

Chapter 2

A Generic Framework for Testing the Web Services Transactions

Rubén Casado, Muhammad Younas and Javier Tuya

Abstract This chapter focuses on web services transactions which support creating robust web services applications by guaranteeing that their execution is correct and the data sources are consistent. More specifically, it investigates into the *testing* of such transactions which has not received proper attention from the current research. It presents a generic framework for testing different models and standards of web services transactions. The framework is implemented as a prototype system using the case study of Jboss Transactions and is applied to test the predominant web services models and standards such as Web Services Business Activity (WS-BA). The results show that the framework automatically generates test cases and detects possible faults or failures during the processing of web services transactions running under different model and standards.

2.1 Introduction

Web services provide a new computing paradigm in which functional and non-functional requirements of specialised services are published over the Internet such that they can be dynamically discovered and composed in order to create composite services that provide integrated and enhanced functionality. Web services transactions (or WS transactions) are used to ensure reliable execution of services and to maintain the consistency of data. WS Transactions are defined as sequences of web services operations or processes that are executed under certain criteria in order to

R. Casado (✉) · J. Tuya
Department of Computing, University of Oviedo, Asturias, Spain
e-mail: rcasado@lsi.uniovi.es

J. Tuya
e-mail: tuya@uniovi.es

M. Younas
Department of Computing and Communication Technologies, Oxford Brookes University,
Oxford, UK
e-mail: m.younas@brookes.ac.uk

achieve mutually agreed outcome regardless of system failures or concurrent access to data sources i.e., either all the web services operations succeed completely or fail without leaving any incorrect or inconsistent outcomes. The classical and most widely used criteria are the ACID (Atomicity, Consistency, Isolation, Durability) which require that a transaction be treated as a single atomic unit of work in order to maintain consistency and persistency of data. Consider, for example, an online service provider (e.g., Amazon) that develops web services based solutions to automate the order and delivery of online books as part of a WS transaction. Such transaction can only be considered as successful once the books (purchased) are delivered to a customer and the payment has received.

Numerous models and protocols have been developed for WS Transactions, including, the OASIS Business Transaction Protocol (BTP) [24], Web Services Business Activity (WS-BA) [29], Web Services Transaction Management (WS-TXM) [25] and other models and frameworks [1, 20]. These aim to improve the quality of WS transactions in terms of response time efficiency, failure recovery, flexibility and support for long running and complex business applications. For example, [1] present an optimistic concurrency control protocol in order to optimise the throughput and response time of WS transactions. The authors in [20] propose an algorithm for selecting QoS-aware transactional web services that meet user's requirements.

This chapter focuses on another quality dimension which is the testing of WS transactions. Though there exists research work on testing non-transactional web services [4, 5], the area of WS transactions testing has not been properly researched yet.

Generally, the software testing aims to systematically explore the behaviour of a system or a component in order to detect unexpected behaviours. In other words, testing identifies whether the intended and actual behaviours of a system differ, or (at gaining confidence) that they do not. In our case, the focus of testing is to detect possible faults or failures in WS transactions running under different models or standards (e.g., BTP, WS-BA). The objective is to identify the observable differences between the behaviours of implementation and what is expected on the basis of specification of WS transaction models and standards. Based on our previous work [8, 9], this chapter presents a generic framework for testing WS transactions. The framework is comprised of the following phases:

- To design a generic model that abstractly represents the commonly used WS transaction models and standards (e.g., BTP, WS-BA).
- To automatically generate test cases and map them to different WS transactions models and standard.
- To perform testing and evaluation using the standard case study of Night Out, which is provided by Jboss [19] in their implementation of the WS-BA standard.
- To automatically compare the expected and actual outcomes in order to identify possible faults or failures in WS transactions.

The chapter is organized as follows. Section 2.2 gives an analysis of WS transaction models and standards. Section 2.3 presents the proposed framework. It also presents the generic transaction model and illustrates the process of representing some of the WS Transactions standards using the proposed transaction model. Section 2.4

presents the evaluation and results. Section 2.5 gives a critical analysis of the proposed framework. Conclusions are presented in Sect. 2.6.

2.2 WS Transactions

WS transactions are defined as sequences of web services operations or processes that are executed under certain criteria in order to achieve mutually agreed outcome regardless of system failures or concurrent access to data sources. But WS transactions have distinct characteristics than the classical database transactions. They are based on various models ranging from classical ACID criteria to advanced or extended transaction models. Two Phase Commit (2PC) protocol and its variants [12] have commonly been used for maintaining ACID properties. ACID properties are vital for WS transactions that need strict isolation and data consistency. However, they are not suitable for long running WS transactions as they result in resource locking/blocking problems. Advanced transaction models have been developed to address 2PC and ACID related issues. These includes, nested transaction model [23], SAGA model [15], open-nested [33], Split-join [31], Contracts [32], Flex [35], and WebTram [34]. The underlying strategy of these models is to relax the strict ACID criteria and to allow for compensation of partially completed transactions in order to maintain application correctness and data consistency.

The work in [11] proposes a theoretical approach in order to specify, analyze and synthesize advanced transaction models. Transactional patterns that combine workflow process adequacy and the transactional processing reliability are identified in [2]. In [16], the authors present a high level UML-based language to design transaction process with diverse transactional semantics. An XML representation is proposed in [18]. In our previous work [7], a risk-based approach is used to define general test scenarios for compensatable transactions. Further, in [6], we present test criteria for transactional web services composition. The approach is based on the dependencies which are defined between participants of a WS transaction. In [21], authors have developed a model of communicating hierarchical timed automata in order to describe long-running transactions. This approach verifies the properties of transactions using model checking. The work presented in [13] translates programs with compensations to tree automata in order to verify compensating transactions. The authors in [22] proposes a formal model to verify the requirement of relaxed atomicity with temporal constraints whilst [14] uses event calculus to validate the transactional behaviour of WS compositions.

In addition to the above, several standards have been developed for WS transactions. For instance, the OASIS Business Transaction Protocol (BTP) [24] coordinates loosely web services. BTP was designed and developed by several major vendors including BEA, Hewlett-Packard, Sun Microsystems, and Oracle. BTP adapts 2PC for short lived transactions and nested transaction model for long-lived transactions.

Web Services Composite Application Framework (WS-CAF) [25] is a set of WS specifications in order to support composite web services applications. Basically, WS-CAF uses WS-Transaction Management (WT-TXM) to manage transactions

Table 2.1 Test execution results

Standards	Coordination	Transaction model		Relationship
		Short	Long	
BTP	✓	ACID/2PC	Nested	✗
TXACID	✓	ACID/2PC	✗	WS-TXM
TXLRA	✓	✗	SAGA	WS-TXM
TXBP	✓	✗	Open-nested	WS-TXM
WS-AT	✓	ACID/2PC	✗	WS-COOR
WS-BA	✓	✗	SAGA	WS-COOR

in composite services. WT-TXM is built around three models: ACID Transaction (TXACID), Long Running Transaction (TXLRA) and Business Transaction Process (TXBP). These models are defined in order to meet the different requirements of web services. For example, if a web service is required to abide by strict isolation and consistency policy then it adapts the TXACID model.

Web Services Atomic Transactions (WS-AT) [28] and Web Services Business Activity (WS-BA) [29] are built on top of Web Services Coordination (WS-COOR) [27]. WS-AT and WS-BA thus follow the coordination mechanism of WS-COOR. WS-AT follows 2PC protocol while WS-BA uses the SAGA model.

The above standards and their underlying transaction models and protocols are summarized in Table 2.1. ‘Coordination’ represents whether a particular standard provides coordination facilities. ‘Transaction Model’ shows the underlying transaction models and protocols on which the WS transaction standard is based on. ‘Short’ and ‘Long’ respectively represent short-lived and long-lived WS transactions. ‘Relationship’ represents the relationship between the WS transaction standards.

From Table 2.1, we make some useful observations that motivate the need for a generic model for testing the WS transactions. Our first observation is that all the standards separate the coordination and the management of transactions and also distinguish between short-lived and long-lived transactions. Second, these standards have proprietary definitions of their underlying transaction models despite the fact that some of them are based on similar concepts. Third, the support for long-lived transactions is based on different advanced transaction models. For instance, TXLRA adapts SAGA while TXBP adapts open-nested transaction model. This reveals that WS transactions do not have a homogeneous transaction models or protocols. Instead they are characterized by a diversity of transaction models and protocols.

Given the diversity of WS transactions standards it is essential to develop a generic model that has the capability to represent and test WS transactions running under different standards. In the next section we define the proposed framework.

2.3 The Generic Framework

This section presents the proposed framework for testing the WS transactions. It first describes the transaction model and then illustrates the process of modelling the current WS transaction standards.

2.3.1 The Transaction Model

This section presents the first phase of the proposed framework i.e., to design a generic model that abstractly represents the commonly used WS transaction models and standards. It provides the basic definitions and relationships of WS transactions and also explains the different roles played by the participants (component systems) in the execution of WS transactions.

WS Transaction: A *WS Transaction*, wT , is defined as a set $S = s_1, \dots, s_n$ of sub-transactions (or activities) which are executed in order to consistently and (semi) atomically acquire web services. Each wT is associated with one *Coordinator*, k , while each sub-transaction, s_i , is executed by an *Executor*, e_i . *Transaction context* is defined as a set of functional information and transaction configuration shared by the sub-transactions. Each s_i can be represented as a single level sub-transaction or as nested sub-transactions, which is denoted as wT_c . wT , s_i , and wT_c are related in a *parent:child* relationship. The outcome of wT is called atomic if all its sub-transactions complete their execution in an agreed manner. Alternatively, the outcome is called mixed if subtransactions can have different final states or outcomes, i.e., some completed and others not.

In the proposed model, subtransactions have different types [3, 20]. A subtransaction, s_i , is *lockable* if the resources (or data) that it uses can be locked until the completion of the parent transaction. A sub-transaction is *compensatable* if its effect can be semantically undone through a compensating transaction. If a sub-transaction is successfully completed and its effects cannot be semantically undone, then it is called *pivot*. A sub-transaction is *retrieable* if it guarantees a successful termination after a finite number of invocations. A sub-transaction is *replaceable* if there is an alternative sub-transaction that can perform a similar task. Note that the different types of sub-transactions are defined as these are commonly used in WS transaction models and standards.

The execution of a wT involves different participants, each of which plays a certain role. We identify four different roles for the participants involved in processing the wT and its sub-transactions:

- *Executor*: represents a participant which is responsible for executing and terminating a sub-transaction.
- *Coordinator*: coordinates the overall execution of wT . For instance, it collects the results (votes) from participants in order to consistently process wT .
- *Initiator*: represents a participant which starts wT . That is, it submits wT to the coordinator and requests a transaction context.

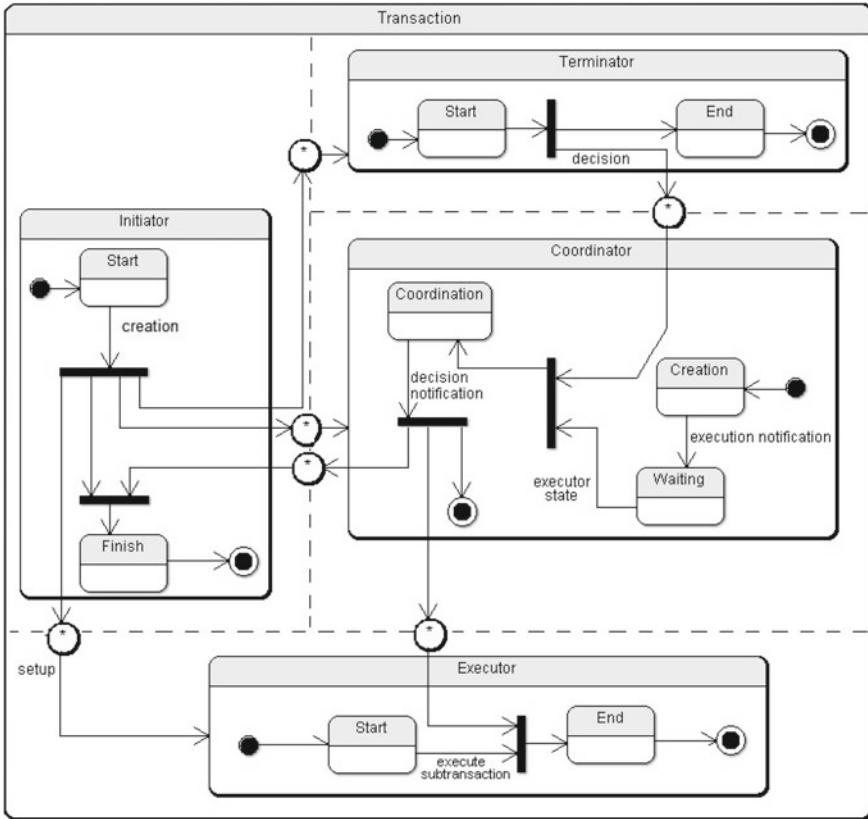


Fig. 2.1 Participant and roles in the proposed transaction model

- *Terminator*: represents a participant which decides when and how wT has to be terminated. It also participates in the coordination tasks. In some situations, it can play the role of a sub-coordinator.

The above roles are diagrammatically represented in Fig. 2.1 using UML state chart notation. The purpose of defining the above roles is to automatically and uniformly represent the different roles of participants in different WS transactions standards. As shown in Fig. 2.1, each participant plays a certain role and makes transition from one state to another during the processing of wT .

2.3.2 Representation of WS Transaction Models and Standards

This section describes the process of modelling WS transaction models and standards using the proposed framework. As proof of concept we model the BTP and WS-BA

standards as these are the commonly accepted standards in WS transactions. The modelling process is composed of the following steps:

2.3.2.1 Role Identification and Modelling

This step identifies the roles of participants in a target WS transaction standard and models it using the roles defined in the proposed framework.

The BTP implements the nested transaction model [23] and defines two main roles; *Superior* and *Inferior*. In other words, it defines *Superior:Inferior* relationship between a parent transaction, wT , and its sub-transactions, s_i . Figure 2.2a shows the BTP representation of wT and its sub-transactions using the *Superior:Inferior* relationship, and Fig. 2.2b represents the same wT using the proposed framework. In BTP the superior makes the decision and the inferior abides such decision in order to complete the transaction. The superior of BTP is modelled as Initiator in the proposed framework. Also the superior can be modelled as Coordinator and Terminator as it decides on the outcome of the subtransactions. Inferior of BTP executes a subtransaction and is therefore modelled as Executor in the proposed framework.

The WS-BA defines two outcomes of wT : (i) *MixedOutcome* allows that sub-transactions may have distinct outcomes or final states, (ii) *AtomicOutcome* requires all the subtransactions to complete their execution in an agreed manner. The main roles are played by the: *Executor* and *Coordinator*. Figure 2.3 depicts the modelling of WS-BA using the proposed framework. Figure 2.3a shows the *AtomicOutcome*, whilst Fig. 2.3b shows the *MixedOutcome* scenario. In both scenarios the role of *Initiator* is taken by the first participant who interacts with a *Coordinator*. In *AtomicOutcome* the role of *Terminator* is taken by the *Coordinator*. This is due to the fact that *Coordinator* can be the participant that knows all Executors's output. It also knows the final outcome: close or terminate wT if all executors have successfully executed their sub-transactions, or compensated otherwise. In *MixedOutcome*,

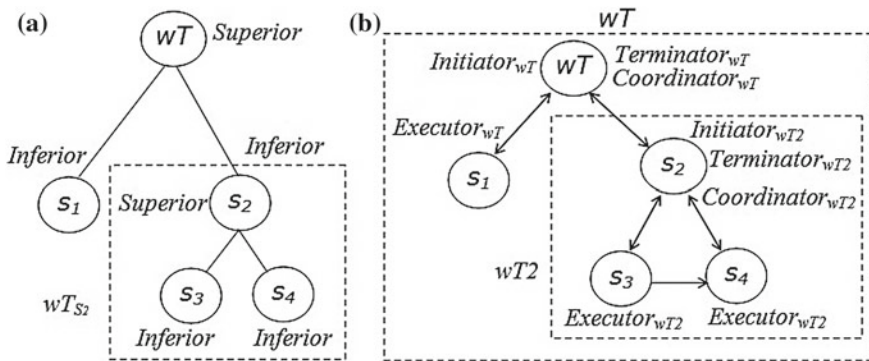


Fig. 2.2 Representation of BTP roles and relationships

the *Initiator* is the *Terminator* since each *Executor* may have its specific or distinct decision so the outcome depends on the business logic.

2.3.2.2 State Transitioning and Messages

This section describes the mapping of the state transitions and messages between a target WS transaction standard and the proposed framework.

Figures 2.4 and 2.5 give more details on the state transitions and message communication between *Executor* and *Coordinator* during the processing of wT . Note that here we only model these two participants as they play a major role in executing wT . The *Inferior* and *Superior* (in BTP) are respectively represented by *Executor* and *Coordinator*. Similarly *Executor* and *Coordinator* are used to represent WS-BA participants involved in wT .

BTP mapping: When a wT is started at the initiative of an *Initiator* a request is sent to the *Coordinator* for creation of a context for the new transaction. The *Coordinator* replies the *Initiator* and other *Executors* with the context information and then moves from *INITIAL* state to *ACTIVE* state. Each *Executor* receives a context, enrolls with the *Coordinator* and then moves from *READY* to *ACTIVE* state. The *Executor* moves to *COMPLETED* state after processing its sub-transaction. *Coordinator* moves to *PREPARE* state awaiting decisions from *Executors*. The *Executor* sends its outcome to the *Coordinator* and moves to *DECISION* state. The *Coordinator* collects the outcomes from all *Executors* and takes the final decision by moving from *PREPARE* state to *DECISION* state. The final decision is sent to each *Executor* and the *Coordinator* then moves to *CONFIRM* state. Each *Executor* sends acknowledgement and changes its state to *END* state through the transition (either completed rollback or completed successfully). Once the *Coordinator* has received all confirmations, it moves to *END* state. Note that an *Executor* can leave the wT before confirming the completion of sub-transaction. So it can move from *ACTIVE* state to *CANCEL* state.

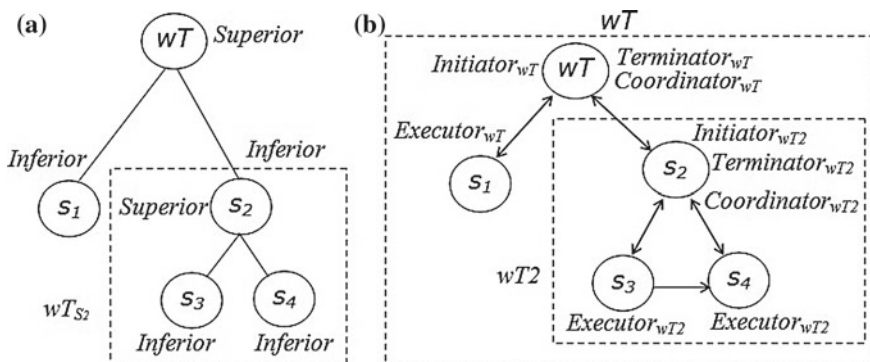


Fig. 2.3 WS-BA relationships modeling

Although BTP uses the 2PC protocol, Executors are not required to lock data on becoming prepared (i.e., in prepared state). This can produce a contradictory decision since the Coordinator could take a decision for all the Executors but some Executors may take their own decisions. When the Coordinator detects any contradiction it notifies the concerned Executor and moves to the END state. If the Coordinator wants to cancel, the Executor uses *completed pivot*. In some cases, it uses *completed rollback*. Further, BTP allows replaceable subtransactions. Thus if an Executor is not able to start or carry on with its sub-transaction, it moves to FAILED state. A new Executor is selected and the previous one moves to END state.

WS-BA mapping: The Initiator initiates wT and requests a context from Coordinator. The Coordinator responds with a context. After wT initiation, Executors join the current wT and move from READY to ACTIVE state, wherein they execute their sub-transactions. After processing sub-transaction, each Executor moves from ACTIVE to COMPLETED state. Coordinator moves from ACTIVE to PREPARE state after receiving decision from all the Executors. In WS-BA, when the transaction is of *MixedOutcome*, the decision for each sub-transaction is taken independently by each Executor. In this case, the Coordinator moves from PREPARE to DECISION state whenever it receives an Executor's notification. The Coordinator decides about its outcome and moves from DECISION to CONFIRM state. In the case of *AtomicOutcome* type, the Coordinator moves from PREPARE to DECISION state after receiving decisions from each Executor. The Coordinator then sends the global decision to all Executors and moves from DECISION to CONFIRM state. Finally it awaits the acknowledgements from Executors. Once these are received, the Coordinator then moves to END state. When an Executor is not able to start executing its sub-transaction it moves from READY to ABORTED state. If the sub-transaction was cancelled while it was still under execution, the Executor moves from ACTIVE to CANCELLED state. In case of failure it moves from ACTIVE to FAILED state.

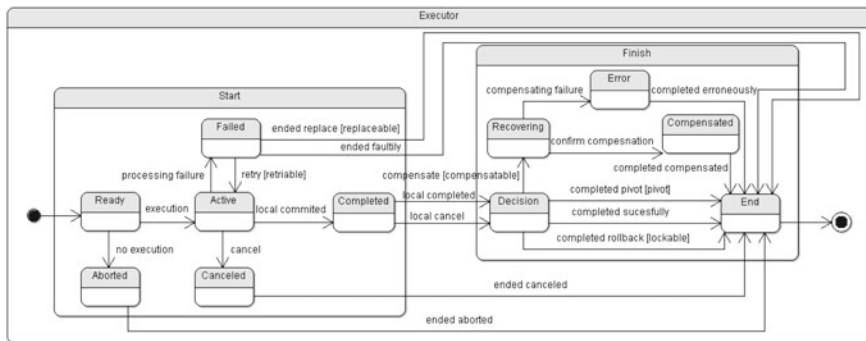


Fig. 2.4 Executor: State transitions and message communication

2.4 Implementation and Evaluation of the Proposed Framework

The process of testing aims at showing whether the intended and actual behaviours of a system differ, or at gaining confidence that they do not. The main goal of testing in our context is failure detection, i.e., the observable differences between the behaviours of implementation and what is expected on the basis of the specification of WS Transaction standards. We exploit a model-based testing approach that encodes the intended behaviour of a system and the behaviour of its environment. Model-based testing is capable of generating suitable test cases and it has also been successfully used in others WS domains [10].

In order to validate and evaluate our framework we have designed a test process which comprises test design, test implementation, test execution and outcome evaluation. In the following, we first explain the testing process. We then illustrate the implementation of the proposed framework. Finally, we discuss the evaluation of the framework.

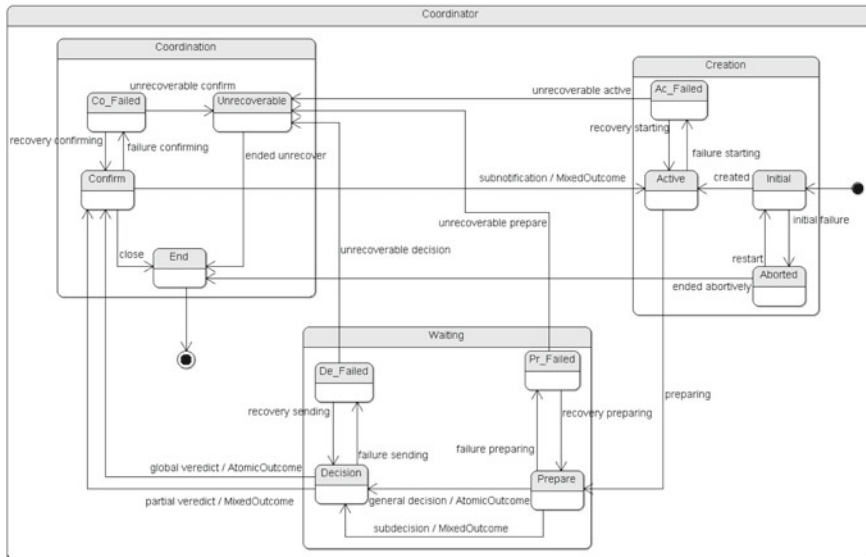


Fig. 2.5 Coordinator: State transitions and message communication

2.4.1 Testing Process

The testing process includes selecting a test criterion, test design, test implementation, test execution and outcome evaluation. This section presents how the proposed framework implements those phases using the generic transaction model.

The first step to design the tests is to select a test criterion. Since the model is based on states and transitions, we use the well known criterion of transition coverage [30]. By applying a test criterion over the generic transaction model, we obtain a set of abstract test cases. Each abstract test case is mapped to a concrete test case which is composed of the test scenario and the expected system outcome. The basic concepts used in the test process are defined as follows.

- *Test criterion*: This defines a rule that imposes constraints (or requirements) on a set of test cases.
- *Transition coverage criterion*: The set of test cases must include tests that cause every transition between states in a state-based model (e.g., as in Figs. 2.4 and 2.5)
- *Abstract Test case*: This represents a sequence of states and transitions of a participant using the generic transaction model. The notation $S_i \rightarrow S'_i$ is used to denote that the participant p_i changes its current state S to S' executing the transition labelled, t . If the participant is the Coordinator, it is denoted by k . We use $S_i^a \rightarrow S_i^b \dots S_i^c \rightarrow S_i^d$ to denote a sequence of state transitions.
- *Test scenario*: This represents a sequence of actions in a human-understandable way to provide guidance to the tester to execute a test case.
- *System outcome*: The internal state of the process defined by a sequence of exchanged messages between participants using a specific WS transaction standard. The notation $i[m_1]j$ is used to denote that the participant p_i sends message m_1 to participant p_j . We use $i[m_1]j - l[m_2]o - \dots - v[m_n]z$ to denote a sequence of messages.

The test phases included in the proposed framework are depicted in Fig. 2.6 and are described as follow:

Test design: This phase defines the test requirements for an item and derives the logical (abstract) test cases. At this stage the test cases do not have concrete values for input and the expected results. The abstract test cases are automatically generated by applying transition coverage criterion over the abstract model. It is obtained from a set of different paths where each path defines an abstract test case. Thus the tests achieved using this criterion are a set of paths that cover all states and transitions of a model.

Test implementation: The sequence of states and transitions specified by the abstract test cases generated in the test design phase are mapped to a specific WS transaction standard, for example, BTP or WS-BA (see Sect. 2.3). As discussed above the proposed generic model has the ability to capture the behaviours of WS transaction standards as well as mapping the abstract cases to different WS transaction standards. These features provide the capability of automatically obtaining the test scenario and the expected system output.

Test execution and outcome evaluation: Once the test cases are implemented, they are executed over the system under test (e.g., BTP or WS-BA) and the actual outcome is obtained. Finally, for each test case, the expected outcome is compared to the actual outcome to find differences in behaviour and to detect failures. Two outcomes are considered: (i) *user outcome*: this refers to what the user perceives; for instance, to reserve theatre tickets and to see whether the number of booked tickets is correct. (ii) *system outcome*: this refers to the non-visible process that the system

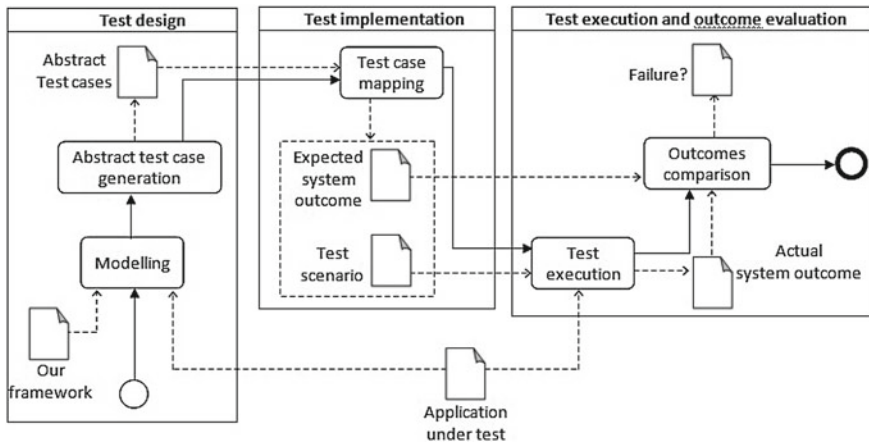


Fig. 2.6 Test process of the proposed framework

has carried out to achieve the requirements e.g., the correct exchange of messages between the participants according to the given transaction standard.

Both outcomes are necessary for detecting the differences in the behaviour of WS transactions. Consider a simple application that runs as a WS transaction in order to book theatre tickets. Assume that there is a fault in creating messages and the format of confirmation messages is incorrect. In a test scenario where the user confirms a reservation, the systems outcome would be to inform the user that the booking was successfully completed because the application has already sent the confirmation message to the theatre service. Since the message was incorrectly created, the theatre service would reject the reservation and, as a result, the tickets cannot be booked. Thus, the tester needs not only the user outcome, but also the internal state of the process to know whether a test case has detected a failure or not. In this work we focus on Executors internal behaviours related to the WS transactions. Thus we only need to evaluate the system outcome.

2.4.2 Prototype System

We have developed a prototype system that implements the main phases of the proposed framework (Fig. 2.6).

- *Modelling*: The prototype system prompts the tester to provide information (e.g. services, roles, transaction standard, etc) and to create the WS transaction.
- *Abstract test case generation*: the abstract test cases for all the participants (Coordinator, Executor, etc) are automatically generated by the prototype system.
- *Test case mapping*: Abstract test cases are mapped to WS transaction standards (e.g., BTP or WS-BA). That is, the prototype system automatically generates the

concrete test cases (for each WS transaction standards) which are composed of the test scenario and the expected system outcome. A test scenario is defined as a sequence of actions in a human-readable way to provide guidance to the tester to execute a test case.

- *Outcomes comparison*: test cases are executed in order to produce the actual systems outcome. The prototype system automatically compares the actual systems outcome with the expected systems outcome in order to detect any fault or failure.

The prototype system is implemented in Java 1.5. It includes three components: *Model*, *Tests* and *Outcome*. The *Model* implements the generic transaction model. It also includes a graphic interface to allow the tester to enter all the necessary information such about the system under test such as roles, URL, WS transaction standard, etc. The *Model* component sends the information to the *Tests* component. The *Tests* component implements two activities: first, it applies the transition coverage criterion in order to generate the abstract test cases for all the participants. It then maps all the abstract test cases into concrete test cases. That is, the *Model* component generates the test scenario (text file) and the expected systems outcome (as an XML file). Finally, the *Outcome* component compares two XML files to identify any possible faults. This component has a graphic interface that allows the tester to add an XML file (the actual systems outcome obtained from the execution of test scenario) and to select the test case for comparison purpose. The result of both outcomes is shown to the tester.

2.4.3 Evaluation

In order to evaluate the proposed framework we utilise the *Night Out* case study which is adopted from the Jbosss implementation of the WS-BA standard [29]. This study concerns booking three independent services for night time leisure: *Restaurant* service allows customers to reserve a table for a specified number of dinner guests. *Theatre* service provides reservation of seats in a theatre and allows customers to book a specified number of tickets for different categories such as seats in circle, stalls, or balcony. *Taxi* service provides the facility to book a taxi. These services are implemented as transactional web services. The client side of the application is implemented as a servlet which allows users to select reservations and then book a night out by invoking each of the services within the scope of a WS transaction. For example, if seats are not available in a restaurant or a theatre, then taxi will not be required. Each service, exposed as Java API for XML Web Services (JAX-WS) [17] endpoint, has a GUI with state information and an event trace log.

In this chapter we described the process of modelling and testing the WS-BA standard using the prototype system. But the prototype system is capable of representing different WS transaction models and standards.

2.4.3.1 Modelling of WS-BA-Based Transactions

The transactional aspects of WS-BA included in the *Night Out* application has been modelled according to the aforementioned procedure. As shown in Fig. 2.7, *Night Out* (client side) takes the role of Initiator since it starts the transaction and asks the other web services to participate in the transaction. *Restaurant*, *Theatre* and *Taxi* services are modelled as Executors as they execute individual sub-transactions. Some sub-transactions (e.g. *Theatre*) are independent of others (e.g. *Restaurant*). That is, if one sub-transaction cannot complete its execution the others are allowed to commit. The *Taxi* activity is dependent on some of the services. For instance, if a table is not available in the restaurant, the customer still needs a taxi to go to the theatre. The role of Coordinator is taken by an external service, *WSCoorII*, provided by the server. It follows the WS-COOR [27] and WS-BA [29] standards to exchange required messages.

2.4.3.2 Abstract Test Case Generation and Mapping

This phase generates various abstract cases for each Executor, i.e., *Restaurant*, *Theatre* and *Taxi*. The abstract test cases are automatically generated and mapped to specific standard in this case, the WS-BA standard. As explained above, these tests cases define the test scenario and the expected system outcome. For example, in the following we explain the process of mapping the abstract test cases to a specific sequence of WS-BA messages. Consider the sequence shown in Fig. 2.8 of state transitioning and messages wherein an Executor moves from Ready to End state (see Fig. 2.4).

Applying the transition coverage criterion over the above, abstract test case is mapped to a specific sequence of WS-BA message (see Fig. 2.9). From this sequence of messages, our prototype system automatically generates the test scenario which is shown in Fig. 2.10.

Based on the above, the prototype system can generate and map various test cases for *Restaurant*, *Theatre* and *Taxi* services. Figure 2.11 contains eight test cases for the *Restaurant*, *Theatre* and *Taxi*. Res_1, Thr_1, and Tax_1 respectively represent test case 1 for *Restaurant*, *Theatre* and *Taxi* services. Res_2, Thr_2, and Tax_2 mean test case 2 and so on. Note that these eight are example test cases. But the prototype system is capable of generating other possible test cases.

2.4.3.3 Test Execution and Outcome Evaluation

The prototype system executes the generated test cases using the *Night Out* services. The results of test execution are summarised in Table 2.2. ‘Pass’ means that a test case is executed but has not detected any failure during the processing of a service (e.g., booking a restaurant, theatre or taxi). ‘Fails’ means that the actual outcome differs from the expected outcome (i.e. a fault has been detected). ‘Blocked’ means

that a test case cannot be executed because the application does not have the interface to perform the required actions.

Pass: Test cases 3, 6, and 7 are executed but the prototype system has detected no failures. That is, Rest_3, 6, 7, Thr_3, 6, 7, and Tax_3, 6, 7 have passed the tests.

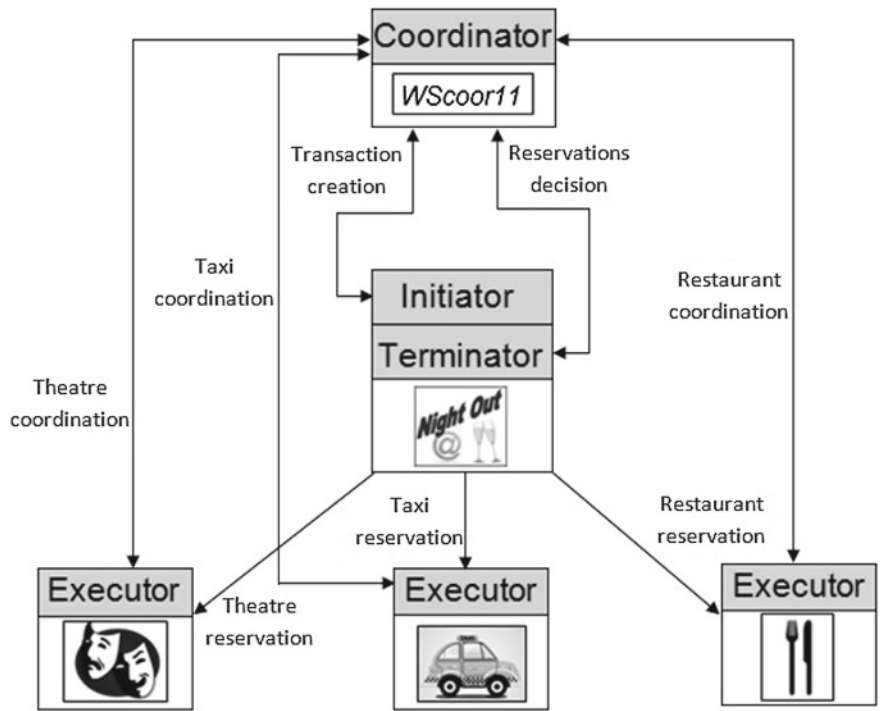


Fig. 2.7 Roles and representation of Night Out Services



Fig. 2.8 Executor abstract sequence

Blocked: Two of the test cases were blocked due to the following reasons. Test case 1 requires cancelling the activity (*Cancel* message) once the Executor has started but has not finished yet. But WS-BA standard does not allow cancelling a concrete booking once the service has started executing its activity. Test case 8 defines a scenario where the Executor is not able to complete its activity (*CanNotComplete* message) but has retried executing its action. However, the WS-BA does not allow the services to retry its activity without starting a new transaction.

Fail: During the execution of test cases 3 and 4 interface-related failures were detected. The application, which allows changing manually the capacity of each

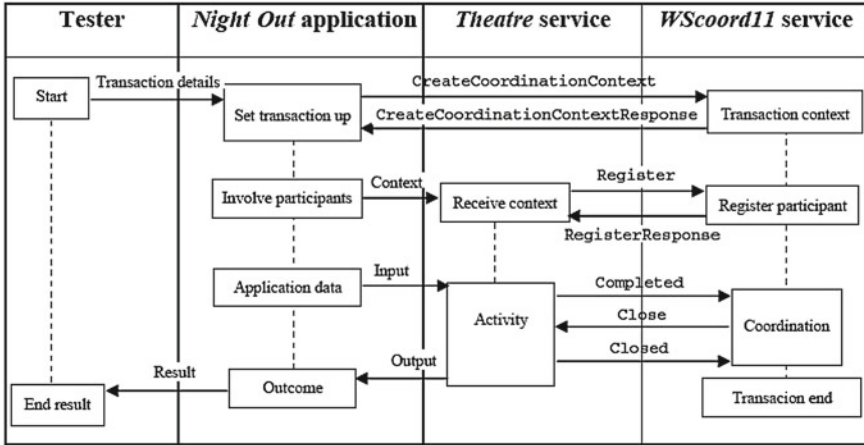


Fig. 2.9 Sequence diagram of a test scenario for theatre service

resource (i.e. number of tables and number of seats in the theatre), either crashes or does not update the capacity when the button is pressed.

Test case 5 detected an important transaction-related failure in the compensation process under WS-BA specification. The goal of this test case is to successfully confirm the booking of theatre tickets when the other service reservations (restaurant and taxi) have been undone through compensating transactions (see Fig. 2.11).

After the execution of the test case, we obtain the expected systems outcome. By comparing the expected systems outcome and the actual systems outcome, a failure is detected by the prototype system. This is shown in the code snippet in Fig. 2.12. The expected systems outcome requires receiving a CLOSE message once the *Theatre* service has successfully completed its activity (see sequence diagram in Fig. 2.9). However, the actual outcome has a COMPENSATED message since *Restaurant* service was not able to commit. As a result, the *Theatre* reservations were automatically undone. The fault which causes such failure is detected by the prototype system as there is a difference (or discrepancy) in the ‘Register’ message the way *Theatre* service is registered in the *Night Out* under the WS-BA specification. That is, it registers the *Theatre* service as an *AtomicOutcome* when a *MixedOutcome* was expected (Fig. 2.13). In other words, if *Taxi* or *Restaurant* services are not able to make their reservations, the *Theatre* service will automatically undo the reservation even if the customer would wish to keep the theatre tickets.

The results obtained from the test comparison are also useful for a debugging process. In the above tests, the faults mean that the transaction was not correctly configured or coded. This can help in identifying the faults in the code. For example, the above fault was found in *BasicClient.java* file, at line number 496 in the code shown in Fig. 2.14. The configuration of the transaction is made using the class *UserBusinessActivityImple*, through the factory pattern using *UserBusinessActivityFactory*

class. By looking at the implementation of that class we found (in Fig. 2.15) that the transaction is defined as an *AtomicOutcome*.

STEP 1: *NightOut* starts the process. It sends a context request (*CreateCoordinationContext* message) to the coordinator *WScorr11*

STEP 2: *WScorr11* sends the transaction context (*CreationCoordinationContextResponse* message) to *NightOut*

STEP 3: *Theatre* receives a transaction context from the initiator *NightOut*

STEP 4: *Theatre* accepts to participate in the process. It requests to be registered in the transaction, thus it sends *Register* message to *WScorr11*

STEP 5: *WScorr11* receives *Register* message from *Theatre* and registers *Theatre* in the transaction. It sends *RegisterResponse* message to *Theatre*

STEP 6: *NightOut* sends the application data to *Theatre*

STEP 7: *Theatre* successfully completes its activity. *Theatre* sends *Completed* message to notify its outcome to the Coordinator *WScorr11*

STEP 8: *Theatre* has successfully completed its activity. *Theatre* notifies the results and leaves the transaction. *Theatre* sends *Close* message to notify the Coordinator *WScorr11* which in turn replies with a *Closed* message

Fig. 2.10 Test scenario for theatre service

2.5 Discussion

This section gives a critical overview of the proposed framework and illustrates its merits and demerits. The prototype system implements the main phases of the testing process. But it still lacks full automation of the overall process. For instance, the tester has to model the given WS standard under test according to the roles defined by the framework such as Initiator, Coordinator, Executor and Terminator. Information on each service such as its URL or the transaction standard used has to be provided by the tester. With such information, the framework automatically generates the abstract test cases and maps them to WS transaction standards. Further, the tester has to manually execute the test scenario in order to get the actual systems outcome. The actual systems outcome is provided to the prototype system by the tester which then automatically compares both outcomes in order to detect faults. Despite the semi-automatic nature of the framework, it still helps the tester in two ways: (i) defining specific test cases for WS transactions and (ii) automating some of the most tedious and error-prone phases of testing. Our future work includes the full automation of the overall testing process.

The framework relies on the capability of the proposed generic transaction model in order to capture the behaviour of existing transaction standards. The generic model,

Test Case Ids			Description
Restaurant	Theatre	Taxi	
Rest_1	Thr_1	Tax_1	Cancel the in-progress booking (of restaurant, theatre, taxi). That is, a service is started but has not confirmed the reservation yet.
Rest_2	Thr_2	Tax_2	Service is executed but is unsuccessful as there is no taxi or seat available in restaurant or theatre
Rest_3	Thr_3	Tax_3	Cancel (undo) booking by executing the compensating action
Rest_4	Thr_4	Tax_4	Confirm successful booking after the commit of the transaction
Rest_5	Thr_5	Tax_5	Successfully confirm the theatre tickets booking when the other services reservations have been undone through compensating transactions
Rest_6	Thr_6	Tax_6	Abort service before it has started its execution
Rest_7	Thr_7	Tax_7	Failure occurs during the compensation process of completed booking
Rest_8	Thr_8	Tax_8	Use retry action if there is a failure during the booking process

Fig. 2.11 Test cases for the Night Out services

Table 2.2 Test execution results

Executor	Generated test cases	Pass	Fail	Blocked
Restaurant	8	3	3	2
Theatre	8	3	3	2
Taxi	8	3	3	2

presented in this chapter, has been designed after an in-depth study of the existing solutions of WS transactions. Currently BTP, WS-BA and WS-TXM transaction standards have been modelled using the generic transaction model. Our analysis revealed that though these standards are incompatible between each other, they are based on same theoretical concepts. Thus they can be modelled using the roles specified in the generic transaction model. In future, we intend to study the capability of the generic transaction model to model transaction-based applications running under non-transaction standards such as [26].

In terms of the test case generation, the proposed framework applies transition test criterion that ensures the coverage of all transitions and states specified in the generic transaction model. The framework however does not guarantee the code coverage. As a part of the future research work we plan to enhance the prototype system in order to monitor the execution of the code.

<pre><soap:Envelope xmlns:soap="http://schemas.xml soap.org/soap/envelope/"> <soap:Header> <Action xmlns="http://www.w3.org/2005 /08/addressing"> http://docs.oasis- open.org/ws- tx/wsba/2006/06/Close </Action></pre> <p>(a) Expected outcome</p>	<pre><soap:Envelope xmlns:soap="http://schemas.x mlsoap.org/soap/envelope/"> <soap:Header> <Action xmlns="http://www.w3.org/200 5/08/addressing"> http://docs.oasis- open.org/ws- tx/wsba/2006/06/Compensate </Action>></pre> <p>(b) Actual outcome</p>
---	--

Fig. 2.12 Fault in message exchange

<pre><wscoord:CoordinationType> http://docs.oasis- open.org/ws- tx/wsba/2006/06/MixedOutcome </wscoord:CoordinationType></pre> <p>(a) Expected outcome</p>	<pre><wscoord:CoordinationType> http://docs.oasis- open.org/ws- tx/wsba/2006/06/AtomicOutcome </wscoord:CoordinationType></pre> <p>(b) Actual outcome</p>
--	---

Fig. 2.13 Fault in registration process

```
private boolean testBusinessActivity(int restaurantSeats, int
theatreCircleSeats, int theatreStallsSeats, int
theatreBalconySeats, boolean bookTaxi) throws Except
{
System.out.println("CLIENT: obtaining
userBusinessActivity...");      UserBusinessActivity uba =
UserBusinessActivityFactory.userBusinessActivity();
```

Fig. 2.14 Fault identification: transaction setup

2.6 Conclusion

This chapter investigated into the issue of testing the WS Transactions. In it we designed, developed and evaluated the generic framework which is capable of dynamically modelling different WS transaction models and standards. The framework exploits model-based testing technique in order to automatically generate test cases for testing the WS transaction standards. The framework is implemented as a prototype system with which various test cases were automatically generated and mapped to different WS transaction standards. The evaluation was performed using the case study of *Night Out*, which is an open source application provided by Jboss [19].

```

public class UserBusinessActivityImple extends User
BusinessActivity {

    public void begin(int timeout) throws
WrongStateException, SystemException {
        try {
            if (_contextManager.currentTransaction() != null)
                throw new WrongStateException();
            CoordinationContextType ctx = _factory.create(
                BusinessActivityConstants.WSBA_PROTOCOL_ATOMIC_OUTCOME,
                null, null);

```

Fig. 2.15 Fault identification: protocol implementation

The experiments show that our framework can effectively be used to define different test cases as well as test the different WS transactions models and standards.

References

1. Alrifai, M., Dolog, P., Balke, W.T., Nejdl, W.: Distributed management of concurrent web service transactions. *Services Computing, IEEE Transactions on* **2**(4), 289–302 (2009)
2. Bhiri, S., Godart, C., Perrin, O.: Transactional patterns for reliable web services compositions (2006)
3. Bhiri, S., Perrin, O., Godart, C.: Ensuring required failure atomicity of composite web services (2005)
4. Bozkurt, M., Harman, M., Hassoun, Y.: Testing web services: A survey. Tech. rep., Department of Computer Science, King's College London (2010)
5. Canfora, G., Penta, M.: *Service-Oriented Architectures Testing: A Survey*, pp. 78–105. Springer-Verlag (2009)
6. Casado, R., Tuya, J., Godart, C.: Dependency-based criteria for testing web services transactional workflows. In: *Next Generation on Web Services Practices*, pp. 74–79. IEEE (2011)
7. Casado, R., Tuya, J., Younas, M.: Testing long-lived web services transactions using a risk-based approach. In: *10th International Conference on Quality Software*, pp. 337–340. IEEE Computer Society, 1849260 (2010)
8. Casado, R., Tuya, J., Younas, M.: Evaluating the effectiveness of the abstract transaction model in testing web services transactions. *Concurrency and Computation: Practice and Experience* pp. n/a–n/a (2012)
9. Casado, R., Tuya, J., Younas, M.: Testing the reliability of web services transactions in cooperative applications (2012)
10. Cavalli, A., Cao, T.D., Mallouli, W., Martins, E., Sadovykh, A., Salva, S., Zadi, F.: Webmov: A dedicated framework for the modelling and testing of web services composition. In: *IEEE International Conference on Web Services* (2010)
11. Chrysanthis, P.K., Ramamritham, K.: Synthesis of extended transaction models using acta. *ACM Trans. Database Syst.* **19**(3), 450–491 (1994)
12. Elmagarmid, A.K.: *Database transaction models for advanced applications*. Morgan Kaufmann Publishers (1992)

13. Emmi, M., Majumdar, R.: Verifying compensating transactions. In: International Conference Verification, Model Checking, and Abstract, Interpretation, pp. 29–43 (2007)
14. Gaaloul, W., Rouached, M., Godart, C., Hauswirth, M.: Verifying composite service transactional behavior using event calculus (2007)
15. Garcia-Molina, H., Salem, K.: Sagas (1987)
16. Gioldasis, N., Christodoulakis, S.: Utml: Unified transaction modeling language. In: The Third International Conference on Web Information Systems Engineering (2002)
17. GlassFish: Jax-ws (2005)
18. Hrastnik, P., Winiwarter, W.: Using advanced transaction meta-models for creating transaction-aware web service environments. *International Journal of Web Information Systems* (2005)
19. Jboss: Jboss transactions (2006)
20. Joyce El, H.: Tqos: Transactional and qos-aware selection algorithm for automatic web service composition. *IEEE Transactions on Services Computing* **3**, 73–85 (2010)
21. Lanotte, R., Maggiolo-Schettini, A., Milazzo, P., Troina, A.: Design and verification of long-running transactions in a timed framework. *Science of Computer Programming* pp. 76–94 (2008)
22. Li, J., Zhu, H., He, J.: Specifying and verifying web transactions. In: International conference on Formal Techniques for Networked and Distributed Systems, pp. 149–168 (2008)
23. Moss, E.: Nested transactions: An approach to reliable distributed computing. Massachusetts Institute of Technology (1981)
24. OASIS: Business transaction protocol (2004)
25. OASIS: Web services composite application framework (2005)
26. OASIS: Web services business process execution language v2.0 (2007)
27. OASIS: Web services coordination, <http://docs.oasis-open.org/ws-tx/wscoor/2006/06> (2007)
28. OASIS: Web services atomic transaction (2009)
29. OASIS: Web services business activity (2009)
30. Offutt, J., Liu, S., Abdurazik, A., Ammann, P.: Generating test data from state-based specifications. *Journal of Software Testing, Verification and Reliability* **13**(13), 25–53 (2003)
31. Pu, C., Kaiser, G.E., Hutchinson, N.C.: Split-transactions for open-ended activities (1988)
32. Reuter: Contracts: A means for extending control beyond transaction boundaries. *Proceedings of the 3rd International Workshop on High Performance Transaction Systems* (1989)
33. Weikum, G., Schek, H.J.: Concepts and applications of multilevel transactions and open nested transactions. *Database transaction models for advanced applications*. Morgan Kaufmann Publishers Inc. (1992)
34. Younas, M., Eaglestone, B., Holton, R.: A formal treatment of a sacred protocol for multidatabase web transactions. *Database and Expert Systems Applications* **1873**, 899–908 (2000)
35. Zhang, A., Nodine, M., Bhargava, B., Bukhres, O.: Ensuring relaxed atomicity for flexible transactions in multidatabase systems. *ACM, SIGMOD Record* (1994)

Advanced Web Services

Bouguettaya, A.; Sheng, Q.Z.; Daniel, F. (Eds.)

2014, XIX, 633 p., Hardcover

ISBN: 978-1-4614-7534-7