSI. ETS 300 406: Methods for testing and Specification (MTS), 1995.

[FBS04] X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *Proceedings of the 13th International Conference on World Wide Web*, pages 621–630. ACM, 2004.

[FG09] Gordon Fraser and Angelo Gargantini. An evaluation of specification based test generation techniques using model checkers. In *Proc. of Testing: Academic & Industrial Conference - Practice and research techniques (TAICPART'09)*, pages 72–81. IEEE Computer Society, 2009.

[FHP02] E. Farchi, A. Hartman, and S.S. Pinter. Using a model-based test generator to test for standard conformance. *IBM systems journal*, 41(1):89–110, 2002.

[FK96] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(1):63–86, 1996.

[For08] Forrester. Enterprise and SMB software survey, North America And Europe, Q4 2008. Business data service survey, Forrester Research, 2008.

[FR07] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *Proceedings of the International Conference on the Future of Software Engineering (FOSE'07)*, pages 37–54. IEEE Computer Society, 2007.

[FTdV06] Lars Frantzen, Jan Tretmans, and Rene de Vries. Towards model-based testing of web services. In Antonia Bertolino and Andrea Polini, editors, *International Workshop on Web Services Modeling and Testing (WS-MaTe2006)*, pages 67–82, Palermo, Sicily, ITALY, June 9th 2006.

[FTW$^+$06] L. Frantzen, J. Tretmans, TAC Willemse, K. Havelund, M. Nunez, G. Rosu, and B. Wolff. A Symbolic Framework for Model-Based Testing. *Formal approaches to software testing and runtime verification*, page 40, 2006.

[GBL07] V. Garousi, L.C. Briand, and Y. Labiche. Traffic-aware stress testing of distributed real-time systems based on UML models using genetic algorithms. *The Journal of Systems & Software*, 2007.

[GGN09] Michaela Greiler, Hans-Gerhard Gross, and Khalid Adam Nasr. Runtime integration and testing for highly dynamic service oriented ict solutions. In *Proc. of Testing: Academic & Industrial Conference - Practice and research techniques (TAICPART'09)*. IEEE Computer Society, 2009.

[GMS98] Neelam Gupta, Aditya P. Mathur, and Mary Lou Soffa. Automated test data generation using an iterative relaxation method. In *SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 231–244, New York, NY, USA, 1998. ACM Press.

[GNRS09] Helmut Götz, Markus Nickolaus, Thomas Roßner, and Knut Salomon. *Modellbasiertes Testen - Modellierung und Generierung von Tests Grundlagen, Kriterien fr Werkzeugeinsatz, Werkzeuge in der Übersicht*. Heise Zeitschriften Verlag, March 2009.

[GOC06] Leonard Gallagher, Jeff Offutt, and Anthony Cincotta. Integration testing of object-oriented components using finite state machines. *Softw. Test., Verif. Reliab.*, 16(4):215–266, 2006.

[GTW03] J. Gao, H.S.J. Tsao, and Y. Wu. *Testing and quality assurance for component-based software.* Artech House on Demand, 2003.

[GVV08] D. Graham and E. Van Veenendaal. *Foundations of Software Testing: ISTQB Certification.* Thomson Learning Emea, 2008.

[Har08] Mark Harman. Open problems in testability transformation. In *Proceedings of IEEE International Conference on Software Testing (ICST)*, pages 196–209. IEEE Computer Society, 2008.

[Hef09] Randy Heffner. Across all vertical industry groups, the majority of SOA users are expanding its use. Research report, Forrester Research, May 2009.

[HHH$^+$04] Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, 2004.

[HKB$^+$08] Loren Heilig, Steffen Karch, Oliver Bttcher, Christophe Mutzig, Jan Weber, and Roland Pfennig. *SAP NetWeaver: The Official Guide.* Galileo Press, 2008.

[HLT07] Markus Helfen, Michael Lauer, and Hans Martin Trauthwein. *Testing SAP Solutions.* SAP Press, 2007.

[HN04] A. Hartman and K. Nagin. The AGEDIS tools for model based testing. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 129–132. ACM New York, NY, USA, 2004.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[HT06] B. Hailpern and P. Tarr. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal*, 45(3):451–461, 2006.

[HU69] J.E. Hopcroft and J.D. Ullman. *Formal languages and their relation to automata.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1969.

[ISO01] ISO/IEC. TR 9126 Software engineering - Product quality. `http://www.iso.org`, 2001.

[Jor08] Paul C. Jorgensen. *Software Testing: A Craftsman's Approach, Third Edition.* AUERBACH, 2008.

[Kin76] J.C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):394, 1976.

[KKK+06] C. Keum, S. Kang, I. Ko, J. Baik, and Y. Choi. Generating test cases for web services using extended finite state machine. In *Proceedings of the 18th IFIP/WG6. International Conference on Testing of Communicating Systems (TEST-COM)*, volume 3964, pages 103–117. Springer, 2006.

[KKKR05] G. Kramler, E. Kapsammer, G. Kappel, and W. Retschitzegger. Towards using UML 2 for modelling web service collaboration protocols. In *Proceedings of the 1st International Conference on Interoperability of Enterprise Software and Applications (INTEROP-ESA).* Springer, 2005.

[KO09] Mandy Krimmel and Joachim Orb. *SAP NetWeaver Process Integration.* Galileo Press, 2nd edition edition, 2009.

[Kön03] H. König. *Protocol Engineering: Prinzip, Beschreibung und Entwicklung von Kommunikationsprodokollen.* Vieweg + Teubner Verlag, 2003.

[Kor90] Bogdan Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[KP06] R. Kazhamiakin and M. Pistore. Choreography conformance analysis: Asynchronous communications and information alignment. In *Proceeding of 3rd International Workshop on Web Services and Formal Methods*, volume 4184, page 227. Springer, 2006.

[KP09] Stefan Kätker and Sabine Patig. Model-driven development of service-oriented business application systems. In *Business Services: Konzepte, Technologien, Anwendungen*, volume Band 1, pages 171–180. sterreichische Computer Gesellschaft, 2009.

[KPV03] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume

2619 of *Lecture Notes in Computer Science*, pages 553–568, Warsaw, 2003. Springer.

[Kru00] Phillipe Kruchten. *The rational unified process: an introduction.* Addison-Wesley Longman Publishing, 2000.

[KRW09] Vitaly Kozyura, Andreas Roth, and Wei Wei. Local enforceability and inconsumable messages in choreography models. In *Proceedings of 4th South-East European Workshop on Formal Methods (SEEFM).* IEEE Computer Society, 2009.

[KT00] S. Kelly and J.P. Tolvanen. Visual domain-specific modeling: Benefits and experiences of using metaCASE tools. In *International workshop on Model Engineering*, 2000.

[KWR10] Vitaly Kozyura, Wei Wei, and Andreas Roth. Deliverable D4.2: Report on enhanced deployment in business information sector. Technical report, Deploy Project, 2010. to be published at `http://www.deploy-project.eu/`.

[Lai02] R. Lai. A survey of communication protocol testing. *Journal of Systems and Software*, 62(1):21–46, 2002.

[LB08] Michael Leuschel and Michael J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.

[Ley01] Frank Leymann. Web Services Flow Language (WSFL 1.0). `http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf`, 2001.

[LI07] R. Lefticaru and F. Ipate. Automatic State-Based Test Generation Using Genetic Algorithms. In *Proceedings of the 9th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 188–195. IEEE Computer Society, 2007.

[Lüb07] Daniel Lübke. Unit testing BPEL compositions. In *Test and Analysis of Web Services*, pages 149–171. Springer, 2007.

[LY96] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines- a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[McM04] Phil McMinn. Search-based software test data generation: a survey. *Software Testing, Verification & Reliability*, 14(2):105–156, 2004.

[Mic96] Microsoft. DCOM Technical Overview. `http://msdn.microsoft.com/en-us/library/ms809340.aspx`, 1996.

[MM04] Nikola Milanovic and Miroslaw Malek. Current solutions for web service composition. *IEEE Internet Computing*, 8(6):51–59, 2004.

[MRR05] A. Milanova, A. Rountev, and B.G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(1):1–41, 2005.

[MS04] Glenford J. Myers and Corey Sandler. *The Art of Software Testing.* John Wiley & Sons, 2004.

[Mur09] Thomas E. Murphy. Using continuous build to drive quality. Gartner Research Report No. G00166848, 2009.

[MXS09] Chris Ma, Qiwen Xu, and J.W. Sanders. A Survey of Business Process Execution Language (BPEL). Technical Report 425, The United Nations University, International Institute for Software Technology (UNU-IIST), 2009.

[OAS07a] OASIS. Web services business process execution language. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf`, 2007. Version 2.0.

[OAS07b] OASIS. Web services reliable messaging. `http://docs.oasis-open.org/ws-rx/wsrm/v1.2/wsrm.pdf`, 2007. Version 1.2.

[OJP99] A. Jefferson Offutt, Zhenyi Jin, and Jie Pan. The dynamic domain reduction procedure for test data generation. *SoftwarePractice & Experience*, 29(2):167–193, 1999.

[O'L00] Daniel E. O'Leary. *Enterprise Resource Planning systems. Systems, life cycle, electronic commerce and risk.* Cambridge University Press, 2000.

[OLAA03] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann. Generating test data from state-based specifications. *Software Testing, Verification and Reliability*, 13(1):25–53, March 2003.

[OMG04] OMG. Common Object Request Broker Architecture: Core Specification. `http://www.omg.org/technology/documents/formal/corba_2.htm`, 2004. Version 3.0.

[OMG06] OMG. Meta object facility (mof) core specification version 2.0. `http://www.omg.org/cgi-bin/doc?formal/2006-01-01`, 2006.

[OMG08a] OMG. Business Process Modeling Notation (BPMN) 1.1, Final Adopted Specification. `http://www.bpmn.org/`, 2008.

[OMG08b] OMG. Business Process Modeling Notation (BPMN) 2.0. `http://www.omg.org/cgi-bin/doc?bmi/08-11-01`, 2008. Submitted Draft Proposal V0.9.

[Org94] International Standards Organization. Iso 9646-1 information technology - open systems interconnection - conformance testing methodology and framework - part 1: General concepts, 1994.

[Ost96] L. Osterweil. Strategic directions in software quality. *ACM Computing Surveys (CSUR)*, 28(4):738–750, 1996.

[OVvdA$^+$07] C. Ouyang, E. Verbeek, W.M.P. van der Aalst, S. Breutel, M. Dumas, and A.H.M. ter Hofstede. Formal semantics and analysis of control flow in WS-BPEL. *Science of Computer Programming*, 67(2-3):162–198, 2007.

[OX04] Jeff Offutt and Wuzhi Xu. Generating test cases for web services using data perturbation. *SIGSOFT Softw. Eng. Notes*, 29(5):1–10, 2004.

[Pap07] Michael P. Papazoglou. *Web Services: Principles and Technology.* Prentice Hall, 2007.

[Par81] D. Park. Concurrency and Automata on Infinite Sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, pages 167–183. Springer-Verlag London, UK, 1981.

[Pel03] Chris Peltz. Web services orchestration and choreography. *Computer*, pages 46–52, 2003.

[PHP99] R.P. Pargas, M.J. Harrold, and R.R. Peck. Test-data generation using genetic algorithms. *Software Testing Verification and Reliability*, 9(4):263–282, 1999.

[PLP04] Alexander Pretschner, Heiko Lötzbeyer, and Jan Philipps. Model based testing in incremental system development. *Journal of Systems and Software*, 70(3):315–329, 2004.

[PMBF05] Patrizio Pelliccione, Henry Muccini, Antonio Bucchiarone, and Fabrizio Facchini. TeStor: Deriving test sequences from model-based specifications. In *Component-Based Software Engineering (CBSE'05)*, volume 3489 of *Lecture Notes in Computer Science*, pages 267–282. Springer, 2005.

[PTDL07] M.P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing: State of the art and research challenges. *Computer*, pages 38–45, 2007.

[PVDBJvV04] I. Pinkster, B. Van De Burgt, D. Janssen, and E. van Veenendaal. *Successful test management: an integral approach.* Springer Verlag, 2004.

[RSM09] Rodrigo Ramos, Augusto Sampaio, and Alexandre Mota. Conformance notions for the coordination of interaction components. *Science of Computer Programming*, 2009. to appear.

[Saf07] L. Safa. The Making Of User-Interface Designer: A Proprietary DSM Tool. In *Proceedings of the 7th OOPSLA Workshop on domain-specific modeling*, pages 21–22, 2007.

[SB01] K. Schwaber and M. Beedle. *Agile software development with Scrum.* Prentice Hall, 2001.

[SBLR07] Manoranjan Satpathy, Michael J. Butler, Michael Leuschel, and S. Ramesh. Automatic testing from formal specifications. In *Proceedings of the 1st International Conference on Tests and Proofs(TAP)*, volume 4454 of *Lecture Notes in Computer Science*, pages 95–113. Springer, 2007.

[SC08a] S-Cube. Po-jra-1.1.1: State of the art report on software engineering design knowledge and survey of hci and contextual knowledge., 2008. project deliverable.

[SC08b] S-Cube. Po-jra-1.3.1: Survey of quality related aspects relevant for sbas, 2008. project deliverable.

[Sea99] C.B. Seaman. Qualitative Methods in Empirical Studies of Software Engineering. *IEEE Transactions on Software Engineering*, 25(4):557, 1999.

[SG08] Carey Schwaber and Mike Gualtieri. The forrester wave: Functional testing solutions 2008. Technical report, Forrester Research, 2008.

[SGS04] David Skogan, Roy Grnmo, and Ida Solheim. Web Service Composition in UML. In *Proceedings of the 8th IEEE Intl Enterprise Distributed Object Computing Conference (EDOC)*, 2004.

[SL96] Yinan N. Shen and Fabrizio Lombardi. Graph algorithms for conformance testing using the rural chinesepostman tour. *SIAM J. Discret. Math.*, 9(4):511–529, 1996.

[Sol06] Torry Harris Business Solutions. Web Services in SOA - Synchronous or Asynchronous? `http://www.thbs.com/pdfs/sync_or_async.pdf`, 2006. Whitepaper.

[SP06] A. Sinha and A. Paradkar. Model-based functional conformance testing of web services operating on persistent data. In *Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications*, pages 17–22. ACM, 2006.

[Ste01] A.P. Steria. France. Atelier B, User and Reference Manuals. `http://www.atelierb.societe.com/indexuk.html`, 2001.

[TCP$^+$05] Wei-Tek Tsai, Yinong Chen, Raymond A. Paul, Hai Huang, Xinyu Zhou, and Xiao Wei. Adaptive testing, oracle generation, and test case ranking for web services. In *29th Int. Computer Software and Applications Conference (COMPSAC'05)*, pages 101–106. IEEE Computer Society, 2005.

[Tha01] S. Thatte. XLANG: Web services for business process design, 2001.

[TPT09] 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.

[Tre04] Jan Tretmans. Model based testing: Property checking for real. `http://www-sop.inria.fr/everest/events/cassis04`, 2004. Keynote at the Int. Workshop for Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04).

[Uhl08] A. Uhl. Model-driven development in the enterprise. *IEEE SOFTWARE*, 25:46–49, 2008.

[UL07] Mark Utting and Bruno Legeard. *Practical model-based testing, a tools approach.* Morgan Kaufmann, 2007.

[UPL06] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing. Technical Report 04/2006, Department of Computer Science, The University of Waikato (New Zeeland), 2006.

[Var02] Daniel Varro. A formal semantics of UML Statecharts by model transition systems. In *Proceedings of the 1st International Conference on Graph Transformation (ICGT)*, volume 2505/2002 of *LNCS*, pages 378–392. Springer, 2002.

[VdADtH03] W.M.P. Van der Aalst, M. Dumas, and A.H.M. ter Hofstede. Web service composition languages: Old wine in new bottles. In *Proceeding of the 29th EUROMICRO Conference: New Waves in System Architecture*, pages 298–305, 2003.

[VdATHKB03] WMP Van der Aalst, AHM Ter Hofstede, B. Kiepuszewski, and AP Barros. Workflow patterns. *Distributed and parallel databases*, 14(1):5–51, 2003.

[vG90] R.J. van Glabbeek. The linear time-branching time spectrum. In *in Proceedings of Theories of Concurrency: Unification and Extension (CONCUR'90)*. Springer, 1990.

[vGG90] Rob J. van Glabbeek and Ursula Goltz. Equivalences and refinement. In *Proceedings of Semantics of Systems of Concurrent Processes*, volume 469 of *Lecture Notes in Computer Science*, pages 309–333. Springer, 1990.

[Vli08] Hans van Vliet. *Software Engineering: Principles and Practice*. Wiley Publishing, 2008.

[W3C01] W3C. Web service definition language (wsdl). `http://www.w3.org/TR/wsdl`, 2001. Version 1.1.

[W3C04a] W3C. Web service glossary. `http://www.w3.org/TR/ws-gloss/`, 2004. Version 20040211.

[W3C04b] W3C. Web Services Choreography Description Language (WS-CDL). `http://www.w3.org/TR/ws-cdl-10/`, October 2004. Version 1.0.

[W3C07] W3C. Simple Object Access Protocol (SOAP). `http://www.w3.org/TR/soap`, 2007. Version 1.2.

[W3C08] W3C. Extensible Markup Language (XML) . `http://www.w3.org/XML/Core`, 2008. Version 1.0 (Fifth Edition).

[WBLH07] Y. Wang, X. Bai, J. Li, and R. Huang. Ontology-based test case generation for testing web services. In *Eighth International Symposium on Autonomous Decentralized Systems*, pages 43–50, 2007.

[Wey98] Elaine J. Weyuker. Testing component-based software: A cautionary tale. *IEEE Software*, 15(5):54–59, 1998.

[WfM08] WfMC. XML Process Definition Language (XPDL). `http://www.wfmc.org/xpdl.html`, 2008. Version 2.1.

[WJ91] Elaine J. Weyuker and Bingchiang Jeng. Analyzing partition testing strategies. *IEEE Transactions of Software Engineering*, 17(7):703–711, 1991.

[WL99] D.M. Weiss and C.T.R. Lai. *Software product-line engineering: a family-based software development process.* Addison-Wesley, 1999.

[WM06] Dan Woods and Thomas Mattern. *Enterprise SOA - Designing IT for Business Innovation.* O'Reilly, 2006.

[ZBDtH06] Johannes Maria Zaha, Alistair P. Barros, Marlon Dumas, and Arthur H. M. ter Hofstede. Let's dance: A language for service behavior modeling. In *OTM Confederated International Conferences (CoopIS'06)*, volume 4275 of *Lecture Notes in Computer Science*, pages 145–162. Springer, 2006.

[ZCCK04] Jia Zhang, Carl K. Chang, Jen-Yao Chung, and Seong W. Kim. WS-Net: A Petri-net Based Specification Model for Web Services. In *Proceedings of the IEEE International Conference on Web Services (ICWS)*, 2004.

[ZHM97] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR)*, 29(4):366–427, 1997.

Publication List

[1] Sebastian Wieczorek and Alin Stefanescu. Challenges for ERP test data generation. In *Proc. of 5th Workshop "System Testing and Validation"*, pages 77–86. Fraunhofer IRB Verlag, 2007.

[2] Sebastian Wieczorek, Alin Stefanescu, and Ina Schieferdecker. Test data provision for ERP systems. In *Proc. of Int. Conf. on Software Testing (ICST'08)*, pages 396–403. IEEE Computer Society, 2008.

[3] Sebastian Wieczorek, Alin Stefanescu, and Jrgen Gromann. Enabling model-based testing for SOA integration. In *Proceedings of Model-based Testing in Practice (MOTIP'08)*, pages 77–82. Fraunhofer IRB Verlag, 2008.

[4] Sebastian Wieczorek and Alin Stefanescu. Service integration: A soft spot in the SOA testing stack. In *Proceedings of the 5th Central and Eastern European Software Engineering Conference in Russia (CEE-SECR'09)*. IEEE Computer Society, 2009.

[5] Sebastian Wieczorek, Alin Stefanescu, Mathias Fritzsche, and Joachim Schnitter. Enhancing test driven development with model based testing and performance analysis. In *Proc. of Testing: Academic & Industrial Conference - Practice and research techniques (TAICPART'08)*, pages 82–86. IEEE Computer Society, 2008.

[6] Sebastian Wieczorek, Andreas Roth, Alin Stefanescu, and Anis Charfi. Precise steps for choreography modeling for SOA validation and verification. In *Proceedings of the IEEE 4th International Symposium on Service-Oriented Software Engineering (SOSE'08)*, pages 148–153. IEEE Computer Society, 2008.

[7] Sebastian Wieczorek, Andreas Roth, Alin Stefanescu, Vitaly Kozyura, Anis Charfi, Frank Michael Kraft, and Ina Schieferdecker. Viewpoints for modeling choreographies in service-oriented architectures. In *Proc. of 8th IEEE/IFIP Conference on Software Architecture (WICSA'09)*, pages 11–20. IEEE Computer Society, 2009.

[8] Sebastian Wieczorek, Alin Stefanescu, and Ina Schieferdecker. Model-based integration testing of enterprise services. In *Proc. of Testing: Academic & Industrial Conference - Practice and research techniques (TAICPART'09)*, pages 56–60. IEEE Computer Society, 2009.

[9] Sebastian Wieczorek, Vitaly Kozyura, Andreas Roth, Michael Leuschel, Jens Bendisposto, Daniel Plagge, and Ina Schieferdecker. Applying model checking to generate model-based integration tests from choreography models. In *Proceedings of the 21st IFIP Int. Conference on Testing of Communicating Systems (TESTCOM'09)*, LNCS. Springer, 2009.

[10] Alin Stefanescu, Sebastian Wieczorek, and Andrei Kirshin. MBT4Chor: A model-based testing approach for service choreographies. In *European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'09)*, volume 5562 of *LNCS*, pages 313–324. Springer, 2009.

[11] Thomas Bauer, Hajo Eichler, Axel Rennoch, and Sebastian Wieczorek, editors. *Proc. of 2nd Workshop on Model-based Testing in Practice (MOTIP'09)*, volume WP09-08 of *Workshop Proceedings Series*. CTIT, 2009.